Correct-by-Construction Evolution of Realisable Conversation Protocols

Sarah Benyagoub, Meriem Ouederni, Neeraj Kumar Singh, Yamine Aït-Ameur

▶ **To cite this version:**

Sarah Benyagoub, Meriem Ouederni, Neeraj Kumar Singh, Yamine Aït-Ameur. Correct-by-Construction Evolution of Realisable Conversation Protocols. MEDI 2016 - Model and Data Engineering - 6th International Conference, Sep 2016, Almería, Spain. pp.260-273. hal-02353832

HAL Id: hal-02353832
https://hal.science/hal-02353832

Submitted on 7 Nov 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Open Archive Toulouse Archive Ouverte (OATAO)

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author-deposited version published in: http://oatao.univ-toulouse.fr/
Eprints ID: 23590

To link to this article : DOI:10.1007/978-3-319-45547-1_21

URL : https://doi.org/10.1007/978-3-319-45547-1_21

To cite this version: Benyagoub, Sarah and Ouederni, Meriem and Singh, Neeraj Kumar and Aït-Ameur, Yamine *Correct-by-Construction Evolution of Realisable Conversation Protocols*. (2016) In: MEDI 2016 - Model and Data Engineering - 6th International Conference, 21 September 2016 - 23 September 2016 (Almería, Spain)

Any correspondence concerning this service should be sent to the repository administrator:
staff-oatao@listes-diff.inp-toulouse.fr

Correct-by-Construction Evolution of Realisable Conversation Protocols

Sarah Benyagoub[1], Meriem Ouederni[2(✉)], Neeraj Kumar Singh[2], and Yamine Ait-Ameur[2]

[1] University of Mostaganem, Mostaganem, Algeria
benyagoub.sarah@univ-mosta.dz
[2] IRIT-INP of Toulouse, Toulouse, France
{meriem.ouederni,neeraje,yamine}@enseeiht.fr

Abstract. Distributed software systems are often built by composing independent and autonomous peers with cross-organisational interaction and no centralised control. These peers can be administrated and executed by geographically distributed and autonomous companies. In a top-down design of distributed software systems, the peers' interaction is often described by a global specification called Conversation Protocol (CP) and one have to check its realisability *i.e.*, whether there exists a set of peers implementing this CP. In dynamic environments, CP needs to be updated *wrt.* new environment changes and end-user interaction requirements. This paper tackles CP evolution such that its realisability must be preserved. We define some evolution patterns and prove that they ensure the realisability. We also show how our proposal can be supported by existing methods and tools based on refinement and theorem proving, using the event-B langage and RODIN development tools.

Keywords: System evolution · Realisability · Conversation protocols · Formal verification · Proof and refinement · Correct-by-construction method · Event-B

1 Introduction

In a top-down design of distributed software, the system interaction is usually modelled as a set of conversations, *i.e.*the allowed sequences of sent messages. Such a model is called conversation protocol (CP) [1] and describes the whole distributed system as a unique entity. Considering a CP, one must check whether there exists a set of peers where their composition generates the same sequences of send messages as specified by the CP. This issue characterises the realisability problem. Several recent work has tackled the CP realisability in order to avoid errors such as deadlocks or no-respect of messaging order specified in a CP.

In this paper, we take the realisability challenge one step forward: we study the correct evolution of realisable CPs. In fact, these specify cross-organisational interactions with no centralised control between peers which can be administrated and executed by geographically distributed and autonomous companies.

In this setting, system interaction and the corresponding CP need to be updated continuously over time in order to cope with new environment changes and end-user requirements. However, changing CP might result in knock-on effects on its realisability. Hence, verifying the correctness of CP evolution to ensure realisability preservation must also be run continuously.

In our work, we rely on the necessary and sufficient conditions defined in [2] for CP realisability, considering asynchronously communication throughout FIFO buffers with no restriction on their buffer sizes. This work solves the realisability issue for a subclass of asynchronously communicating peers, namely, the synchronisable systems, *i.e.*, systems composed of interacting peers behave equally by applying synchronous or asynchronous communication. A CP is realisable if there exists a set of peers implementing that CP, *i.e.*, they send messages to each other in the same order as in the CP, and such that their composition is synchronisable. In [2], the full checking of CP realisability applies the following steps: (i) peer projection from CP; (ii) checking synchronisability; and (iii) checking equivalence between CP and its distributed system.

Regarding the literature, existing work such as [3–5] give some solutions for system evolution. In [3,4], the authors propagate the choreography updates into communicating peers. Roohi et al. [5] focus on system reconfiguration meaning that a CP has been updated into CP' by checking whether a set of traces that has been executed in CP can be performed again in CP'. This reconfiguration can be better applied for run-time system to ensure execution consistency. All these approaches do not consider realisability preservation.

There exist other research approaches which can be applied as a posteriori evolution checking. The approaches suggest solutions every time the realisability check fails. For example, existing work on enforcing CP realisability, such as the one given in [6] and recently on CP repairability [7] can be used to ensure the realisability of an already updated CP.

