A correct-by-construction model for asynchronously communicating systems

Zoubeyr Farah[1] · **Yamine Ait-Ameur**[2] · **Meriem Ouederni**[2] · **Kamel Tari**[1]

Abstract The design and verification of distributed software systems is often hindered by their ever-increasing com-plexity and their asynchronous operational semantics. This article considers choreography specifications for distributed systems to reduce that complexity. We use labelled state-transitions systems as ground model for both choreographies and the corresponding distributed systems. Based on Event-B method, we propose a stepwise correct-by-construction model to build asynchronous distributed systems which a pri-ori realise their choreographies. We rely on a sufficient and necessary realisability condition and we apply several refine-ment steps w.r.t. that condition to generate the distributed peers. The first refinement returns peer behaviours obtained by synchronous projection. The previously computed system is then refined into its asynchronous version using unbounded FIFO buffers. We prove, thanks to invariant preservation, that a sequence of exchanged messages is preserved at each refinement step. We provide a formalised proof of a realis-ability algorithm for deterministic choreographies. Besides that, our contribution is twofold: the approach is a priori and the problackposed solution scales up to any number of peers communicating with each other.

✉ Yamine Ait-Ameur
yamine@enseeiht.fr

Zoubeyr Farah
zoubeyr.farah@gmail.com

Meriem Ouederni
meriem.ouederni@n7.fr

Kamel Tari
tarikamel59@gmail.com

[1] LIMED Laboratory, Department of Computer Science, University of Bejaia, Bejaïa, Algeria

[2] IRIT-INPT/ENSEEIHT, Institut de Recherche en Informatique de Toulouse, Toulouse, France

Keywords Conversation protocols · Correct-by-construction realisability · Proof-based-formal methods · Event-B · Refinement · Asynchronous communication

1 Introduction

Context In software engineering, choreographies are originally inspired from "Bob and Alice" notation [29] and they describe, from a global point of view, the interaction among endpoint distributed peers[1] running concurrently. The choreography paradigm is well advocated as a support which alleviates the complexity when designing, verifying and implementing distributed but potentially complex systems. Here, the interaction often consists in the set exchanged messages (i.e., sent and received) between interacting peers. Then, the choreography specifies the set of conversations, i.e. conversation protocol or CP for short, that is the allowable order of messages exchanged between the peers. Examples of such conversation-based languages [10] are collaboration diagrams of UML notation and message sequence chart (MSC) graphs, business process languages like BPMN or the choreography description languages such as CDL in service-oriented architectures or multi-party session types or global types. Given a choreography specification, the local behaviours of endpoint peers can be computed by projection of global conversations into local ones. However, it is required that the distributed peers must behave exactly as specified in their CP. This problem is known as the *realisability* problem [7,10]. It requires checking that the sequences of messages defined at the conversation level are

[1] The generic term peer is used in our article. It refers to a set of distributed software components or services that can be composed and communicating together in a complex system.

equal to those produced by the different peers interactions, i.e., the peer behaviours conform to the CP. Notice that several message-passing systems interact asynchronously such that sent messages are stored in FIFO buffers at receiver side. Later, once being ready, the receiver consumes messages available at the head position of its reception buffer. Although peer state space can be finite, their interaction can result in an infinite state space system since buffers can grow infinitely. Thus, CP realisability can be undecidable in the most general case.

CPs realisability has been studied for different previously mentioned formalisms, e.g., MSC [7], collaboration diagrams [12], BPMN 2.0 choreographies [31], Erlang contracts [8], Singularity channels [10], session types [16]. In particular, the work stated in [10] defines a sufficient and necessary condition under which CPs realisability can be checked even if systems interact asynchronously through unbounded FIFO buffers. Such a condition relies on equivalence checking between CP and its distributed version. This check is possible for a class of distributed systems, called synchronisable, meaning that the order of the exchanged messages in the system is independent of the fact that operational semantics is asynchronous or synchronous. This is recognised as synchronisability checking and it is used to verify well-formedness which means that any sent message will be eventually received in asynchronous communication. Model checking techniques have been set up to handle the verification in the proposed formal approaches. However, as done in [10] most of existing work follows a posteriori realisability checking where local peer behaviours are projected first and then using model checking techniques their composition is checked against the global behaviour described by the choreography, i.e. equivalence checking. Last, following model checking spirit, state explosion is a major issue for verifying systems with reasonable state space.

Verifying and enforcing choreography realisability is an active research area with a lot of recent results, e.g. [13,15,16,20,24,27,30,32,34]. However, most of existing techniques are constrained by the limit of model checking techniques which is state-explosion. As a result, the number of communicating peers is also limited and the complex behaviour of distributed systems is seldom faced.

Our approach In this article, we address the realisability problem based on a priori verification techniques, using refinement and proof-based formal methods. We formally describe the choreographies and peers behaviours, i.e. order of message exchanges, using labelled state-transition systems. These systems suit well for various formal modelling and verification approaches like model checking or proof and refinement-based formal techniques. We assume that peers communicate asynchronously such that messages are exchanged over (possibly) unbounded FIFO buffers. We suggest a stepwise correct-by-construction development approach that builds a distributed system from a given CP where the realisability condition introduced in [10] is strictly satisfied. We use the Event-B method to develop our approach as follows: First a CP is formalised; then two refinement steps describing, respectively, a synchronous and an asynchronous realisation are given. We prove, thanks to invariant preservation, that the sequence of exchanged messages is preserved at each refinement step. These refinements ensure the correctness of our approach, particularly synchronisability and well-formedness. From a foundational and theoretical viewpoint, we provide a correctness proof for the realisability algorithm given in [10]. Moreover, the refinement and proof-based approach is supported by Event-B and developed in the RODIN platform. Besides advocating a solution for enforcing realisability which is not a new contribution in itself, we suggest a scalable verification method thanks to the ability of handling arbitrary sets and values, i.e. an arbitrary set of communicating peers and exchanged messages in our case. Thus, we avoid the state-explosion issue and better handle the complexity of distributed systems.

Main contribution To sum up, the main contributions of our article are as follows:

- We suggest an a priori method for realising choreographies where peers are communicating asynchronously via (possibly) unbounded FIFO buffers.
- Distributed peers are computed from a choreography specification by refinement w.r.t. a sufficient and necessary realisability condition identified in [10].
- Our proposal gives a proof of correctness of the realisability algorithm proposed in [10].
- Our approach is scalable without restriction on the number of communicating peers, i.e., any number of peers and messages can be considered.
- Using our refinement-based approach, several asynchronous systems can be built from their synchronous version.

Our approach brings many advantages for today's distributed systems such as "web-based applications", e.g., e-government, e-commerce, e-learning, online health care systems, "cloud computing" and even "cyber-physical systems". Asynchronous communication is often adopted as operational semantics. The use of our techniques would considerably alleviate the ever-increasing complexity when designing and checking such systems.

The remainder of this article is structured as follows: The next section presents all the formal notations on which our approach relies. Section 3 presents a case study, borrowed from [20]. It is used to illustrate our approach throughout this article. Section 4 presents the main features of the Event-B method used in our contribution. Section 5 overviews our application of Event-B to the realisability problem. The formal development is then detailed in Sect. 6. We discuss

in Sect. 7 some criteria related to our approach. Section 8 presents a positioning of our approach related to other approaches. Last, we sum up this work and present some challenging perspectives in Sect. 9.

2 Formal notations

In this section, we introduce the formal definitions of peer specification, conversation protocols, synchronous and asynchronous systems including the realisability conditions.

2.1 Basic definitions

Definition 1 (*Peer*) A peer is a LTS $\mathscr{P} = \langle S, s^0, \Sigma, T \rangle$ where S is a finite set of states, $s^0 \in S$ is the initial state, $\Sigma = \Sigma^! \cup \Sigma^?$ is a finite alphabet partitioned into a set of send and receive messages, and $T \subseteq S \times \Sigma \times S$ is a transition relation. We denote a send message action as $m!$ for a message $m \in \Sigma^!$ and a receive message action as $m?$ for $m \in \Sigma^?$.

Definition 2 (*Conversation protocol*). A conversation protocol CP for a set of peers, $\{\mathscr{P}_1, \ldots, \mathscr{P}_n\}$ with $n \geq 2$, is a LTS

$$\text{CP} = \langle S_{\text{CP}}, s^0_{\text{CP}}, L_{\text{CP}}, T_{\text{CP}} \rangle$$

where

- S_{CP} is a finite set of states and
- $s^0_{\text{CP}} \in S_{\text{CP}}$ is the initial state,
- L_{CP} is a set of labels where a label $l \in L_{\text{CP}}$ is a tuple $(\mathscr{P}_i, m, \mathscr{P}_j)$ such that $\mathscr{P}_i$ and $\mathscr{P}_j$ are the sending and receiving peers, respectively; $\mathscr{P}_i \neq \mathscr{P}_j$, and m is a message on which those peers interact;
- finally, $T_{\text{CP}} \subseteq S_{\text{CP}} \times L_{\text{CP}} \times S_{\text{CP}}$ is the transition relation.

We require that

- each message has a unique sender and receiver:

$$\forall (\mathscr{P}_i, m, \mathscr{P}_j), (\mathscr{P}'_i, m', \mathscr{P}'_j) \in L_{\text{CP}} : m = m' \implies \mathscr{P}_i = \mathscr{P}'_i \wedge \mathscr{P}_j = \mathscr{P}'_j$$

- peers cannot exchange reflexive messages:

$$\forall (\mathscr{P}_i, m, \mathscr{P}_j) \in L_{\text{CP}} : \mathscr{P}_i \neq \mathscr{P}_j$$

Example 1 (A conversation protocol (CP)). We illustrate on Fig. 1 a simple CP using three services $\{S_1, S_2, S_3\}$ interacting with each other by exchanging four messages $\{a, b, c, d\}$.

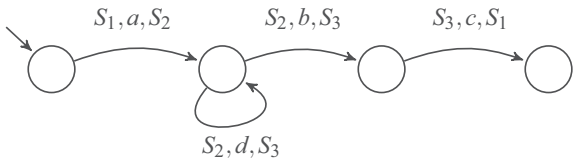

Fig. 1 An illustrative CP

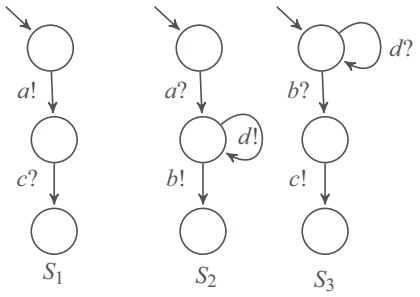

Fig. 2 The set of Peers projected from the CP of Example 1

The reflexive transition S_2, d, S_3 on the second state describes the possibility of exchanging 0 or more d messages between the sending peer S_2 and the receiving one S_3.

Definition 3 (*Projection*) Peer LTSs $\mathscr{P}_i = \langle S_i, s^0_i, \Sigma_i, T_i \rangle$ are obtained by replacing in $\text{CP} = \langle S_{\text{CP}}, s^0_{\text{CP}}, L_{\text{CP}}, T_{\text{CP}} \rangle$ each label $(\mathscr{P}_j, m, \mathscr{P}_k) \in L_{\text{CP}}$ with $m!$ if $j = i$ with $m?$ if $k = i$ and with τ (internal action) otherwise, and finally removing the τ-transitions by applying standard minimisation algorithms [23].

Example 1 (Continued) (*Projected Peers of the previous CP*) Considering the CP depicted on Figs. 1 and 2 shows the corresponding projected peers.

Definition 4 (*Synchronous system*) Given a set of peers $\{\mathscr{P}_1, \ldots, \mathscr{P}_n\}$ with $\mathscr{P}_i = \langle S_i, s^0_i, \Sigma_i, T_i \rangle$, the synchronous system

$$(\mathscr{P}_1 \mid \cdots \mid \mathscr{P}_n)$$

is the

$$LTS_s = \langle S_s, s^0_s, \Sigma_s, T_s \rangle,$$

where

- $S_s = S_1 \times \cdots \times S_n$ defines the set of global states of the described synchronous system. Each global state is defined as a tuple of states of each peer $\mathscr{P}_i$ involved in the synchronous system.
- $s^0_s \in S_s$ such that $s^0 = (s^0_1, \ldots, s^0_n)$
- $\Sigma_s = \cup_i \Sigma_i$

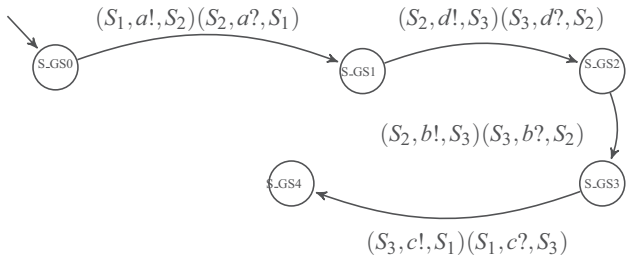

Fig. 3 A trace of the synchronous system on the projected peers of Fig. 2 conforms to the CP of Fig. 1

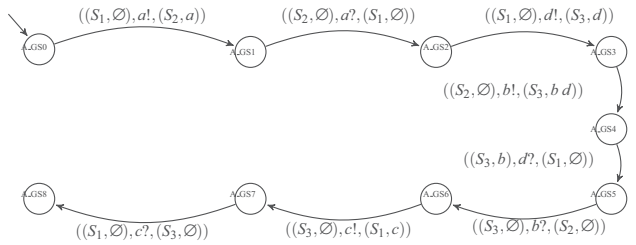

Fig. 4 A trace of the asynchronous system on the projected peers of Fig. 2 conforms to the CP of Fig. 1

- $T_s \subseteq S_s \times \Sigma \times S_s$, and for $s_s = (s_1, \ldots, s_n) \in S_s$ and $s'_s = (s'_1, \ldots, s'_n) \in S_s$, where
  - (interaction) expresses the instantaneous send–receive actions of a message $s_s \xrightarrow{m} s'_s \in T_s$ if $\exists i, j \in \{1, \ldots, n\} : m \in \Sigma_i^! \cap \Sigma_j^?$ where $\exists\ s_i \xrightarrow{m!} s'_i \in T_i$, and $s_j \xrightarrow{m?} s'_j \in T_j$ such that $\forall k \in \{1, \ldots, n\}, k \neq i \wedge k \neq j \Rightarrow s'_k = s_k$

Example 1 (Continued) (A trace of the synchronous system on the obtained projected peers) Figure 3 illustrates a trace of the synchronous system $(S_1 \mid S_2 \mid S_3)$ composed of the peers shown on Fig. 1. Notice that each arrow is labelled with an interaction, i.e. send and receive message, between sending and receiving peers, respectively, referred to as S_i and S_j where $i, j \in \{1, 2, 3\}$ and $i \neq j$. For instance, the interaction $(S_1, a!, S2)(S2, a?, S_1)$ stands for the sending of $a!$ from peer S_1 to S_2 and also the reception of $a?$ by S_2 from S_1.

Definition 5 (*Asynchronous system*) Given a set of peers $\{\mathscr{P}_1, \ldots, \mathscr{P}_n\}$ with $\mathscr{P}_i = \langle S_i, s_i^0, \Sigma_i, T_i \rangle$, and Q_i being its associated FIFO buffers, the asynchronous system

$$((\mathscr{P}_1, Q_1) \mid\mid \cdots \mid\mid (\mathscr{P}_n, Q_n))$$

is the

$$LTS_a = \langle S_a, s_a^0, \Sigma, T_a \rangle$$

defined as follows:

- $S_a \subseteq S_1 \times Q_1 \times \cdots \times S_n \times Q_n$ where $\forall i \in \{1, \ldots, n\},\ Q_i \subseteq (\Sigma_i^?)^*$ defines the set of global states of the described asynchronous system LTS_a. Each global state is defined as a tuple of pairs made of a state and the current value of the messages queue associated to each peer $\mathscr{P}_i$ involved in the projection.
- $s_a^0 \in S_a$ such that $s_a^0 = (s_1^0, \varnothing, \ldots, s_n^0, \varnothing)$
- $\Sigma_a = \cup_i \Sigma_i$
- $T_a \subseteq S_a \times \Sigma_a \times S_a$, and for $s_a = (s_1, Q_1, \ldots, s_n, Q_n) \in S_a$ and $s'_a = (s'_1, Q'_1, \ldots s'_n, Q'_n) \in S_a$ where:
  - (send) $s_a \xrightarrow{m!} s'_a \in T_a$ if $\exists i, j \in \{1, \ldots, n\} : m \in \Sigma_i^! \cap \Sigma_j^?$,
    - (i) $s_i \xrightarrow{m!} s'_i \in T_i$,
    - (ii) $Q'_j = Q_j m$,
    - (iii) $\forall k \in \{1, \ldots, n\} : k \neq j \Rightarrow Q'_k = Q_k$, and
    - (iv) $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$
  - (receive) $s_a \xrightarrow{m?} s'_a \in T_a$ if $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^?$,
    - (i) $s_i \xrightarrow{m?} s'_i \in T_i$,
    - (ii) $m\,Q'_i = Q_i$,
    - (iii) $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow Q'_k = Q_k$, and
    - (iv) $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

Example 1 (Continued) (A trace of the asynchronous system on obtained projected peers) Figure 4 gives a trace of the asynchronous system $((S_1, Q_1) \mid\mid (S_2, Q_2) \mid\mid (S_3, Q_3))$ composed of the peers on Fig. 1.

Notice that the graphical traces that are shown on all figures are denoted as follows for simplification purposes: The trace records the messages exchange ($m!$ and $m?$ for sending and receiving a message m, respectively), the sending and receiving peers, and the local queues associated with each peer. We use the notation $(S_i, Q_i), l, (S_j, Q_j)$ where S_i and S_j are the sending and receiving peers, respectively, if $l = m!$. S_i and S_j are the receiving and sending peers, respectively, if $l = m?$. We denote by Q_i and Q_j the local queues.

Last, the traces of messages stored in the local queue follow the FIFO order from right (first message) to left (last message). For instance, for (S_3, bd) the queue messages b and d are ordered as shown from right to left:

Definition 6 (*Trace*) For any LTS $\mathscr{P} =< S, s^0, \Sigma, T >$, states $p, q \in S$ and action sequence $\sigma \in \Sigma^*$, with $\sigma = a_1 \ldots a_n$ for some $n \geq 0$, we denote by $p \xrightarrow{\sigma} q$ the fact that there exist $s_0 \ldots s_n \in S$ such that $p = s_0$, $q = s_n$, and $(s_i, a_{i+1}, s_{i+1}) \in T$ for all $0 \leq i < n$. Note that for all states s it holds that $s \xrightarrow{\tau} s$.

Definition 7 (*Trace equivalence* [28]) Given two LTSs, $\mathscr{P}_1 = \langle S_1, s_1^0, \Sigma_1, T_1 \rangle$ and $\mathscr{P}_2 = \langle S_2, s_2^0, \Sigma_2, T_2 \rangle$, two states $p \in S_1$ and $q \in S_2$ are related modulo trace equivalence (p $\equiv_{tr}$ q) if and only if

- for each trace $p \xrightarrow{\sigma} p'$ in LTS $\mathscr{P}_1$ there is a trace $q \xrightarrow{\sigma} q'$ in LTS $\mathscr{P}_2$
- for each trace $q \xrightarrow{\sigma} q'$ in LTS $\mathscr{P}_2$ there is a trace $p \xrightarrow{\sigma} p'$ in LTS $\mathscr{P}_1$

Two LTSs $\mathscr{P}_1 = \langle S_1, s_1^0, \Sigma_1, T_1 \rangle$ and $\mathscr{P}_2 = \langle S_2, s_2^0, \Sigma_2, T_2 \rangle$ are equivalent if and only if their initial states are related modulo trace equivalence, i.e. $s_1^0 \equiv_{tr} s_2^0$.

2.2 Relevant properties for realisability

We now define the synchronisability which consists in an equivalence relation between LTS_s (synchronous system) and LTS_a (asynchronous system) where peers queues can be unbounded. In [10], it is proved that if the equivalence holds between 1-bounded system (for every peer $\mathscr{P}_i$ its queue size is equal to 1 and it is denoted Q_i^1) and its synchronous version, then the result holds for all bounds.

Property 1 (*Synchronisability*) Let us consider a set of peers $\{\mathscr{P}_1, \ldots, \mathscr{P}_n\}$, the synchronous system $(\mathscr{P}_1 \mid \ldots \mid \mathscr{P}_n) = (S_s, s_s^0, L_s, T_s)$, and the 1-bounded asynchronous system $((\mathscr{P}_1, Q_1^1) \mid\mid \ldots \mid\mid (\mathscr{P}_n, Q_n^1)) = (S_a, s_a^0, L_a, T_a)$.

Two states $r \in S_s$ and $s \in S_a$ are synchronisable if there exists a relation R such that $R(r, s)$ and

- for each $r \xrightarrow{m} r' \in T_s$, there exists $s \xrightarrow{m!} s' \in T_a$, such that $R(r', s')$;
- for each $s \xrightarrow{m!} s' \in T_a$, there exists $r \xrightarrow{m} r' \in T_s$, such that $R(r', s')$;
- for each $s \xrightarrow{m?} s' \in T_a$, $R(r, s')$.

The set of peers is synchronisable if $R(s_s^0, s_a^0)$ holds i.e. the initial states s_s^0 and s_a^0 are synchronisable.

Intuitively, the synchronisability property ensures that the synchronous system and the corresponding asynchronous system obtained after projection are trace equivalent, i.e. the asynchronous system does not introduce extra behaviours. In the work presented in this paper, refinement is used to guarantee this property.

We use well-formedness as a criterion to check that every message m sent by one peer and stored in a receiving queue Q_i must be eventually consumed from that queue. Well-formedness states that whenever the i-th peer buffer Q_i is non-empty, the asynchronous system can eventually move to a global state where Q_i is empty. Given an asynchronous system LTS_a built over a set of deterministic peers, well-formedness holds for this system if LTS_a is synchronisable.

Table 1 Correspondence between A_GS, S_GS and global *queue* in the traces of Figs. 3 and 4

A_GS	Global queue	S_GS
A_GS0	$\varnothing$	S_GS0
A_GS1	a	S_GS0
A_GS2	$\varnothing$	S_GS1
A_GS3	d	S_GS1
A_GS4	$b\ d$	S_GS1
A_GS5	d	S_GS2
A_GS6	$\varnothing$	S_GS3
A_GS7	c	S_GS3
A_GS8	$\varnothing$	S_GS4

Property 2 (*Well-formedness*) Let $(S_a, s_a^0, L_a, T_a) = ((\mathscr{P}_1, Q_1^1) \mid\mid \ldots \mid\mid (\mathscr{P}_n, Q_n^1))$ be an asynchronous system defined over a set of peers $\{\mathscr{P}_1, \ldots, \mathscr{P}_n\}$. (S_a, s_a^0, L_a, T_a) is well-formed, if and only if
$\forall s_a = (s_1, Q_1, \ldots, s_n, Q_n) \in S_a, \forall Q_i, \mid Q_i \mid > 0$, there exists a state $s_a' = (s_1', \varnothing, \ldots, s_n', \varnothing) \in S_a$ and a trace $\sigma \in L_a^*$ such that $s_a \xrightarrow{\sigma} s_a'$.

The defined property expresses that, at any global state s_a with non-empty local queues, another global state s_a' with empty local queues is reached. The notation $s \xrightarrow{\sigma} t$ defines a finite trace as introduced in Definition 6 in the asynchronous system to reach the destination global state.

Example 1 (Continued) (*State correspondence and global queue*) Table 1 shows the correspondence between global states (A_GSi and S_GSj) in both synchronous and asynchronous traces presented previously in Figs. 3 and 4.

The middle column of Table 1 shows the evolution of the global FIFO queue, i.e. the queue corresponding to the sent messages at the conversation protocol level. The order defined by this queue shall be preserved by the projection to ensure trace equivalence.