Our statement is different than existing work and it is as follows: an evolution is allowed if it does not violate the CP realisability. By doing so, we suggest a priori verification approach of CP evolution. Instead of running the full realisability checking as described previously and detailed in Sect. 2, our proposal consists in performing partial verification uniquely at the CP level in order to answer the question if there *still* exists a set of peers implementing the updated CP. In this work, we consider the evolution at the CP level and we study its realisability effect on the distributed peers. The main issue is considering that system specifications may change over time (*e.g.*, service upgrade or degrade by adding and/or removing either exchanging messages or interacting peers). So, how can we ensure realisability preservation? To answer these questions, we formally describe the systems using Labelled Transition Systems (LTSs). We identify a set of behavioural properties sufficient to assert that the CP evolution due to the application of each evolution operator and their composition is correct. For this purpose, we define a set of patterns of correct evolution and we suggest a naive method to prove that these patterns preserve CP realisability.

The remainder of this paper is structured as follows: Sect. 2 introduces the background on which our proposal relies. Section 3 presents a *correct-by-construction* checking of CP realisability. In Sect. 4, we suggest some evolution operators and an algebra. Section 5 illustrates our contribution through an illustrative example. Section 6 presents our tool support. Section 7 overviews related work. Finally, Sect. 8 concludes our work with some future perspectives.

2 Background

This section presents the main definitions for CP realisability. We use Labeled Transition Systems (LTSs) for modelling CP and peers. This behavioural model defines the order of sent messages in a CP. At the peers level, the LTS can be computed by projection from CP and they describe the order in which those peers execute their send and receive actions. Lastly, we define synchronisable systems, and we present the realisability condition considering asynchronous communication.

Definition 1 (Peer). *A peer is an LTS* $\mathcal{P} = (S, s^0, \Sigma, T)$ *where* S *is a finite set of states,* $s^0 \in S$ *is the initial state,* $\Sigma = \Sigma^! \cup \Sigma^? \cup \{\tau\}$ *is a finite alphabet partitioned into a set of send messages, receive messages, and the internal action, and* $T \subseteq S \times \Sigma \times S$ *is a transition relation.*

We write $m!$ for a send message $m \in \Sigma^!$ and $m?$ for a receive message $m \in \Sigma^?$. We use the symbol τ (tau in figures) for representing internal activities. A transition is represented as $s \xrightarrow{l} s'$ where $l \in \Sigma$ and $\{s, s'\} \subseteq S$.

Example 1. The right side of Fig. 1 shows an example of three peers modelled as LTSs.

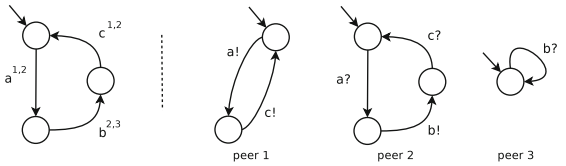


Fig. 1. CP (left side), Peers (right side)

Definition 2 (Conversation Protocol: CP). *A conversation protocol CP for a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *is an LTS* $CP =< S_{CP}, s^0_{CP}, L_{CP}, T_{CP} >$ *where* S_{CP} *is a finite set of states and* $s^0_{CP} \in S_{CP}$ *is the initial state;* L_{CP} *is a set of labels where a label* $l \in L_{CP}$ *is denoted* $m^{\mathcal{P}_i,\mathcal{P}_j}$ *such that* $\mathcal{P}_i$ *and* $\mathcal{P}_j$ *are the sending and receiving peers, respectively,* $\mathcal{P}_i \neq \mathcal{P}_j$*, and* m *is a message on which those peers interact; finally,* $T_{CP} \subseteq S_{CP} \times L_{CP} \times S_{CP}$ *is the transition relation. We require that each message has a unique sender and receiver:* $\forall(\mathcal{P}_i, m, \mathcal{P}_j), (\mathcal{P}'_i, m', \mathcal{P}'_j) \in L_{CP} : m = m' \implies \mathcal{P}_i = \mathcal{P}'_i \wedge \mathcal{P}_j = \mathcal{P}'_j$.

In the remainder of this paper, we denote a transition $t \in T_{CP}$ as $s \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_j}} s'$ where s and s' are source and target states and m^{P_i,P_j} is the transition label.

Example 2. The left side of Fig. 1 shows an example of CP modelled as LTS.

Definition 3 (Projection). *Peer LTSs* $\mathcal{P}_i =< S_i, s_i^0, \Sigma_i, T_i >$ *are obtained by replacing in* $CP =< S_{CP}, s_{CP}^0, L_{CP}, T_{CP} >$ *each label* $(\mathcal{P}_j, m, \mathcal{P}_k) \in L_{CP}$ *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* [8].

Example 3. Notice that the peers on Fig. 1 are obtained by projection from the CP shown on left side of the same Figure.

Synchronous System. Here, an interaction occurs between two peers if both agree on a synchronisation label, *i.e.*, if one peer is in a state in which a message can be sent, then another peer must be in a state where the same message can be received. The peers can however evolve independently from the others through internals actions.