Property 3 (*Realisability*) A conversation protocol CP is realisable if and only if

(i) the peers $\{\mathscr{P}_1 \ldots \mathscr{P}_n\}$ computed by projection from CP are synchronisable,
(ii) $LTS_a = ((\mathscr{P}_1, Q_1) \mid\mid \ldots \mid\mid (\mathscr{P}_n, Q_n))$ is well-formed such that all Q_i are unbounded, and
(iii) LTS_a is equivalent to CP.

The reader interested in detailed definitions, theorems and proofs of this section may refer to [10].

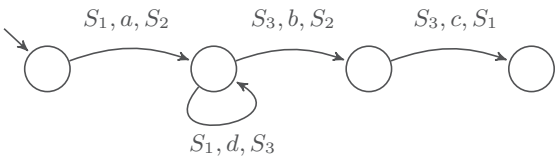

Fig. 5 A non-realisable CP

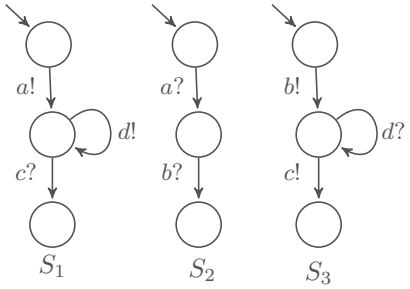

Fig. 6 The set of Peers projected from CP of Fig. 5

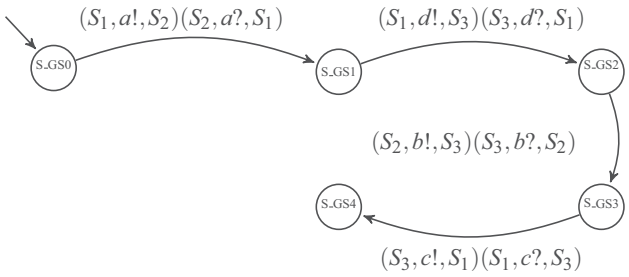

Fig. 7 A synchronous trace on the projected peers of Fig. 5 conforms to the CP of Fig. 5

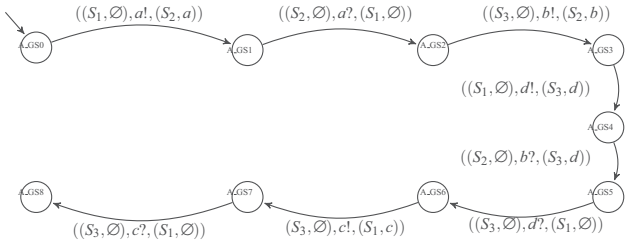

Fig. 8 An asynchronous trace on the projected peers of Fig. 5 conforms to the CP of Fig. 5

2.3 An example of a non-realisable conversation protocol

The CP presented in Example 1 illustrates a realisable CP where both synchronous and asynchronous systems are synchronisable. The traces on Figs. 3 and 4 are equivalent considering the order of sending messages. More generally, all traces that can be generated from synchronous and asynchronous systems can be shown as equivalent for that CP. We develop below another example which illustrates a non-realisable conversation protocol.

Example 2 (*A non-realisable conversation protocol* (*CP*)) Figure 5 depicts a non-realisable conversation protocol where S_1 and S_3 can send messages in the second state.

Example 2 (Continued) (*Projected Peers of the CP*) Figure 6 describes the LTS corresponding to the behaviours resulting from projection of the CP presented on Fig. 5. This projection shows that , when S_1 sends message $a!$ at initial state, both S_1 and S_3 peers can send messages, i.e. either $d!$ at S_1 intermediate state or $b!$ at S_3 initial state. However, on Fig. 5, we observe that these sendings of messages are not allowed due to the choice that must be taken between both sending actions $b!$ and $d!$ at the second state on Fig. 5. In the case if $d!$ actions can be fired, the CP specification requires that those actions $must$ be sent before firing any $b!$ action.

When two peers are involved in parallel (the sendings is possible if only one peer is involved).

Old The CP specifies that $d!$ messages $must$ be sent before sending any $b!$ message.

So, the messages sending order of the projection differs from the messages sending order of the CP. The synchronisability property is violated. The traces shown in Figs. 7 and 8 below illustrate this problem:

Example 2 (Continued) (*A synchronous trace on the obtained projected peers*) Figure 7 shows a trace of message exchanges on the synchronous system. Although the projection is not realisable, this trace respects the messaging order of the CP. The messages sending and receiving order does not appear explicitly on the synchronous system since sent messages are received instantaneously.

Example 2 (Continued) (*An asynchronous trace on the obtained projected peers*) Although all traces in the synchronous system (Fig. 7) respect the same messaging order as in the CP, the asynchronous system can hold the trace of Fig. 8 which violates this order. Such an issue is detected thanks to the synchronisability condition, i.e. both synchronous and asynchronous systems are not equivalent in that case.

Example 2 (Continued) [State correspondence and global queue].

Table 2 shows the correspondence between states (A_GSi and S_GSj) in both synchronous and asynchronous traces shown previously on Figs. 7 and 8. It also shows the evolutions of the global FIFO queue for the asynchronous system where we notice that message b appears before d while the CP requires the reverse order. Moreover, this table also shows that the trace of synchronous states (S_GS states of the right column from top to bottom) does not respect the correct synchronous trace depicted in Fig. 7. The states in **S_GS3*** and

Table 2 Correspondence between A_GS, S_GS and *queue* in the traces of Figs. 7 and 8

A_GS	Global queue	S_GS
A_GS0	$\varnothing$	S_GS0
A_GS1	a	S_GS0
A_GS2	$\varnothing$	S_GS1
A_GS3	b	S_GS1
A_GS4	$d\ b$	S_GS1
A_GS5	d	S_GS3*
A_GS6	$\varnothing$	S_GS2*
A_GS7	c	S_GS3
A_GS8	$\varnothing$	S_GS4

S_GS2* do not appear in the right order of exchanged messages and must not correspond to any asynchronous state A_GS of the asynchronous trace. Hence, trace equivalence is not preserved.

3 Case study

To illustrate our work, we use an example (see Fig. 9) borrowed from [20]. The system involves four peers: a client (*cl*), a Web interface (*int*), a software application (*appli*), and a database (*db*). We consider a CP example (*cf.* upper-side of Fig. 9) representing the designer expectation to be respected by composed peers. Each transition triple (s, m, t) describes the exchange of a message m between source and destination peers s and t, respectively. The client logs on via ($connect$) interaction between the client and the interface. It is followed by setting up the application triggered by the interface ($setup$). Then, the client may access and use the application zero or many times via $access$ interaction. Finally, the client logs out from the interface ($logout$) and the application stores relevant log information in the database (log). The lower-side of Fig. 9 shows one possible implementation for the peer behaviour, i.e. *cl*, *int*, *appli*, and *db* corresponding to CP. Each transition is labelled either by a send ($m!$) or receive ($m?$) of a message m.

Although the peers realise CP considering synchronous communication, realisability does not hold if they communicate asynchronously through FIFO buffers. Here, synchronous communication returns the sequence of messages $connect, setup, access$, whereas asynchronous communication may give rise to the sequence $connect!, access!, setup!, \ldots$. Since, the sequence $connect, access, \ldots$ is not allowed by CP, we conclude that the peers are not a correct implementation (realisation) of CP.

As a consequence, more constraints on message ordering are required to realise CPs correctly. In order to get a realisable implementation in an asynchronous and distributed setting, the approach given in [20] proposes a mediation-based solution. Every peer is augmented by a monitor which controls the conversation and ensures the coordination between the peers to meet the initial conversation model. The monitor delays the sendings and thus the receptions so as correct message sequencing is obtained. The monitors are computed iteratively to incrementally add a new synchronisation message each time the realisability check fails, i.e. using returned counter-example.

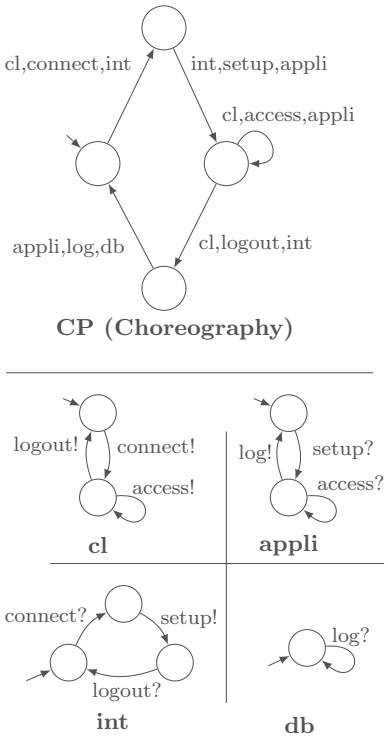

Fig. 9 Interacting Peers: global (CP) and local views (projections)

Figure 10 shows the results of two main steps of computing the monitors which are themselves peers added to the system. Guided by realizability counter-example analysis, the first step consists in performing many iterations to extend CP with a synchronisation message in each iteration. The iteration process ends up when the CP becomes realisable, e.g. the upper side of Fig. 10 shows the CP extended with the first synchronisation message. Then, the second step consists in projecting CP into peers and their monitors[2]. For illustration, the bottom side of Fig. 10 shows the monitor of *int* peer obtained by projection and which ensures CP realisability.

The upper part of Fig. 10 shows an intermediate step in computing the monitor for the *int* peer displayed on the bottom part of the same figure.

[2] The projection algorithm is given in [20].

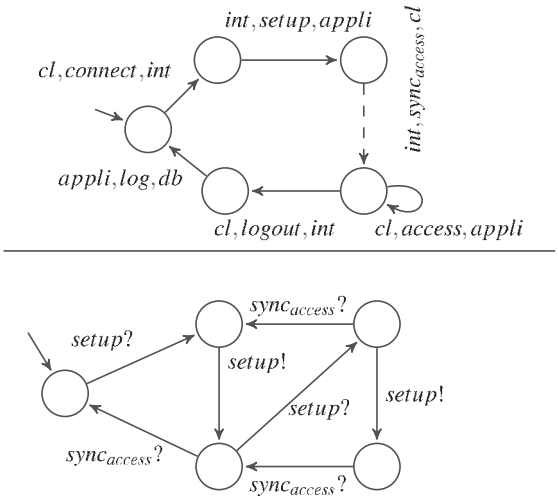

Fig. 10 Extended CP (*upper side*), and Monitor of *int* Peer (*bottom side*)

With the computing of the monitor for the *int* peer, the authors show that several synchronisation messages shall be added when communications violating realisability are iden-tified during equivalence checking.

An iterative approach allowing a designer to discover the synchronisation messages to add is proposed. This approach may require the modification of the conversation proto-col. This iterative process ends when the realisability with asynchronous communication is reached. As a consequence, equivalence has to be checked whenever synchronisation is added.

4 Event-B: a correct-by-construction method

The Event-B method [2] is a recent evolution of the B method [1]. This method is based on the notions of pre-conditions and post-conditions of Hoare [22], the weakest pre-condition of Dijkstra [17] and the substitution calculus [1]. It is a formal method based on mathematical foundations: first-order logic and set theory.

4.1 Event-B model

An Event-B model is characterised by a set of variables, defined in the VARIABLES clause that evolve thanks to events defined in the EVENTS clause. It encodes a state-transitions system where the variables represent the state and the events represent the transitions from one state to another. An Event-Bmodel consists of components of two kinds : Machines and Contexts. The Machines contain the dynamic parts (states and transitions) of a model whereas the Con-texts contain the static parts (axiomatisation and theories) of a model. AMachine can be refined by another one, and a context can be extended by another one.

Context *ctxt_id_1*	Machine *mach_id1*
Extends *ctxt_id_2*	Refines *mach_id2*
Sets s	Sees *ctxt_id1*
Constants c	Variables v
Axioms $A(s,c)$	Invariants $I(s,c,v)$
Theorems $T(s,c)$	Theorems $T(s,c,v)$
End	Variant $V(s,c,v)$
	Events $<$ *event_list* $>$
	End

Fig. 11 The structure of an Event-B development

Moreover, a Machine can see one or several Contexts.

A Context is defined by a set of clauses (left side of Fig. 11) as follows:

- CONTEXT represents the name of the component that should be unique in a model.
- EXTENDS declares the Context(s) extended by the described Context.
- SETS describes a set of abstract and enumerated types.
- CONSTANTS represents the constants used by a model.
- AXIOMS describes, in first-order logic expressions, the properties of the elements defined in the CONSTANTS and SETS clauses. Types and constraints are described in this clause as well.
- THEOREMS are logical expressions that can be deduced from the axioms.

Similarly to Contexts, Machines are defined by a set of clauses (Right side of Fig. 11):

- MACHINE represents the unique name of the component in an event-B model.
- REFINES declares the Machine refined by the described Machine.
- SEES declares the list of Contexts imported by the described Machine.
- VARIABLES represents the state variables of the model. Refinement may introduce new variables in order to enrich the described system.
- INVARIANTS describes, using first-order logic expressions, the properties of the variables defined in the VARIABLES clause. Typing information, functional and safety properties are usually described in this clause. These properties shall remain true in the whole model. Invariants need to be preserved by events. It also expresses the gluing invariant required by each refinement.
- THEOREMS defines a set of logical expressions that can be deduced from the invariants.
- VARIANT introduces a natural number or finite set that the "convergent" events must strictly make smaller at every execution.

Event evt =	Any x Where $G(s,c,v,x)$ Then $v :\mid BA(s,c,v,x,v')$ End

Fig. 12 Event structure

$<variable\_id>$	$:=$	$<expression>$	(1)
$<variable\_id\_list>$	$:\mid$	$<before\_after\_predicate>$	(2)
$<variable\_id>$	$:\in$	$<set\_expression>$	(3)

Fig. 13 The three kinds of actions for defining an event

- EVENTS defines a list of events (transitions) that can occur in a given model. Each event is characterised by its guard and is described by a set of actions (substitutions). Each Machine must contain the "*Initialisation*" event. The events occurring in an Event-B model affect the state described in VARIABLES clause. An event consists of the following clauses (Fig. 12):
  - **Refines** declares a list of events refined by the described event.
  - **Any** lists a set of parameters of the event.
  - **Where** expresses a set of guards for the event. An event can be fired when its guard becomes to true. If several guards of events become true, only a single event is fired with a non-deterministic choice.
  - **Then** contains a set of actions of the event that are used to modify variables.

Event-B offers three kinds of actions that can be deterministic or non-deterministic (Fig. 13). For the first case, the deterministic action is represented by the "*assignment*" operator $:=$ that modifies the value of a variable. This operator is illustrated by the action (1). The non-deterministic action (2) represents the "*before-after*" operator (also named before–after predicate) acting on a set of variables whose effect is represented by a predicate, expressing the relationship between the contents of variables before and after triggering the action. Finally, action (3) represents the non-deterministic choice operator, acting on a variable, by modifying its content with an undetermined value in a set of values.

The Rodin[3] platform [3,33] developed in the context of the Rodin project provides a set of tools to support Event-Bdevelopment.

4.2 Proof obligation rules

Proof obligations (PO) are associated with any Event-B model. They are automatically generated. *The proof obligation generator plugin* in the Rodin platform [3,33] generates them. These POs need to be proved to ensure the correctness of developments and refinements. The obtained PO can be proved automatically or interactively using *the prover plugin* in the Rodin platform.

The rules for generating proof obligations follow the substitution calculus [1,2] close to the weakest precondition calculus [17]. In order to define some proof obligation rules, we use the notations defined in Figs. 11 and 12 where s denotes the sets, c the constants, and v denotes the variables of the Machine. Seen axioms are denoted by $A(s,c)$ and theorems by $T(s,c)$, whereas invariants are denoted by $I(s,c,v)$ and local theorems by $T(s,c,v)$. For an event evt, the guard is denoted by $G(s,c,v,x)$ and the action is denoted by the before–after predicate $BA(s,c,v,x,v')$ (the action (2) of Fig. 13).

Definition 8 (*The theorem proof obligation rule*) This rule ensures that a proposed context or machine theorem is indeed correct.

$$A(s,c) \Rightarrow T(s,c)$$
$$A(s,c) \wedge I(s,c,v) \Rightarrow T(s,c,v)$$

Definition 9 (*Invariant preservation proof obligation rule*) This rule ensures that each invariant in a machine is preserved by each event.

$$A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x) \wedge BA(s,c,v,x,v') \Rightarrow I(s,c,v')$$

Definition 10 (*Feasibility proof obligation rule*) The purpose of this proof obligation is to ensure that a non-deterministic action is feasible.

$$A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x) \Rightarrow \exists v'.BA(s,c,v,x,v')$$

Definition 11 (*The numeric variant proof obligation rule*) This rule ensures that under the guards of each convergent or anticipated event, a proposed numeric variant is indeed a natural number.

$$A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x) \Rightarrow V(s,c,v) \in \mathbb{N}$$

Definition 12 (*The variant proof obligation rule*) This rule ensures that each event decreases the proposed numeric variant.

$$A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x) \wedge BA(s,c,v,x,v') \Rightarrow V(s,c,v') < V(s,c,v)$$

There are other rules for generating proof obligations to prove the correctness of refinement. These rules are given in [2].

[3] http://www.event-b.org/.

4.3 Semantics of Event-B models

The new aspect of the Event-B method [2], in comparison with classical B [1], is related to the semantics. Indeed, the events of a model are atomic events of a state-transition system. The semantics of an Event-Bmodel is trace-based semantics with interleaving. A system is characterised by the set of licit traces corresponding to the fired events of the model which respect the described properties. The traces define a sequence of states that may be observed by properties. All the properties will be expressed on these traces.

4.4 Refinement of Event-B models

The refinement operation [4] offered by Event-B enables stepwise model development. A transition system is refined into another transition system with more and more design decisions while moving from an abstract level to a less abstract one. A refined Machine is defined by adding new events, new state variables and a gluing invariant. Each event of the abstract model is refined in the concrete model by adding new information expressing how the new set of variables and the new events evolve. All the new events appearing in the refinement refine the *skip* event of the refined Machine. Refinement preserves the proved properties and, therefore, it is not necessary to prove them again in the refined transition system, usually more complex. The preservation of the properties results from the proof of the gluing invariant in the refined machine.

Observe that different refinements may refine a given abstract machine. Each refinement machine corresponds to a possible behaviour, implementation or concretisation of the abstract machine. Thus, several candidate refinements are offered for a given abstract machine. This observation will be used next to characterise the set of correct web services compositions that behaves as described by an abstract web service composition.

The Event-Bmethod proved its capability to represent event-based systems like railway systems, embedded systems or web services. Moreover, complex systems can be gradually built in an incremental manner by preserving the initial properties (invariant preservation).

5 Overview of our refinement-based realisability

The approach developed in [20] enforces choreography realisability using counter-example guided analysis. However, this is an a posteriori method since equivalence between the distributed system and its CP is checked once the projection is performed. Realisability enforcement consists in iterative computation of distributed monitors which ensure that peers communication do respect the same messaging order as required by the CP. Particularly, this iterative computation ends up once the final system composed by peers and their respective monitors are checked to be equivalent to its CP. This approach is implemented using model checking techniques.

Our proposal is different since it considers a priori approach based on a correct-by-construction development, using Event-B method. It relies on three pillars: (i) given a CP, peer projections are correct by construction. *Refinement* guarantees that the resulting peers a priori realise the CP. There is no need to augment the system by monitors; (ii) then, our projection obtained by refinement preserves relevant *invariants* (*and particularly gluing invariants*) that carry the conditions to preserve realisability in the asynchronous communication; (iii) last, based on *proof-based techniques*, we are able to handle arbitrary sets and values, i.e. any number of peers and exchanged messages, ensuring thus scalability.

We implemented our solution on the Rodin platform [3] which supports Event-B modelling and reasoning. Finally, we applied our proposal to the case study used in [20]. The model checking of such a case showed that this is not realisable due to violation of messaging order in the asynchronous system w.r.t. to its CP. In [20], the authors computed a set of monitors to enforce system realisability. We show in the following sections how our solution is applied to the same case study where realisability is enforced by refinement with no need for explicit monitors.[4]

5.1 The Event-B contexts

The different resources needed to define our Event-B models are specified in a context which is referenced via *SEES* clause in these models. Notice that a context can be extended with additional elements.

As shown in Sect. 4.2, the proved theorems, together with the sets, constants and axioms of a context are used as hypotheses to prove the proof obligations of the machine and its refinements.

Table 3 gives an excerpt of the $LTS\_CONTEXT$. This context formalises the concepts together with their properties needed to describe the different LTSs dealt with in the models developed in next sections:

- *STATES*, *MESSAGES* and *SET_LTS* are deferred sets representing states, messages and LTSs. These sets are defined at abstract level. Their extension is given for each particular case. An example of such set extensions for the considered case study is given in Table 5.

[4] The detailed Event-B models can be downloaded from http://yamine.perso.enseeiht.fr/RealisabilityEventBModels.pdf.

Table 3 An excerpt of the LTS_CONTEXT for defining peers and their properties

LTS_CONTEXT
SETS
SET_LTS
– Set of labeled transition systems
$STATES$
– Set of possibles states for all the LTS
$MESSAGES$
– Exchanged messages between LTS
$ACTIONS$
– Send, Receive or internal actions
CONSTANTS
LTS_STATES
– Relation between LTS and their states
$Send, Receive, Internal$
– Action constants
$EXCHANGED\_MESSAGES$
– Set of exchanged messages
$LABEL$
– Set of edges relating two LTS states
$TRANSITIONS$
– Set of transitions
S_Next_States
– The transition between states
– in the synchronous communication
A_Next_States
– The transition between states in the
– asynchronous communication
...
AXIOMS
$SET\_LTS \neq \varnothing$
– The set of LTS is not empty

$FINITE(SET\_LTS) \land card(SET\_LTS) \geq 2$
– At least two LTS are needed to
– define a conversation

$LTS\_STATES \in SET\_LTS \leftrightarrow STATES$
– States are related to their LTS

$MESSAGES \neq \varnothing$
– The set of exchanged messages
– is not empty
$PARTITION(ACTION, \{Send\}, \{Receive\}, \{Internal\})$
– The set of actions is partitioned
– to contain three actions only.
...
$\forall lts1, lts2 \cdot$
$(lts1 \in dom(LTS\_STATES) \land lts2 \in dom(LTS\_STATES) \land$
$lts1 \neq lts2 \Rightarrow$
$LTS\_STATES[\{lts1\}] \cap LTS\_STATES[\{lts2\}] = \varnothing)$
– Any two LTS have disjoint sets of states
...
$EXCHANGED\_MESSAGES \in SET\_LTS \leftrightarrow MESSAGES$
– Exchanged messages are
– related to each LTS
$LABEL \subseteq ACTIONS \times MESSAGES \times SET\_LTS$
$TRANSITIONS \in (STATES \times LABEL) \nrightarrow STATES$

$S\_Next\_States \in \mathbb{P}(TRANSITIONS) \times \mathbb{P}(SET\_LTS \times STATES)$
$\nrightarrow \mathbb{P}(SET\_LTS \times STATES)$
$A\_Next\_States \in \mathbb{P}(TRANSITIONS) \times LTS\_STATES$
$\nrightarrow LTS\_STATES$
...
END

- The *ACTIONS* set contains the constant actions {*Send, Receive, Internal* } used for the labels of LTSs transitions. *Internal* stands for the τ action.
- The *LTS_STATES* is a relation associating with each LTS, its set of states. The notation $lts \mapsto s \in LTS\_STATES$ means that s is state of the LTS lts. This relation is used in next sections to model the synchronous system S_GS and asynchronous system A_GS global states.
- Labels of LTS are defined by the *LABEL* set. We note $(a \mapsto m \mapsto lts) \in LABEL$ to describe a label composed of an action a, a message m and a LTS lts. They are used to label the transitions of the transition relation.
- The $TRANSITIONS \in (STATES \times LABEL) \nrightarrow STATES$ partial function models the transition relation. It is defined for both synchronous and asynchronous systems. It takes a state and a label as parameters and returns a state. For a label l of the form $a \mapsto m \mapsto lts$, the term $s \mapsto l \mapsto s'$ denotes the transition from state s to state s' with label l (s and s' being states of the LTS lts of the label l).
- The S_Next_States and A_Next_States are two partial functions $\mathbb{P}(TRANSITIONS) \times LTS\_STATES \nrightarrow LTS\_STA\ TES$ defined for both synchronous and asynchronous systems. These functions take a transition and a global state as parameters and return the next global state. For example, the term $S\_Next\_States(\{tr\} \mapsto gs))$ returns the next global state $gs' \in LTS\_STATES$.