Definition 4 (Synchronous System). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$*, the synchronous system* $(\mathcal{P}_1 \mid \ldots \mid \mathcal{P}_n)$ *is the LTS* (S, s^0, Σ, T) *where:*

- $S = S_1 \times \ldots \times S_n$
- $s^0 \in S$ *such that* $s^0 = (s_1^0, \ldots, s_n^0)$
- $\Sigma = \cup_i \Sigma_i$
- $T \subseteq S \times \Sigma \times S$*, and for* $s = (s_1, \ldots, s_n) \in S$ *and* $s' = (s'_1, \ldots, s'_n) \in S$

interact $s \xrightarrow{m} s' \in T$ *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* $\forall k \in \{1, \ldots, n\}, k \neq i \wedge k \neq j \Rightarrow s'_k = s_k$

Asynchronous System. Here, the peers communicate with each other through FIFO buffers. Each peer $\mathcal{P}_i$ is equipped with a (possibly) unbounded message buffer Q_i. A peer can either send a message $m \in \Sigma^!$ to the tail of the receiver buffer Q_j at any state where this send message is available, read a message $m \in \Sigma^?$ from its buffer Q_i if the message is available at the buffer head, or evolve independently through an internal action. Since reading from the buffer is not considered as an observable action, it is encoded as an internal action in the asynchronous system.

Definition 5 (Asynchronous Composition). *Given a set of peers* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *with* $\mathcal{P}_i = (S_i, s_i^0, \Sigma_i, T_i)$*, and* Q_i *being its associated buffer, the asynchronous composition* $((\mathcal{P}_1, Q_1) \mid\mid \ldots \mid\mid (\mathcal{P}_n, Q_n))$ *is the labeled transition system* $LTS_a = (S_a, s_a^0, \Sigma_a, T_a)$ *where:*

1. $S_a \subseteq S_1 \times Q_1 \times \ldots \times S_n \times Q_n$ *where* $\forall i \in \{1, \ldots, n\},\ Q_i \subseteq (\Sigma_i^?)*$
2. $s_a^0 \in S_a$ *such that* $s_a^0 = (s_1^0, \epsilon, \ldots, s_n^0, \epsilon)$ *(where* ϵ *denotes an empty buffer)*
3. $\Sigma_a = \cup_i \Sigma_i$

4. $T_a \subseteq S_a \times \Sigma_a \times S_a$, *and for* $s = (s_1, Q_1, \ldots, s_n, Q_n) \in S_a$ *and* $s' = (s'_1, Q'_1, \ldots s'_n, Q'_n) \in S_a$

send $s \xrightarrow{m!} s' \in T_a$ *if* $\exists i, j \in \{1, \ldots, n\}$ *where* $i \neq j : m \in \Sigma_i^! \cap \Sigma_j^?$, *(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)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

consume $s \xrightarrow{\tau} s' \in T_a$ *if* $\exists i \in \{1, \ldots, n\} : m \in \Sigma_i^?$, *(i)* $s_i \xrightarrow{m?} s'_i \in T_i$, *(ii)* $mQ'_i = Q_i$, *(iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow Q'_k = Q_k$, *and (iv)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

internal $s \xrightarrow{\tau} s' \in T_a$ *if* $\exists i \in \{1, \ldots, n\}$, *(i)* $s_i \xrightarrow{\tau} s'_i \in T_i$, *(ii)* $\forall k \in \{1, \ldots, n\} : Q'_k = Q_k$, *and (iii)* $\forall k \in \{1, \ldots, n\} : k \neq i \Rightarrow s'_k = s_k$

Synchronisability. A system built over a set of peers $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ is synchronisable [2], when both synchronous and asynchronous compositions obtained by application of Definitions 4 and 5, respectively, are equivalent. The equivalence holds if and only if the order of sending messages is the same. This relation is referred to as language equivalence and it is formalised in [2].

Well-formedness. The realisability condition on which we rely requires that the asynchronous system must be well-formed. It consists in checking whenever the i-th peer buffer Q_i is non-empty, the system can eventually move to a state where Q_i is empty. It has been shown in [2] that for every synchronisable set of peers, if the peers are deterministic, *i.e.*, for every state, the possible send messages are unique, well-formedness holds.

Definition 6 (Realisability). *A conversation protocol CP is realisable, denoted R(CP), if and only if (i) the peers computed by projection from this protocol are synchronisable, (ii) the asynchronous system resulting from the peer composition is well-formed, and (iii) the synchronous version of the distributed system* $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ *is equivalent to CP.*

3 *Correct-by-Construction* Realisability

In order to check the realisability condition given in Definition 6, we rely on a stepwise *correct-by-construction* approach to build asynchronous distributed systems [9]. We apply several refinement steps where the sufficient and necessary realisability conditions must be preserved in each step. Notice that these conditions are described using invariants. 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. By doing so, this method gives a formalised proof of an algorithm for a priori realisability checking and such a method scales up to any number of peers communicating with each other.