5.2 The refinement strategy

We present in this section our refinement-based method for enforcing CP realisability. Our work applies for all non-faulty choreographies (an example of a non-faulty choreography is given by Examples 1 and 2 where realisability can be enforced) meaning that their correspondent specifications do not present design errors. For instance, choreography specifications which involve divergent choices are considered as faulty [35]. Realisability cannot be enforced in that case, as it is impossible to control divergent choices in a distributed system without changing the local behaviour of the peers. Faulty choreographies can be identified beforehand by detecting non-confluent diamonds of interactions in the conversation protocol using the executable temporal logic (XTL) [26].

We first discard faulty choreographies[5] using the axioms of the *LTS_CONTEXT* context which characterise only choreographies that fulfil the realisability property 3. Our realisability solution is then developed following several steps. First, the CP is described by an LTS encoded into an Event-B machine as an initial specification.

[5] Detection of faulty choreographies is considered out of scope of this article.

Then a stepwise approach is used. We consider a CP described by a labelled transition system (LTS) as the initial specification. Then, the LTS is refined into a distributed system that realises CP. The first refinement defines a projection onto a set of peers. It produces a synchronous distributed system as a first realisation. At this level, the interaction is synchronous w.r.t. Definition 4.

A second refinement results in an asynchronous realisation that overcomes message order limitation, i.e. the order problem identified in Sect. 3. This refinement strengthens the conditions and the invariants through a causal order of messages fitting with the realisability conditions as given in Property 3 defined in Sect. 2.2 (see Sect. 6.3 for order preservation).

Notice that at each refinement step, thanks to invariant preservation, we also prove that the sequence of exchanged messages is preserved from one refinement to another.

The Event-B machines presented in the remainder of the article follow the same method to encode a transition system. Starting from the conversation, four main events compose the communication: initialisation of the communication, progress, internal(τ) and reset (see Table 4). As stated before, at the specification level, a first machine describes the CP.

The first refinement is a projection on peers for a synchronous communication.

A second refinement returns the asynchronous communication that realises the initial conversation defined at the top specification machine level.

To show how the different models work, the case study of Sect. 3 illustrates each refinement.

6 Realising conversation protocols

In the following, we give an outline of the formal development leading from a CP definition to a realisable projection. The Event-B models for the application to the case study can be downloaded from http://yamine.perso.enseeiht.fr/RealisabilityEventBModels.pdf.

6.1 Conversation protocols

An initial Event-B model is defined according to Definition 2. This model is characterised by two model variables:

(1) *Conversation* that records the sequence of (indexed by natural numbers) messages exchanged in the conversation and
(2) *Index* that records the message exchange order.

This model also contains four events: $initialisation$, $Interact\_Event$, $Internal\_Event$ and $Reset\_Event$ (See Table 4). $init-ialisation$ sets the conversation to the empty set. The $Interact\_Event$ represents the progress due to the send and receive actions. It is triggered when two guards $grd1$ and $grd2$ formalising the conditions of Definition 2 are fulfilled. The $Conversation$ and $Index$ variables are updated accordingly ($act1$ and $act2$). In order to have a complete description, $Internal\_Event$ events (τ-transitions) are present throughout the whole Event-B models. Finally, the $Reset\_Event$ event resets the conversation.

Table 4 Definition of the *Interact_Event* Event for Conversation Progress

Event	$initialisation \triangleq \ldots$	
Event	$Interact\_Event \triangleq$	
any		
	lts_source,	– the source lts
	$lts\_destination$,	– the target lts
	$message$	– the exchanged message
where		
grd1:	$\exists send\_st\_src, send\_st\_dest \cdot$	
	$(((send\_st\_src \mapsto$	
	$(Send \mapsto message \mapsto lts\_destination))$	
	$\mapsto send\_st\_dest)$	
	$\in TRANSITIONS)$	
grd2:	$\exists receive\_st\_src, receive\_st\_dest \cdot$	
	$(((receive\_st\_src \mapsto$	
	$(Receive \mapsto message \mapsto lts\_source))$	
	$\mapsto receive\_st\_dest)$	
	$\in TRANSITIONS)$	
then		
act1:	$Conversation := Conversation \cup \{$	
	$(lts\_source \mapsto message \mapsto lts\_destination)$	
	$\mapsto Index\}$	
act2:	$Index := Index + 1$	
End		
Event	$Internal\_Event \triangleq \ldots$	
Event	$Reset \triangleq \ldots$	

Example 3 Table 5 presents an excerpt of the $LTS\_CONTEXT\_Instantiation$ Event-B context which encodes the CP LTS depicted on Fig. 9. *LTS_CONTEXT* context defining *SET_LTS*, *STATES*, *MESSAGES*, *TRANSITIONS*, etc. is extended by $LTS\_CONTEXT\_Instantiation$. Other contexts for the case study are defined for further refinements. Figure 14 gives a graphical representation of an example of a valid trace with indexed transitions.

Remark Note that the notation for the transitions of Fig. 14 has been enriched by the $index$ variable of the Event-B models ($\cdots \mapsto index$) used to register the global order of messaging. This index is used in the next refinement to build a global queue.

Table 5 An excerpt of the Event-B context encoding the CP of the case study

CONTEXT	
	LTS_CONTEXT_Instanciation
EXTENDS	
	LTS_CONTEXT
…	
Peers:	$partition(SET\_LTS,$
	$\{cl\}, \{appli\}, \{int\}, \{db\})$
States:	$partition(STATES,$
	$\{cl\_state1\}, \{cl\_state2\},$
	$\{appli\_state1\}, \{appli\_state2\},$
	$\{int\_state1\}, \{int\_state2\},$
	$\{int\_state3\}, \{db\_state1\})$
Conversation alphabet:	$partition(MESSAGES,$
	$\{connect\}, \{access\},$
	$\{logout\}, \{log\}, \{setup\})$
Exchanged messages :	$EXCHANGED\_MESSAGES$
	$= \{cl \mapsto connect, cl \mapsto access,$
	$cl \mapsto logout, appli \mapsto setup,$
	$appli \mapsto access, appli \mapsto log,$
	$int \mapsto connect, int \mapsto setup,$
	$int \mapsto logout, db \mapsto log\}$
Process states:	$LTS\_STATES = \{$
	$cl \mapsto cl\_state1, cl \mapsto cl\_state2,$
	$appli \mapsto appli\_state1,$
	$appli \mapsto appli\_state2,$
	$int \mapsto int\_state1, int \mapsto int\_state2,$
	$int \mapsto int\_state3, db \mapsto db\_state1\}$
…	
END	

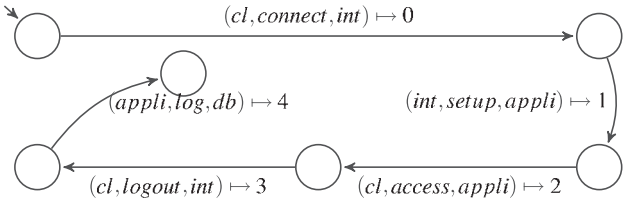

Fig. 14 Abstract level of interaction

6.2 Synchronous realisation

Our first refinement produces the synchronous projection on peers LTSs introduced in Definition 3. The transition $s \xrightarrow{m} s'$ in CP is split into a pair of actions (Send, Receive) of the form $(s_i \xrightarrow{m!} s'_i, s_j \xrightarrow{m?} s'_j)$ of a source state i and a destination state j. The communication semantics considered here is synchronous where the peers composition results in a system as introduced in Definition 4. Therefore, from the results of [10,20], we get that the projected system is well-formed.

Regarding the refined model (excerpts are shown in Tables 6, 7), some new variables are introduced. Namely

- S_GS describes the global state of synchronous system
- S_trace holds the synchronous sequences of interactions (synchronisations) split into $Send$ and $Receive$ actions (corresponding to S_Send_Trans and $S\_Receive\_Trans$ transitions) in the projected peers. For each synchronised interaction, this is referred to by S_msg in $Conversation$
- Lastly, S_index stands for the size of S_trace.

Moreover, the Event-B event $Send\_Receive\_Event$ introduced in the refined model refines the $Interact\_Event$. It encodes the projection and includes

- four parameters, namely *S_Send_Trans* and *S_Receive_Trans* transitions, the *message* (*S_msg*) which consists in the synchronisation over both send and receive actions, and *peers* (source S_lts_s and destination S_lts_d) involved in the interaction.
- Two guards $grd1$ and $grd2$ ensure that the $Send$ and $Receive$ actions can be synchronised.
- When the guards are satisfied, $act1$ updates the $Conversation$, $act2$ increases the index, $act3$ sets the new synchronous global state after consuming the message msg and $act4$ updates the synchronous conversation sequence.

Table 6 Invariants of the synchronous model

INVARIANTS	
inv1:	$S\_GS \in SET\_LTS \leftrightarrow STATES$
inv2:	$S\_trace \in TRANSITIONS \leftrightarrow \mathbb{N}$
inv3:	$S\_index \in \mathbb{N}$
inv4:	$S\_index = Index$

According to Definition 4, the witnesses ($With$ clause for event feasibility) ensure the correct projection. These witnesses correspond to correct expressions for the parameters of the $Interact\_Event$ event of the refined model.

6.2.1 Invariants for synchronous projection

The invariants defined at the synchronous projection level (see Table 6) consist of the definition of S_GS and S_trace variables and their properties. Invariant *inv4* stands for a gluing invariant that ensures that the projected behaviours are the same ones as described in the original CP. This gluing

Table 7 *Send_Receive_Event* event of the Synchronous Model

Event	$Send\_Receive\_Event \triangleq$	
REFINES	$Interact\_Event$	
any		
	S_Send_Trans,	
	$S\_Receive\_Trans$,	– The synchronous send – and receive transitions
	S_lts_s,	
	S_lts_d,	– The source and target lts
	S_msg	– The exchanged message
where		
grd1:	$\exists send\_st\_src, send\_st\_dest \cdot$ $((S\_lts\_s \mapsto send\_st\_src) \in S\_GS$ $\wedge$ $((send\_st\_src \mapsto$ $(Send \mapsto S\_msg \mapsto S\_lts\_d))$ $\mapsto send\_st\_dest) \in TRANSITIONS$ $\wedge$ $(S\_Send\_Trans = (send\_st\_src \mapsto$ $(Send \mapsto S\_msg \mapsto S\_lts\_d))$ $\mapsto send\_st\_dest))$	
grd2:	$\exists receive\_st\_src, receive\_st\_dest \cdot$ $(((S\_lts\_d \mapsto receive\_st\_src) \in S\_GS) \wedge$ $((receive\_st\_src \mapsto$ $(Receive \mapsto S\_msg \mapsto S\_lts\_s))$ $\mapsto receive\_st\_dest) \in TRANSITIONS \wedge$ $(S\_Receive\_Trans =$ $(receive\_st\_src \mapsto$ $(Receive \mapsto S\_msg \mapsto S\_lts\_s))$ $\mapsto receive\_st\_dest))$	
grd3:	$\{S\_Send\_Trans\} \mapsto S\_GS \in dom(S\_Next\_States)$	
grd4:	$\{S\_Receive\_Trans\} \mapsto$ $(S\_Next\_States(\{S\_Send\_Trans\} \mapsto S\_GS))$ $\in dom(S\_Next\_States)$	
With		
	$lts\_source = S\_lts\_s$	
	$lts\_destination = S\_lts\_d$	
	$message = S\_msg$	
then		
act1:	$Conversation := Conversation \cup$ $\{(S\_lts\_s \mapsto S\_msg \mapsto S\_lts\_d) \mapsto S\_index\}$	
act2:	$S\_index := S\_index + 1$	
act3:	$S\_GS := S\_Next\_States(\{S\_Receive\_Trans\} \mapsto$ $(S\_Next\_States(\{S\_Send\_Trans\} \mapsto S\_GS)))$	
act4:	$S\_trace := S\_trace \cup$ $(\{S\_Send\_Trans \mapsto S\_index,$ $S\_Receive\_Trans \mapsto S\_index$ $\})$	
End		

invariant ensures that the modification of $index$ variable in the $Interact\_Event$ of the abstract machine is preserved in the refinement. In other terms, S_index records, at the refined machine level, the same index as the $index$ variable of the conversation described in Table 4.

6.2.2 Events of the synchronous projection

The $Send\_Receive\_Event$ event (see Table 7) refines the $Inter\ act\_Event$, it encodes the synchronous communication as described in Definition 4. The *msg*? and *msg*!

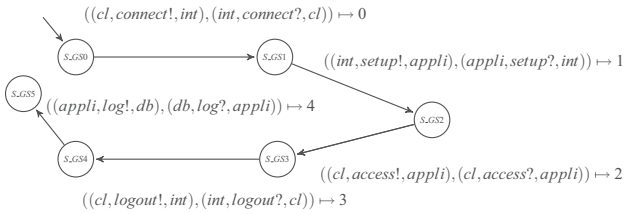

Fig. 15 A valid trace of the synchronous model

Receive and Send actions, respectively, are described in the $Send\_Receive\_\ Event$ event by S_Send_Trans and $S\_Receive\_Trans$ transitions.

The guards $grd1$ and $grd2$ require that there are source and destination LTSs (S_lts_s, S_lts_d) synchronising on a message and that synchronisation exists in the transitions set *TRANSITIONS*. Then, guards $grd3$ and $grd4$ express that a pair of S_Send_Trans and $S\_Receive\_Trans$ transitions can be synchronised from the current S_GS state. The effect of the $Send\_Receive\_Event$ event is to switch the system to the next state ($act3$), adding the pairs of actions to the S_trace ($act4$) and gluing with the variables of the conversation level using witnesses ($With$ clause).

Example 4 Going back to our case study, we show on Fig. 15 the trace resulting from the synchronous projection and corresponding to the example of Fig. 14. Here, pairs of consecutive send and receive actions illustrate the synchronous transition.

This figure shows also the evolution of the index variable by applying "act4" in Table 7. It is worth noticing that we simplify the notation on this figure for readability purposes.

Remark Here again, the notation for the transitions of Fig. 15 has been enriched by the S_index variable of the Event-B models ($\cdots \mapsto S\_index$) used to record the global order of messaging at the synchronous level. S_index is linked to the $Index$ variable of the abstraction by equality with the gluing invariant $inv4 : S\_index = Index$ of Table 6. It preserves the right ordering of messages when building a global queue.

6.3 Asynchronous realisation

The asynchronous realisation introduces FIFO queues to support asynchronous behaviours. According to Definition 5, the second refinement produces the final model of asynchronous realisation (see Tables 8, 9, 10, 11). New variables are introduced, namely A_GS for the global state in the asynchronous system; $queue$ is a FIFO queue of indexed (by the sending order) messages corresponding to sent messages not yet consumed; natural numbers $front$ and $back$ that encode the indexes of messages available in $queue$; and $Queue\_size$ corresponding to an arbitrary (unbounded) queue size. The Boolean variable $Interaction\_Completed$ is equal to 0

Table 8 Invariants of the asynchronous model

INVARIANTS	
inv1:	$queue \subseteq SET\_LTS \times MESSAGES \times \mathbb{N}$
inv2:	$back \in \mathbb{N}$
inv3:	$front \in \mathbb{N}$
inv4:	$A\_GS \in SET\_LTS \leftrightarrow STATES$
inv5:	$A\_Send\_Trans \in TRANSITIONS$
inv6:	$A\_Receive\_Trans \in TRANSITIONS$
inv7:	$A\_lts\_s \in SET\_LTS$
inv8:	$A\_lts\_d \in SET\_LTS$
inv9:	$A\_msg \in MESSAGES$
inv10:	$Interaction\_Completed \in \{0, 1\}$
inv11:	$Queue\_Size \in \mathbb{N}_1$
inv12:	$queue = \varnothing \Rightarrow front = back$
gluing_1 :	$Interaction\_Completed = 0 \Rightarrow front = S\_index$
gluing_2 :	$Interaction\_Completed = 1 \Rightarrow front = S\_index + 1$
gluing_3 :	$queue = \varnothing \wedge$ $Interaction\_Completed = 0$ $\Rightarrow$ $A\_GS = S\_GS$
gluing_4 :	$queue = \varnothing \wedge$ $Interaction\_Completed = 1$ $\Rightarrow$ $A\_GS = S\_Next\_States($ $\{A\_Receive\_Trans\} \mapsto$ $(S\_Next\_States(\{A\_Send\_Trans\} \mapsto S\_GS))$ $)$
...	

when a pair of send and receive of a given message is synchronised.

6.3.1 Invariants for asynchronous realisation

The invariant is made of two parts (see an excerpt in Table 8). One relates to the definitions of the asynchronous communications ($inv1-12$) while the second is devoted to gluing the synchronous model variables with the asynchronous ones ($gluing1-4$). It is worth noticing that these invariants describe Properties 1 and 2 defined in Sect. 2.2.

The gluing invariants are fundamental to preserve the well-formedness as well as the equivalence between asynchronous projection and CP, thus ensuring CP realisability. For instance, $gluing\_1$ and $gluing\_2$ connect the S_index of the synchronous projection with the $front$ and $back$ indexes for each send–receive pair of transitions. The global asynchronous state A_GS of the asynchronous realisation and the S_GS global state of the synchronous projection are equivalent when the queue is empty by $gluing\_3$. When a send–receive pair is synchronised ($Interaction\_Completed$) and the queue is empty, $gluing\_4$ says that the next global synchronous state S_GS will be the current global asynchronous state A_GS.

Table 9 $Send\_Event$ Event of the Asynchronous Model

Event	$Send\_Event \triangleq$	
any		
	$send\_trans$,	– The send transiton
	lts_s,	
	lts_d,	– The source and target lts
	msg	– The sent message
where		
grd1:	$\exists send\_st\_src, send\_st\_dest \cdot$ $((lts\_s \mapsto send\_st\_src) \in A\_GS \wedge$ $((send\_st\_src \mapsto$ $(Send \mapsto msg \mapsto lts\_d)) \mapsto$ $send\_st\_dest)$ $\in TRANSITIONS \wedge$ $(send\_trans = (send\_st\_src \mapsto$ $(Send \mapsto msg \mapsto lts\_d)) \mapsto$ $send\_st\_dest))$	
grd2:	$\{send\_trans\} \mapsto A\_GS \mapsto queue \in$ $dom(A\_Next\_States)$	
grd3:	$finite(queue) \wedge$ $card((\{lts\_d\} \lhd dom(queue)) \lhd queue) \leq$ $Queue\_Size$	
grd4:	$Interaction\_Completed = 0$	
then		
act1 :	$A\_GS := A\_Next\_States($ $\{send\_trans\} \mapsto A\_GS \mapsto queue$ $)$	
act2 :	$queue, back := queue \cup \{lts\_d \mapsto msg \mapsto back\},$ $back + 1$	
End		

Remark Note that first, the two invariants $gluing\_3$ and $gluing\_4$ guarantee the well-formedness property since the defined behaviour of synchronous projection is preserved in the asynchronous projection. Second, the gluing invariants 1, 2, 3 and 4 ensure the synchronisability condition expressed in Property 1. Indeed, they define the relation $R(r, s)$ that links any state r in the synchronous projection with a state s of the asynchronous realisation. Formally, the invariant describing equality between synchronous and asynchronous states corresponds to equivalence checking.

6.3.2 Events for asynchronous realisation

Following the refinement strategy for asynchronous realisation, the $Send\_Event$ and $Receive\_Event$ Event-B events are introduced. According to Definition 5, they make an explicit separation between the sending and receiving actions (allowing an interleaving satisfying the causal order defined in the refined machines).

The $Send\_Event$ event (see Table 9) sets up the next asynchronous global state A_GS in $act1$, enqueues a message in the queue, and updates the $back$ index ($act2$). The guards

Table 10 *Receive_Event* Event of the Synchronous Model

Event	$Receive\_Event \triangleq$	
any		
	$send\_trans$,	– The send transiton
	$receive\_trans$,	– The receive transiton
	lts_s,	
	lts_d,	– The source and target lts
	msg	– The received message
where		
grd1:	$queue \neq \varnothing$	
grd2:	$lts\_d \mapsto msg \mapsto front \in queue$	
grd3:	$\exists receive\_st\_src, receive\_st\_dest \cdot$ $(((lts\_d \mapsto receive\_st\_src) \in A\_GS) \wedge$ $((receive\_st\_src \mapsto$ $(Receive \mapsto msg \mapsto lts\_s)) \mapsto$ $receive\_st\_dest)$ $\in TRANSITIONS \wedge$ $receive\_trans = ($ $receive\_st\_src \mapsto (Receive \mapsto msg \mapsto lts\_s))$ $\mapsto receive\_st\_dest$ $)$	
grd4:	$\{receive\_trans\} \mapsto A\_GS \mapsto queue \in dom(A\_Next\_States)$	
grd5:	$Interaction\_Completed = 0$	
grd6:	$back > front$	
...		
grd10:	$\{send\_trans\} \mapsto S\_GS \in dom(S\_Next\_States)$	
grd11:	$\{receive\_trans\} \mapsto$ $(S\_Next\_States(\{send\_trans\} \mapsto S\_GS))$ $\in dom(S\_Next\_States)$	
grd12:	$queue \setminus \{lts\_d \mapsto msg \mapsto front\} = \varnothing \Rightarrow$ $A\_Next\_States(\{receive\_trans\} \mapsto A\_GS \mapsto queue)$ $=$ $S\_Next\_States(\{receive\_trans\} \mapsto$ $(S\_Next\_States(\{send\_trans\} \mapsto S\_GS)))$	
then		
act1 :	$A\_GS := A\_Next\_States($ $\{receive\_trans\} \mapsto A\_GS \mapsto queue$ $)$	
...		
act7 :	$front := front + 1$	
act8 :	$queue := queue \setminus \{lts\_d \mapsto msg \mapsto front\}$	
act9:	$Interaction\_Completed := 1$	
End		

allow this event to be triggered several times before a receive event is triggered.

The *Receive_Event* event (see Table 10) consumes ($act8$) the message from $queue$ according to the sending order ($front$ message is consumed). The event is triggered when the queue is not empty ($grd1$), the message is available in $queue$ of the considered peer ($grd2$ and $grd3$[6]) and the corresponding receiving state is in the next asynchronous global state ($grd4$). Guards $grd10$, $grd11$ and $grd12$ guarantee that the receiving action lead to a state equivalent to a state of the synchronous projection.