In this section, we define only the abstract model of CP to show feasibility and scalibilty of our proposed approach. To define the static properties of the system,

we declare three sets: *SET_LTS* - a set of Labeled Transition Systems, *STATES* - a set states and *MESSAGES* - a set of exchanged messages. A set of required additional properties related to these sets is given using axioms ($axm1$-$axm6$). An enumerated set is declared in $axm7$. A set of constants is declared in order to specify the required behaviour of the system. These constants are: *LTS_STATES* a relation between LTS and their states, *INITIAL_STATES* - a set of initial states, *FINAL_STATES* - a set of final states, *EXCHANGED_MESSAGES* - a set of exchanged messages, *ETIQ* - a set of edges relating two LTS states, *TRANSITIONS* - a set of transitions, *S_Next-_States* - a next synchronous state and *A_Next_States* - a next asynchronous state. Additional properties are defined using other axioms.

$$
\begin{array}{l}
axm1 : SET\_LTS \neq \varnothing \\
axm2 : finite(SET\_LTS) \land card(SET\_LTS) \geq 2 \\
axm3 : STATES \neq \varnothing \\
axm4 : finite(STATES) \land card(STATES) \geq 2 \\
axm5 : MESSAGES \neq \varnothing \\
axm6 : finite(MESSAGES) \land card(MESSAGES) \geq 1 \\
axm7 : partition(ACTIONS, \{Send\}, \{Receive\}, \{Internal\}) \\
axm8 : LTS\_STATES \in SET\_LTS \leftrightarrow STATES \\
\ldots \\
axm10 : INITIAL\_STATES \in SET\_LTS \leftrightarrow STATES \\
\ldots \\
axm15 : FINAL\_STATES \in SET\_LTS \leftrightarrow STATES \\
axm16 : EXCHANGED\_MESSAGES \in SET\_LTS \leftrightarrow MESSAGES \\
\ldots \\
axm21 : ETIQ \subseteq ACTIONS \times MESSAGES \times SET\_LTS \\
\ldots \\
axm23 : TRANSITIONS \in (STATES \times ETIQ) \nrightarrow STATES \\
\ldots \\
axm29 : S\_Next\_States \in \mathbb{P}(TRANSITIONS) \times \mathbb{P}(SET\_LTS \times STATES) \nrightarrow \mathbb{P}(SET\_LTS \times STATES) \\
axm30 : A\_Next\_States \in \mathbb{P}(TRANSITIONS) \times \mathbb{P}(SET\_LTS \times STATES) \times \\
\qquad \mathbb{P}(SET\_LTS \times MESSAGES \times \mathbb{N}) \nrightarrow \mathbb{P}(SET\_LTS \times STATES)
\end{array}
$$

Abstract model describes CP behaviour abstractly that contains only initialisation of communication, progress, internal actions and reset. To model the dynamic behaviour, we declare a list of variables using invariants ($inv1$-$inv4$). These variables are: *Conversation* - a sequence that records a set of exchanged messages in the conversation, *Index* - a message exchange order, *lts* - a subset of *LTS_SET*, and *transitions* - a subset of *TRANSITIONS*.

$$
\begin{array}{l}
inv1 : Conversation \in SET\_LTS \times MESSAGES \times SET\_LTS \leftrightarrow \mathbb{N} \\
inv2 : Index \in \mathbb{N} \\
inv3 : lts \subseteq SET\_LTS \\
inv4 : transitions \subseteq TRANSITIONS
\end{array}
$$

Initially, we define three events *Interact*, *Internal* and *Reset*. The *Initialisation* event is a default event that initialises initial state of the system, for example, this event sets the conversation to the empty set. The next *Interact* event shows the progress of CP by sending and receiving actions. In this event, the first guard shows the type of given parameters, and the next two guards states the required conditions according to Definition 2. The actions ($act1$ and $act2$) of the this event are used to update the message sequencer *Conversation* and the message index order *Index*. The next *Internal* event is used to model the internal actions (τ) and the last *Reset* event is used to reset the conversation.

EVENT Initialisation $\triangleq$
...
EVENT Interact $\triangleq$
ANY lts_source, lts_destination, message
WHERE
grd1 : $lts\_source \in lts \wedge lts\_destination \in lts \wedge message \in MESSAGES$
grd2 : $\exists send\_st\_src, send\_st\_dest \cdot send\_st\_src \in STATES \wedge send\_st\_dest \in STATES \wedge$
$(((send\_st\_src \mapsto (Send \mapsto message \mapsto lts\_destination)) \mapsto send\_st\_dest) \in transitions)$
grd3 : $\exists receive\_st\_src, receive\_st\_dest \cdot receive\_st\_src \in STATES \wedge receive\_st\_dest \in STATES \wedge$
$(((receive\_st\_src \mapsto (Receive \mapsto message \mapsto lts\_source)) \mapsto receive\_st\_dest) \in transitions)$
THEN
act1 : $Conversation := Conversation \cup \{(lts\_source \mapsto message \mapsto lts\_destination) \mapsto Index\}$
act2 : $Index := Index + 1$
END
EVENT Internal $\triangleq$
...
EVENT Reset $\triangleq$
...

The refinement models of the abstract events presents synchronous and asynchronous behaviour of CP, which are not presented here due to page limitations.

4 *Correct-by-Construction* Evolution

4.1 Behavioural Properties

In order to check the realisability of a CP that has been updated, we must ensure that the resulting LTS does not violate the realisability condition. We define in the following some properties which enable us to check the realisability preservation after evolution.

Branches Related Properties.