Table 11 Refined *Send_Receive_Event* Event of the asynchronous model

Event	$Send\_Receive\_Event \triangleq$
REFINES	$Send\_Receive\_Event$
where	
grd1:	$Interaction\_Completed = 1$
With	
	– Witnesses to glue the synchronous and – asynchronous states $S\_Send\_Trans = A\_Send\_Trans$ $S\_Receive\_Trans = A\_Receive\_Trans$ $S\_lts\_s = A\_lts\_s$ $S\_lts\_d = A\_lts\_d$ $S\_msg = A\_msg$
then	
act1 :	$Conversation := Conversation \cup$ $\{(A\_lts\_s \mapsto A\_msg \mapsto A\_lts\_d) \mapsto S\_index\}$
act2 :	$S\_index := S\_index + 1$
act3 :	$S\_GS := S\_Next\_States($ $\{A\_Receive\_Trans\} \mapsto$ $S\_Next\_States(\{A\_Send\_Trans\} \mapsto S\_GS)$ $)$
act4 :	$S\_trace := S\_trace \cup$ $\{(A\_Send\_Trans) \mapsto (S\_index),$ $(A\_Receive\_Trans) \mapsto (S\_index)\}$
act5 :	$Interaction\_Completed := 0$
End	

Finally, the *Send_Receive_Event* event (see Table 11) refines the previous event defined in the refined machine of the synchronous projection. It gives witnesses ($With$ clause) to glue the synchronous and asynchronous transitions, messages and peers. Actions $act1$ and $act2$ update, respectively, the conversation and the index. The next synchronous state ($act3$) is refined using the asynchronous A_Send_Trans and $A\_Receive\_Trans$ transitions. The synchronous trace is updated in $act4$ and 0 is assigned to $Interaction\_Completed$ to allow other receptions ($act5$).

Regarding the realisability property, the sequencing of the events *Receive_Event* and *Send_Receive_Event* is important. Indeed, *Send_Receive_Event* delays the next conversation transition until the received messages are consumed from $queue$ and the action $act9$ of *Receive* event which sets up $Interaction\_Completed$ to 1 is fundamental to avoid wrong realisation similar to the one identified on the case study of Sect. 3. After $act9$, the event *Send_receive_Event* is triggered permitting progress of the conversation respecting the order of sent messages.

It is worth noticing that

- the refined *Send_Receive_Event* event preserves all the properties issued from the previous refinement (synchro-

[6] The operator $\lhd$ stands for domain restriction.

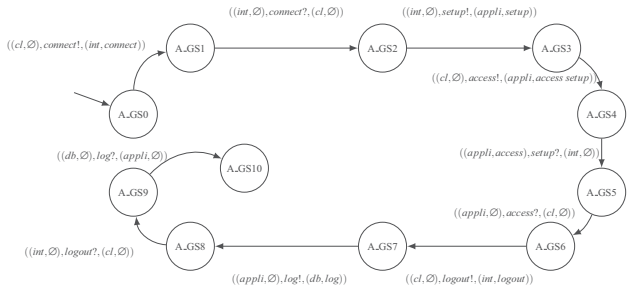

Fig. 16 Valid trace in the asynchronous realisation conform to the one of Fig. 15

nous system) and the specification (conversation protocol),

- the $Index$ and S_index for messaging order, the synchronous trace S_trace, the conversation $Conversation$ variables are preserved in the last refinement. The invariants ensure that each time a $Send\_Receive\_Event$ is triggered, then the asynchronous and the synchronous global states are equivalent.

Remark It is straightforward to build a local queue for each peer from the global queue. The ordering of messages in the global queue is provided by the S_index natural number associated with each message. Indeed, each element of the queue is represented as $lts \mapsto m \mapsto n$ to mean that the queue of the peer lts contains a message m indexed by the natural number n. Each local queue associated with each peer is obtained by projection of the global queue for each considered peer. Enqueueing and dequeueing the global queue can be implemented on local queues using classical distributed election algorithm (for example a token ring algorithm).

Example 5 Figure 16 shows a trace of the asynchronous realisation which refines the synchronous projection (shown on Fig. 15).

Table 12 shows the correspondence between the asynchronous trace and the messaging order recorded by the S_index column. The ordering defined at the synchronous projection is preserved thanks to the $queue$ variable ($global\ queue$ column in Table 12). The S_index column shows that the value of this index changes each time a received message is dequeued, i.e. a $Send\_Receive\_Event$ is triggered.

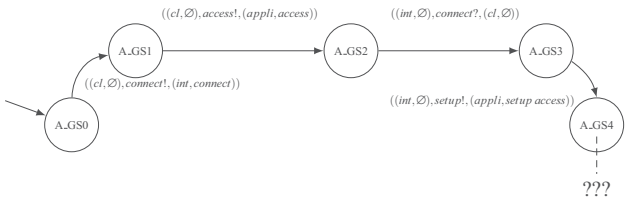

Fig. 17 A wrong trace in the asynchronous realisation

6.4 A counter example

The trace of Fig. 17, instead, is not a correct sequence of the obtained asynchronous realisation.

When the trace $(cl, connect!, int) \mapsto 0, (cl, access!, appli) \mapsto 1, (cl, connect?, int) \mapsto 2, (int, setup!, appli) \mapsto 3$ reaches A_GS4, the guard of the next ($Receive$) event to be triggered is not true. Indeed, $(cl, setup?, appli) \mapsto 1$ is required when the $queue$ set contains a $setup$ message with an index 2. Thus, it cannot satisfy the guard $grd2$ of this $Receive$ event, the next A_GS state is not reached using action $act1$ of the same event, and finally the $Send\_Receive\_Event$ event is never triggered since $Receive\_Event$ is never triggered. Therefore, S_GS2 is never reached (See Table 13).

7 Learned lessons

7.1 Proof statistics

All the models presented above have been encoded within the Rodin platform [3]. The main machine and the refine-

Table 12 Correspondence between A_GS, S_GS and $queue$ for the trace of Figs. 15 and 16

A_GS	Global queue	S_GS	S_index
A_GS0	$\varnothing$	S_GS0	0
A_GS1	$\{int \mapsto connect \mapsto 0\}$	S_GS0	0
A_GS2	$\varnothing$	S_GS1	1
A_GS3	$\{appli \mapsto setup \mapsto 1\}$	S_GS1	1
A_GS4	$\{appli \mapsto access \mapsto 2, appli \mapsto setup \mapsto 1\}$	S_GS1	1
A_GS5	$\{appli \mapsto access \mapsto 2\}$	S_GS2	2
A_GS6	$\varnothing$	S_GS3	3
A_GS7	$\{int \mapsto logout \mapsto 3\}$	S_GS3	3
A_GS8	$\{db \mapsto log \mapsto 4, int \mapsto logout \mapsto 3\}$	S_GS3	3
A_GS9	$\{db \mapsto log \mapsto 4\}$	S_GS4	4
A_GS10	$\varnothing$	S_GS5	4

Table 13 Correspondence between A_GS, S_GS and *queue* for the trace of Figs. 15 and 17

A_GS	Global queue	S_GS	S_index
A_GS0	$\varnothing$	S_GS0	0
A_GS1	$\{int \mapsto connect \mapsto 0\}$	S_GS0	0
A_GS2	$\{appli \mapsto access \mapsto 1, int \mapsto connect \mapsto 0\}$	S_GS0	0
A_GS3	$\{appli \mapsto access \mapsto 1\}$	S_GS1	1
A_GS4	$\{appli \mapsto setup \mapsto 2, appli \mapsto access \mapsto 1\}$	S_GS1	1
…	…	…	…

Table 14 RODIN proofs' statistics

Event-B model	Proof obligations	Automatic proofs	Interactive proofs
Abstract	6	6	0
Synchronous	17	13	4
Asynchronous	74	66	8
Total	97	85	12

ment led to 97 proof obligations. We noticed that 85 were proved automatically and 12 needed few interactive proof steps. Table 14 gives the details of these results. The ProB model checker [11] has been used to instantiate the Event-B models. All the models together with their instantiations can be downloaded from http://yamine.perso.enseeiht.fr/RealisabilityEventBModels.pdf.

7.2 Model properties

The models developed in this article offer interesting results. First, the correct-by-construction approach overcomes the state explosion problem thanks to the definition of deferred sets and variables (for peers). Second, thanks to the refinement relationship that preserves the properties established at the abstraction level, we have been able to guarantee the well-formedness property *entailed* from the refinement and to describe the relationship between synchronous and asynchronous states that guarantees synchronisability. Finally, instantiation of the model on specific case studies has been checked by supplying witnesses for the sets, variables and constants that fulfil the conditions expressed in both Event-B Machines and Contexts.

7.3 About scalability

The developments presented in this article are conducted within a proof- and refinement-based method. As shown in Table 14 all the proof obligations associated with the formal Event-B development presented in this article have been proved either within the automatic provers associated with the RODIN platform or using interactive proofs handled by the developer on the RODIN platform as well. The key-point related to scalability concerns the instantiation of specific choreography cases of the models presented in this article. Indeed, the development presented above is a generic one, defined at a meta-level, where the realisability and well-formedness properties of Sect. 2 act as meta-theorems.

The use of the ANY generalised substitution shows that the development considers any peers satisfying the guards and the invariants expressed in the corresponding Event-B, i.e. ANY peers that fulfil the guards and invariants can be considered. To prove the correctness of this event, among the proof obligations (as defined in Sect. 4.2), we need to prove the event feasibility of proof obligation rule of Definition 10. This proof obligation states that all the events shall be feasible. The feasibility of proof obligation expressed as $A(s,c) \wedge I(s,c,v) \wedge G(s,c,v,x) \Rightarrow \exists v'.BA(s,c,v,x,v')$ requires the proof of an existential first-order logic formula. It is well known that one of the proof rules to give a demonstration of this kind of formula consists in providing an explicit witness (a specific value or expression for the quantified variables v).

The witness can be produced by either automatic tools like model checkers, or interactive provers with the assistance of the developer (interactive proof). The first option is fully automatic. The second option does not suffer from state explosion. It can be set up when model checkers fail to produce witnesses in case of state explosion problems or infinite systems, but it requires interactive proof steps. In the case of the Event-B method, the second option consists in defining another model, which refines the one to be proved, where each event with an ANY generalised substitution is refined by an event where a witness is built by the developer for each parameter. The event refinement strategy is shown in Table 15. Once the refinement is produced, the model shall be proved using the same process as for classical Event-B models. The witnesses can be any set of peers whatever is its cardinality and such that the peers in this set may also be of any size, i.e. state and transition number.

In this article, two proof techniques, experimented in this article, have been developed:

Table 15 Proof-based instantiation

Event of the model	Instantiation by a witness
Event $evt \triangleq$	**Event** $Ref\_of\_evt \triangleq$
	Refines evt
Any	**where**
x	grd1 : $G(s, c, v, x)$
Where	**With**
grd1 : $G(s, c, v, x)$	$x = WitnessFor\_x$
Then	**Then**
act1 : $v :\mid BA(s, c, v, x, v')$	act1 : $v :\mid BA(s, c, v, x, v')$
End	**End**

- The first one uses model checking with the ProB [11] model checker. In this case, the model checker verifies the instances (set partitions i.e. set extensions defining conversation protocols and projection peers) similar to the one of Table 5 given by the user to instantiate the model.[7] This approach has been followed for the case study of Sect. 3 presented in this article as a validation case.
- The second technique relies on a proof-based approach. It is used if the model checker fails to verify the instance given by the user. This instance corresponding to the described system is given, in a refinement, as a witness for all the parameters involved in a ANY Event-B substitution. Note that this substitution produces an existential proof obligation that shall be proved interactively using the provided witness (the instance). Such an approach defines another model which refines the one presented in Sect. 6.3. It follows the refinement strategy shown in Table 15. This approach has been followed in [9].

8 State-of-the-art and discussion

Determining whether a CP is realisable is an active research area for which several solutions, e.g., [7,10,12,18,19,25,34], have been advocated in the literature. These approaches consider a CP as a formalism to describe, e.g. collaboration diagrams [12], BPMN 2.0 choreographies [31], Singularity channel Contracts [35], Message Sequence Charts (MSC) [7], Erlang contracts [8], etc.

In the following, we focus on related approaches which propose solutions for realising CPs, i.e. computation of a distributed system where the order of messages exchanged among interacting peers is equal to what is specified in a given realisable CP. Distributed peers are usually generated by projection of their behaviour from CP into their local specifications.

For example, in [13], the authors identify three principles for global descriptions under which they define a sound and complete end-point projection from a given choreography. If these rules are respected, the distributed system obtained by projection will behave exactly as specified in the choreography. Realisability of BPMN2.0 choreographies follows the same approach in [30]. In [32], the authors enable modification of choreography specifications by including two operators, namely dominated choice and loop, into their language.

During projection of these new operators, some communication messages are added to make the peers realise the choreography specification. However, these solutions prevent the designer from specifying what (s)he wants to and complicate the design by enforcing the designer to define explicit extra-constraints in the specification, e.g., by associating *dominant roles* with certain peers. In [15], the authors propose a Petri Net-based formalism for choreographies and algorithms to check realisability and local enforceability. A choreography is locally enforceable if interacting peers are able to satisfy a subset of the requirements of the choreography. To ensure this, some message exchanges in the distributed system are disabled.

In [34], the authors propose automated techniques to check the realisability of collaboration diagrams for different communication models. In case of non-realisability, messages are added directly to the peers to enforce realisability. Collaboration diagrams are much less expressive than conversation protocols, as choices and loops cannot be specified, except for repetition of the same interaction. The approach given in [20] presents an iterative and incremental computation of distributed monitors to enforce CP realisability. Here, the peers obtained by projection from a CP do not respect that CP interaction order. Hence, a monitor is computed for every peer following a counter-example guided approach. Considering communication over possibly unbounded FIFO buffers, this work realises a subclass of communicating systems, called synchronisable and well-formed. Under both conditions, CP realisability is decidable [10].

A similar approach is proposed in [27] to automatically enforce choreography realisability. The authors start from a BPMN specification which they translate into a transition system. Then, a set of *coordination delegators* is generated to prevent any undesired behaviour, e.g. deadlock, among services that can realise a choreography. The delegators are computed based on a set of rules applied following choreography analysis to detect all behaviour, e.g. concurrency that might lead to realisability violation. To resolve that, the delegators hold acknowledgement messages to enforce the same message exchange order as in the choreography. Thus, the distributed services behave exactly as described by their

[7] The contexts giving the partitioning for the case study can be found in http://yamine.perso.enseeiht.fr/RealisabilityEventBModels.pdf.

choreography. In this work, it is not clear whether the authors do handle asynchronously communicating systems.

Pistore et al. [24] propose five realisability relations depending on communication criteria that must be preserved by the distributed systems realising a choreography specification. For instance, to realise a given choreography, the peers obtained by projection from the specification must behave, through both internal actions and observable interactions with other peers, in the same order as described by the choreography. Other criteria can be considered, e.g., for synchronous realisability, two cases can be considered. When considering pairs of communicating peers running concurrently, first peers communication only preserve the order of observable interactions, or second they preserve a partial order of observable interactions. The buffer boundedness is a requirement for realisability checking. Unfortunately, to decide whether a system is buffer bounded is undecidable.

In [16], the authors use multi-party session types to describe CP for distributed systems communicating asynchronously. The end-point projection and then realisability checking require the computation of an upper bound for buffers size. Here, unbounded buffers are left as an open issue and there is no solution for realisability enforcing. The work on session types [14] is also related to the realisability of conversation protocols and has been used as a formal basis for modelling choreography language [14]. The restrictions used in session types to guarantee that local implementations follow the global interaction protocol are similar to the sufficient conditions for realisability given in [18] and they are not necessary conditions, i.e., there are realizable choreography specifications which fail the conditions given in these earlier results. In particular, both of these earlier approaches do not allow a protocol containing a state with an arbitrary initiator [21], i.e., a state where more than one peer could send the next message and the protocol works fine for either case. Protocols which are of this type and are realisable appear in practice (for example, protocols where one of the peers can cancel the interaction at an arbitrary point) and cannot be shown to be realisable with these earlier approaches.

To sum up, the aforementioned approaches to enforce CP realisability can be split into groups. The first one requires that the CP does respect a set of constraints, the second group allows CP modification by adding needed synchronisation messages, while the third one equips the distributed system with monitors.

A common feature on which most existing work on CP realisability rely is model checking techniques. Model checking, due state explosion, has the limit of not being scalable. Therefore, to be checked efficiently, CP realisability can be handled only for a limited number of peers. Our approach relies on proof-based methodologies (Event-B); it overcomes this problem. It follows the same method as the one performed on orchestration [5,6] where refinement and proof-based methods with Event-B have been set-up for building correct services compositions expressed in business process execution language (BPEL). We compute a distributed system from a correct-by-construction realisable CPs. Following refinement steps, we ensure that the asynchronous system computed from a CP is synchronisable and well-formed.

By doing so, we avoid the iterative approach to compute monitors, if necessary, as done in [20]. Based on refinement, we also guarantee that distributed peers exchange messages in same order as specified by their CP, and this is considering unbounded FIFO buffers.

9 Conclusion and future work

In this article, we addressed CP realisability using refinement and proof-based techniques using the Event-B method.[8] A CP is a low-level formal model which can be computed from other existing specification formalisms. In our work, we described a CP using a transition system which specifies the behaviours of all communicating peers, i.e., order of message exchange from a global viewpoint.

Distributed peer behaviours are then obtained through a first refinement step, by projection from a given CP, such that their synchronous composition realises CP. We, then, refine the distributed system into its asynchronous version where every peer is augmented by possibly unbounded FIFO queue. Based on refinement, we ensure that both systems are equivalent so that the asynchronous composition realises its CP specification. More precisely, two main Event-B refinements describing a synchronous and an asynchronous realisation, were necessary. The correctness of these refinements ensures synchronisability and well-formedness which are a sufficient and necessary condition for CP realisability. An illustration was given using our case study.

Moreover, our approach proved scalable enabling us to define and efficiently handle arbitrary number of interacting peers specified in a CP with no restriction on buffers sizes.

Last, our solution can be applied to several real-world applications such as service choreographies, singularity channels contracts and Erlang contracts. More generally, this can be of particular interest to implement in an efficient way distributed but complex systems where communication holds asynchronously with no restriction on buffer size.

As future work, we aim at addressing the CP realisability problem in presence of divergent choices together with multi-cast message exchanges. We also plan to study partial realisability for dynamic choreographies at run-time and hierarchical conversation protocols [36]. Finally, studying

[8] The complete Event-B models developed for this work can be downloaded from http://yamine.perso.enseeiht.fr/RealisabilityEventBModels.pdf.

adaptation and evolution aspects is one of the main extensions of this work. This study can be dealt with at two levels: i) a bottom-up approach to study the effect, on the CP, of changes in the projected peers. These changes may result from a degradation of the services offered by the peers; ii) a top-down approach corresponding to CP evolution. Checking the impact of such evolutions on realisability, synchronisability, well-formedness properties and/or trace equivalence remains an open challenge.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
4. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. Fundamenta Informaticae **77**, 1–28 (2007)
5. Ait-Sadoune, I., Ait-Ameur, Y.: A proof based approach for modelling and verifying web services compositions. In: Proc. of ICECCS'09, IEEE Computer Society, New York, pp. 1–10 (2009)
6. Ait-Sadoune, I., Ait-Ameur, Y.: Stepwise design of BPEL web services compositions. An Event B refinement based approach. In: Proc of ACIS'10, pp. 51–68. Montréal, Canada (2010)
7. Alur, R., Etessami, K., Yannakakis, M.: Realizability and verification of MSC graphs. Theor. Comput. Sci. **331**(1), 97–114 (2005)
8. Armstrong, J.: Getting Erlang to talk to the outside world. In: Proc. ACM SIGPLAN Work. on Erlang, pp. 64–72 (2002)
9. Babin, G., Ait-Ameur, Y., Pantel, M.: Correct instantiation of a system reconfiguration pattern: a proof and refinement-based approach. In: 17th IEEE International Symposium on High Assurance Systems Engineering, HASE 2016, Orlando, FL, USA, January 7–9, 2016, pp. 31–38. IEEE Computer Society Press, New York (2016)
10. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: In Proc. of POPL'12, pp. 191–202. ACM, New York (2012)
11. Bendisposto, J., Clark, J., Dobrikov, I., Karner, P., Krings, S., Ladenberger, L., Leuschel, M., Plagge, D.: Prob 2.0 tutorial. In: Proc. of 4th Rodin User and Developer Workshop, TUCS Lecture Notes. TUCS (2013)
12. Bultan, T., Fu, X.: Choreography modeling and analysis with collaboration diagrams. IEEE Data Eng. Bull. **31**(3), 27–30 (2008)
13. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: Proc. of ESOP'07, LNCS. Springer, Berlin (2007)
14. Carbone, M., Honda, K., Yoshida, N., Milner, R., Brown, G., Ross-Talbot, S.: A theoretical basis of communication-centred concurrent programming. In: Web Services Choreography Working Group Mailing List. WS-CDL working report (2006). (**to appear**)
15. Decker, G., Weske, M.: Local enforceability in interaction petri nets. In: Proc. of BPM'07, pp. 305–319. Springer, Berlin (2007)
16. Deniélou, P.M., Yoshida, N.: Buffered communication analysis in distributed multiparty sessions. In: Proc. CONCUR'10, LNCS, vol. 6269, pp. 343–357. Springer, Berlin (2010)
17. Dijkstra, E.W.: A Discipline of Programming, 1st edn. Prentice Hall PTR, Upper Saddle River (1977)
18. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and analysis of reactive electronic services. Theor. Comput. Sci. **328**(1–2), 19–37 (2004)
19. Graf, S., Quinton, S.: Knowledge-based construction of distributed constrained systems. Softw. Syst. Model., 1–18 (2015). doi:10.1007/s10270-014-0451-z
20. Güdemann, M., Salaün, G., Ouederni, M.: Counter example guided synthesis of monitors for realizability enforcement. In: Proc. of ATVA'12, pp. 238–253 (2012)
21. Hallé, S., Bultan, T.: Realizability analysis for message-based interactions using shared-state projections. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 27–36. ACM, New York (2010)
22. Hoare, C.A.R.: An axiomatic basis for computer programming. ACM **12**, 576–580 (1969)
23. Hopcroft, J.E.: Introduction to Automata Theory, Languages, and Computation. Pearson Education, India (1979)
24. Kazhamiakin, R., Pistore, M.: Analysis of realizability conditions for web service choreographies. In: Proceedings of the 26th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems, FORTE'06, pp. 61–76. Springer, Berlin (2006)
25. Lohmann, N., Wolf, K.: Realizability is controllability. In: Laneve, C., Su, J. (eds.) Proc. of WS-FM'09. LNCS, vol. 6194, pp. 110–127. Springer, Berlin (2009)
26. Mateescu, R., Garavel, H.: XTL: a meta-language and tool for temporal logic model-checking. STTT **98**, 33–42 (1998)
27. Autili, M., Ruscio, D.D., Salle, A.D., Inverardi, P., Tivoli, M.: A model-based synthesis process for choreography realizability enforcement. In: Proc. of FASE 2013, LNCS, vol. 7793, pp. 37–52. Springer, Berlin (2013)
28. Milner, R.: A Calculus of Communicating Systems. Springer, Berlin (1980)
29. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (1978)
30. OMG: Business Process Model and Notation (BPMN) Version 2.0. http://www.omg.org/spec/BPMN/2.0 (2010)
31. Poizat, P., Salaün, G.: Checking the realizability of BPMN 2.0 choreographies. In: Proc. of SAC'12. ACM Press, New York (2012)
32. Qiu, Z., Zhao, X., Cai, C., Yang, H.: Towards the theoretical foundation of choreography. In: Proc. of WWW'07. ACM Press, New York (2007)
33. Rodin: User Manual of the RODIN Platform (2007). http://deploy-eprints.ecs.soton.ac.uk/11/1/manual-2.3.pdf
34. Salaün, G., Bultan, T., Roohi, N.: Realizability of choreographies using process algebra encodings. IEEE T. Serv. Comput. **5**(3), 290–304 (2012)
35. Stengel, Z., Bultan, T.: Analyzing singularity channel contracts. In: Proc. of ISSTA'09. ACM, New York (2009)
36. Zoubeyr, F., Tari, A., Ouksel, A.M.: Backward validation of communicating complex state machines in web services environments. Distrib. Parallel Databases **27**(3), 255–270 (2010)