Property 1 (Non-Deterministic Choice). Given a $CP =< S_{CP}, s^0_{CP}, L_{CP}, T_{CP} >$, a state $s_{CP} \in S_{CP}$ is a non-deterministic branching state if: $\exists\{s_{CP} \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_j}} s'_{CP}, s_{CP} \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_j}} s''_{CP}\} \subseteq T_{CP}$ such that $s'_{CP} \neq s''_{CP}$.

We define in the following divergent choice (also called non-local branching choice in the literature). It differs from process divergence definition [10].

Property 2 (Divergent-Choice). Given a $CP =< S_{CP}, s^0_{CP}, L_{CP}, T_{CP} >$, a state $s_{CP} \in S_{CP}$ is a divergent branching state if: $\exists\{s_{CP} \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_j}} s'_{CP}, s_{CP} \xrightarrow{m'^{\mathcal{P}_j,\mathcal{P}_i}} s''_{CP}\} \subseteq T_{CP}$ such that $s'_{CP} \neq s''_{CP}$, and $m \neq m'$.

Sequences Related Properties. Given a CP, there is at least two partitions of peers where no interaction between both partitions exists.

Property 3 (Independent Sequences). Given a $CP =< S_{CP}, s^0_{CP}, L_{CP}, T_{CP} >$, a transition sequence $s_{CP} \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_j}} s'_{CP} \ldots s''_{CP} \xrightarrow{m'^{\mathcal{P}_k,\mathcal{P}_q}} s'''_{CP}$, where "..." denotes a trace of transitions leading from state s'_{CP} to state s''_{CP} and such that all transitions are in T_{CP}, is called independent sequence if $\{P_i, P_j\} \cap \{P_k, P_q\} = \varnothing$.

The following property enables us to detect traces in a CP which lead to non-local emission choices made by two different peers in the distributed system. To avoid this situation, every peer that joins the conversation at an intermediate state (*i.e.*, different than the initial state) must be receiver the first time it appears in a CP. Otherwise, if a peer is sending a message m at an intermediate state, then this peer must appear as receiver in its last interaction before sending m.

Property 4 (Divergent Sequences). Given a $CP =< S_{CP}, s^0_{CP}, L_{CP}, T_{CP} >$, there exists a transition sequence $s^0_{CP} \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_j}} \dots s'_{CP} \xrightarrow{m'^{\mathcal{P}_k,\mathcal{P}_q}} s''_{CP}$ where all transitions are in T_{CP} such that:

- for every sender peer P_t involved in a transition before state s'_{CP}, $t \neq k$, or
- there exists at least a transition $s_{CP} \xrightarrow{m^{\mathcal{P}_k,\mathcal{P}_t}} s'''_{CP} \in T_{CP}$ such that:
  - s'_{CP} is reachable from s_{CP}, and
  - there is no transition in $s_{CP} \xrightarrow{m^{\mathcal{P}_k,\mathcal{P}_t}} s'''_{CP} \dots s'_{CP} \xrightarrow{m'^{\mathcal{P}_k,\mathcal{P}_q}} s''_{CP}$ where P_k is a receiver peer.

4.2 Evolution Patterns

CP evolution stands for two possible tasks, namely, adding and/or removing either messages and/or interacting peers. We define here how CP realisability can be preserved by applying some evolution patterns presented in the following.

Operators. We introduce two operators denoted as $\otimes_{(+,s_{CP})}$ and $\otimes_{(\gg,s_{CP})}$ for branching and sequential composition, respectively, at a state s_{CP} in CP. We also assume other operators not presented here for lack of space, namely, $\otimes_{(\|,s_{CP})}$ for parallel composition, and $\otimes_{(\circlearrowleft,s_{CP})}$ for looping composition. The operator $\otimes_{(\|,s_{CP})}$ generates at a state s_{CP} all the interleaved behaviour of a set of transitions such that every generated branch must satisfy sequence related properties. The operator $\otimes_{(\circlearrowleft,s_{CP})}$ enables us to add self-loop of the form $s \xrightarrow{m^{\mathcal{P}_i,\mathcal{P}_j}} s$ where $i \neq j$ and such that sequence related properties must be preserved.

Remark 1. The application of an operator $\otimes_{(op,s_{CP})}(CP, CP')$ assumes that the initial state of CP' is fused with the state s_{CP}.

Definition 7. $\otimes_{(\gg,s_{CP})}$ *Given a* $CP =< S_{CP}, s^0_{CP}, L_{CP}, T_{CP} >$, *a* $CP' =< S_{CP'}, s^0_{CP'}, L_{CP'}, T_{CP'} >$ *and a state* $s_{CP} \in S_{CP}$, *the sequential composition* $\otimes_{(\gg,s_{CP})}(CP, CP')$ *means that* CP *must be executed before* CP' *such that Properties 3 and 4 do not hold.*

Definition 8. $\otimes_{(+,s_{CP})}$ *Given a* $CP =< S_{CP}, s^0_{CP}, L_{CP}, T_{CP} >$, *a set* $\{CP'_i\}$, $i = 1..n$ *such that* $CP'_i =< S_{CP'_i}, s^0_{CP'_i}, L_{CP'_i}, T_{CP'_i} >$ *and a state* $s_{CP} \in S_{CP}$, *the branching composition* $\otimes_{(+,s_{CP})}(CP, \{CP'_1, \dots, CP'_n\})$ *means that there is a choice at* s_{CP} *between the remaining behaviour of* CP *(i.e., starting from* s_{CP}*) and all* CP'_i *such that:*

– *Properties 1 and 2 do not hold at the state* s_{CP}*, and*
– $\forall CP'_i,\ \otimes_{(\gg, s_{CP})}(CP, CP'_i)$ *holds*

An Algebra of Operators. We introduce in Listing 1.1 a CP algebra the evolution such that realisability is preserved. We refer to a state s^f as final if there is no outgoing transition at that state. We denote *ECP* as a *CP* that evolutes over time while preserving realisability. The expression ECP^+ stands for one or more *ECP*. We refer to a basic *CP* as $ECP_b = s \xrightarrow{\mathcal{P}_i, m, \mathcal{P}_j} s'$.

$$ECP ::= ECP_b \mid ECP\ op\ {ECP_b}^+$$

$$ECP_b ::= s \xrightarrow{\mathcal{P}_i, m, \mathcal{P}_j} s' \mid \varnothing$$

$$op ::= \otimes_{(+, s^f)} \mid \otimes_{(\gg, s^f)} \mid \otimes_{(\|, s^f)} \mid \otimes_{(\circlearrowleft, s^f)}$$

Listing 1.1. CP Evolution Grammar

4.3 About Correctness

In this section, we discuss the method we have used to check the correctness of the operators definitions and their composition introduced in Sect. 4.2 i.e. check how these operators preserve the realisability condition while building a CP by composing these operators. To do so, we rely on the global approach developed in [9]. The approach of [9] uses the Event-B method and refinement to produce the asynchronous projection of a CP. An abstract model corresponding to the CP description, is first refined to obtain the synchronous projection, and a second refinement produces the asynchronous projection from the synchronous one. The developed approach is a *correct-by-construction* approach, it relies on the Event-B method. The given sufficient and necessary realisability conditions borrowed from [2] are used to prove the correctness of these refinements.

In the context of CP evolution, we proceed as follows to ensure realisability preservation. We prove that each ECP_{bi} is realisable using the approach of [9]. Then, after each application of the composition operator, we apply the approach of [9] until the whole ECP correctness is checked. Another approach consists in checking the correctness of the whole ECP once for all using the approach of [9].

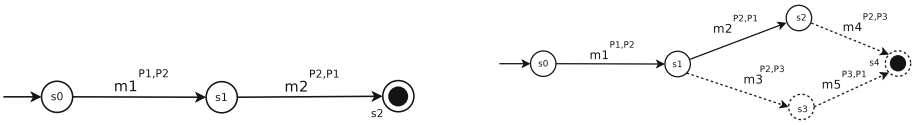
Note that the proposed verification procedure is a naive one. It requires to check the ECP each time an evolution operator is applied. Our intention in a future work, is to propose to check the sufficient conditions for realisability defined in Sects. 4.1 and 4.2. There is no need to re-check the whole ECP.

5 Two Illustrative Examples

This section, we first give an illustration of CP evolution using a simple example. Then, we give a more complex example to better illustrate the evolution operators introduced in this paper.

5.1 A First Example

The first example concerns the sequence and choice evolution operators. An initial CPs and a possible evolution of this CP are shown on Figs. 2a and b, respectively. To illustrate the evolution from one CP to the other one, the added behaviour is presented with dashed transitions on Fig. 2b.



(a) A simple CP (b) A possible ECP

Fig. 2. An evolution example of CP

Valid Evolution. In this example, the evolution preserves realisability. it produces an ECP by application of the evolution operators identified in (Listing 1.1) as follows. First, we identify below five basic CPs.

$$CP_{b0} = s0 \xrightarrow{m_1^{P1,P2}} s1$$

$$CP_{b1} = s1 \xrightarrow{m_2^{P2,P1}} s2$$

$$CP_{b2} = s1 \xrightarrow{m_3^{P2,P3}} s3$$

$$CP_{b3} = s2 \xrightarrow{m_4^{P2,P3}} s4$$

$$CP_{b4} = s3 \xrightarrow{m_5^{P3,P1}} s4$$

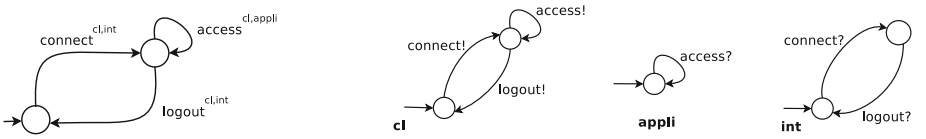
By applying the evolution operators on the basic CPs, we then obtain the following definition of the CP depicted on Fig. 2b:

$ECP = \otimes_{(\gg,s3)}(\otimes_{(\gg,s2)}(\otimes_{(+,s1)}(CP_{b0}, \{CP_{b1}, CP_{b2}\}), CP_{b3}), CP_{b4})$ Notice that the conditions defined in Sects. 4.1 and 4.2 each operator application hold for the ECP expression.

5.2 A More Complex Example

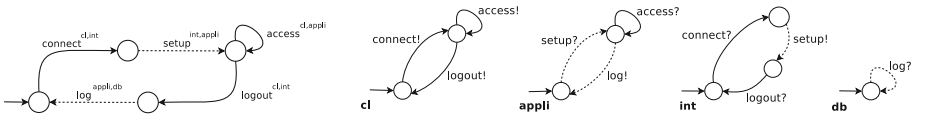
For illustration purposes we specify the use of an application in the cloud. This system involves four peers: a client (cl), a Web interface (int), a software application (appli), and a database (db). We show first a conversation protocol (Fig. 3a) describing the requirements that the designer expects from the composition-to-be. The conversation protocol starts with a login interaction (connect) between the client and the interface, followed by the access request (access) triggered by the client. This request can be repeated as far as necessary. Finally, the client decides to logout from the interface (logout)

Invalid Evolution. We show on Fig. 4a an updated version of the CP given on Fig. 3a describing the new requirements that the designer expects from the



(a) A realisable CP (b) Peers Projection

Fig. 3. CP and its projection



(a) A non correct CP Evolution (b) Peers Projection

Fig. 4. ECP and its projection

composition-to-be. The conversation protocol starts with a login interaction (connect) between the client and the interface, followed by the setup of the application triggered by the interface (setup). Then, the client can access and use the application as far as necessary (access). Finally, the client decides to logout from the interface (logout) and the application stores some information (start/end time, used resources, etc.) into a database (log).

Figure 4b shows the four peers obtained by projection. This set of peers seems to respect the behaviour specified in the conversation protocol, yet this is difficult to be sure using only visual analysis, even for such a simple example. In addition, as the *CP* involves looping behaviour, it is hard to know whether the resulting distributed system is bounded and finite, which would allow its formal analysis using existing verification techniques. Actually, this set of peers is not synchronisable (and therefore not realisable), because the trace of send messages "connect, access" is present in the 1-bounded asynchronous system, but is not present in the synchronous system. Synchronous communication enforces the sequence "connect, setup, access" as specified in the *CP*, whereas in the asynchronous system peer cl can send connect! and access! in sequence.

This kind of evolution resulting in non realisable CP can be avoided using our method with no need of CP projection as done in [6]. Starting from the initial state of CP that is shown on Fig. 3a and using the algebra given in Listing 1.1, there is no way to generate the interaction sequence "connect, setup, access" because adding "access" interaction violates Property 4.

6 Tool Support

Event-B [11,12] is a modelling method based on set-theory that enables to model a system by supporting a *correct by construction* approach, which allows to design a complex system using stepwise refinement by introducing the required system behaviour and desired functionalities in a new refinement step.

Each refinement step is verified by generated proof obligations corresponding to an abstract model and new refined behaviour. The stepwise modelling process finally leads to a concrete implementation of a system. In the Event-B language, *context* and *machine* are two important components, which describe static behaviour and dynamic behaviour, respectively. The static properties can be described using *carrier sets*, *enumerated sets*, *constants*, *theorems* and *axioms*, while a machine can be described using *variables*, *invariants*, *events* and *theorems*. To characterise the dynamic behaviour of a system, a list of events can be used to modify state variables by providing appropriate guards. In order to preserve the desired behaviour of a system, we defined a list of safety properties using invariants and theorems. Moreover, to introduce the convergence properties in the model, we can use *variant* clause in a machine. In a refinement step, an event can be refined by (1) keeping the events as it is; (2) splitting the event into several new events (3) strengthening the guards and actions (to make non-deterministic to deterministic). However, a new refinement level also allows to introduce a new event by modifying the new state variables.

The Rodin platform, java based integrated development environment (IDE) for Event-B, is set of tools to support model development, refinement, proof assistance, code generation and model animation. Due to page limitations, we have not presented a detailed introduction to Event-B. There are numerous publications and books available for an introduction to Event-B and related refinement strategies [11,12].

7 Related Work

Dynamic reconfiguration [13] is an interesting topic that play an important role for designing and developing a class of systems, such as distributed systems, graph transformation, software adaptation and meta modelling. Leite et al. [14] cover a survey on web service evolution, including various techniques and tools. In our work, we focus on the evolution of conversation protocols, and in this section, we describe existing work related to the evolution of CP. Roohi et al. [5] proposed a method to check CP reconfigurability by defining two different CPs, an initial CP and a new CP', and two sets of peers PS and PS'. These peers are obtained by projection of both CPs. A given trace t of CP consists in the history of the current execution. If the trace t can be executed in the reconfiguration peers (generated from CP'), the reconfiguration can take place.

Wombacher et al. [15] use the annotated Finite State Automata (aFSA) to describe the formal model of web service interaction using choreography. This approach preserves changes between updated choreography and the corresponding orchestration, in which the changes are made through adding and/or removing sequences of messages from the distributed peers. The proposed solution is implemented using DYCHOR framework, which requires human validation. A control evolution method, where propagating the changes into one peer requires to check its effect on other partner peers, is proposed in [3,16], which also use DYCHOR framework for implementation.

The evolution that might arise at the peers side is reported in [17,18], in which the authors propagate the change from one peer to other partners. Fdhila et al. [18] study the Business Process Management (BPM) and Service Oriented Architecture (SOA) to describe service choreographies using tree-based model considering some changes like delete, update, replace and insert for behavioural fragments.

A new programming language DIOC, free from deadlocks and races by construction, is defined for distributed applications [17]. The semantic of this language relies on labelled transition systems. The given approach enables to update the fragment of codes of distributed peers. In addition, it can be used at choreography level where code blocks can be updated dynamically and these code blocks must be tagged when describing the choreography. The run-time evolution and the required solution is discussed in [5,17,19].

Börger et al. [20] propose the semantics of concurrent Abstract State Machine (ASM), which overcome the problems of Gurevich's distributed ASM runs and generalise Lamport's sequentially consistent runs. The proposed semantics can also be used for designing the CPs in order to handle the concurrent scenarios.

To the best of our knowledge, this work is a pioneer in applying a *correct by construction* approach to model and verify the evolution of CP to guarantee that the realisability is preserved. Moreover, in our proposed solution, we have no restriction on the application domain, and we can use formal modelling techniques for designing, verification, and implementation of different distributed systems, *e.g.* web services, concurrent systems, Cyber Physical Systems, etc. Our result also applies for asynchronous systems as far as these systems are synchronisable without restricting the buffer size.

8 Conclusion and Perspectives

This paper presentes a preliminary solution for correct evolution of distributed systems for which their interaction is described with a conversation protocol. We proposed a language which enables one to incrementally design distributed systems that can be updated over time such that their realisability is preserved while applying evolution operators. Our naive method is used to prove that these operators preserve CP realisability. Finally, we illustrated our contribution using real-world example. As a main perspective of this work, we are extending the static model of [9] to support CP evolution. We aim at implementing the set of operators based on *correct-by-construction* method using Event-B.

References

1. Bultan, T.: Modeling interactions of web software. In: Proceedings of WWV 2006, pp. 45–52. IEEE(2006)
2. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: Proceedings of POPL 2012, pp. 191–202. ACM (2012)

3. Rinderle, S., Wombacher, A., Reichert, M.: On the controlled evolution of process choreographies. In: Proceedings of ICDE 2006, pp. 124–124. IEEE (2006)
4. Ryu, S.H., Casati, F., Skogsrud, H., Benatallah, B., Saint-Paul, R.: Supporting the dynamic evolution of web service protocols in service-oriented architectures. ACM Trans. Web (TWEB) **2**(2), 13 (2008)
5. Roohi, N., Salaün, G.: Realizability and dynamic reconfiguration of chor specifications. Informatica (Slovenia) **35**(1), 39–49 (2011)
6. Güdemann, M., Salaün, G., Ouederni, M.: Counterexample guided synthesis of monitors for realizability enforcement. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 238–253. Springer, Heidelberg (2012)
7. Basu, S., Bultan, T.: Automated choreography repair. In: Stevens, P., et al. (eds.) FASE 2016. LNCS, vol. 9633, pp. 13–30. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49665-7_2
8. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory. Languages and Computation. Addison Wesley, Reading (1979)
9. Farah, Z., Ait-Ameur, Y., Ouederni, M., Tari, K.: A correct-by-construction model for asynchronously communicating systems. Int. J. Softw. Tools Technol. Transfer 1–21 (2016)
10. Ben-Abdallah, H., Leue, S.: Syntactic detection of process divergence and non-local choice in message sequence charts. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 259–274. Springer, Heidelberg (1997)
11. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New York (2010)
12. Project RODIN: Rigorous Open Development Environment for Complex Systems (2004). http://rodin-b-sharp.sourceforge.net/
13. Medvidovic, N.: ADLs and dynamic Architecture Changes. In: Proceedings of SIGSOFT 1996 Workshops, pp. 24–27. ACM (1996)
14. Leite, L.A., Oliva, G.A., Nogueira, G.M., Gerosa, M.A., Kon, F., Milojicic, D.S.: A Systematic Literature Review of Service Choreography Adaptation. SOCA **7**(3), 199–216 (2013)
15. Wombacher, A.: Alignment of choreography changes in BPEL processes. In: Proceedings of SCC 2009. pp. 1–8. IEEE (2009)
16. Rinderle, S., Wombacher, A., Reichert, M.: Evolution of process choreographies in DYCHOR. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 273–290. Springer, Heidelberg (2006)
17. Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies. In: Holvoet, T., Viroli, M. (eds.) Coordination Models and Languages. LNCS, vol. 9037, pp. 67–82. Springer, Heidelberg (2015)
18. Fdhila, W., Indiono, C., Rinderle-Ma, S., Reichert, M.: Dealing with change in process choreographies: design and implementation of propagation algorithms. Inf. Syst. **49**, 1–24 (2015)
19. Jureta, I.J., Faulkner, S., Thiran, P.: Dynamic requirements specification for adaptable and open service-oriented systems. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 270–282. Springer, Heidelberg (2007)
20. Börger, E., Schewe, K.D.: Concurrent abstract state machines. Acta Informatica 1–24 (2015)