Specifying stateful asynchronous properties for distributed programs

Tzu-Chun Chen and Kohei Honda

Queen Mary College, University of London

Abstract. Having *stateful* specifications to track the states of processes, such as the balance of a customer for online shopping or the booking number of a transaction, is needed to verify real-life interacting systems. For safety assurance of distributed IT infrastructures, specifications need to capture states in the presence of asynchronous interactions. We demonstrate that not all specifications are suitable for asynchronous observations because they implicitly rely on an order-preservation assumption. To establish a theory of asynchronous specifications, we use the interplay between synchronous and asynchronous semantics, through which we characterise the class of specifications suitable for verifications through asynchronous interactions. The resulting theory offers a general semantic setting as well as concrete methods to analyse and determine semantic well-formedness (healthiness) of specifications with respect to asynchronous observations, for both static and dynamic verifications. In particular, our theory offers a key criterion for suitability of specifications for distributed dynamic verifications.

1 Introduction: the challenges while states are considered

This technical report is a full version of the published paper "Specifying Stateful Asynchronous Properties for Distributed Programs" [5].

In this paper, the theories and properties of *stateful* specifications for *dynamic observing* among communicating processes are introduced. They are based on the condition that observations are done asynchronously, where a semantic problem in specifications for distributed systems arises.

The semantic problem arises in a concrete engineering setting, through the collaboration with the design and development of a large IT infrastructure for ocean sciences [16]. In that infrastructure, applications are predominantly built as asynchronous interactions among distributed components. Some of these components may be contributed by the third party so that they may be buggy or untrusted. The desire is to filter undesirable messages by having system-level observers (e.g. runtime monitors) observe their behaviours, and then *dynamically* enforce safe behaviour. However, putting system-level observers at every endpoint is very expensive and they might be polluted by malicious endpoints.

During runtime, to *judge* the behaviours of distributed endpoint components, an observer needs a judgement basis, *a specification*, so that she can understand if those behaviours are well-behaved or not *against to the specification* through observing the asynchronous messages transmitted among the components. It is thus needed to formulate an expressive *specification language* usable for asynchronously observing compo-

nents during runtime. Moreover, besides the nature of asynchrony, in large-scale distributed systems the use of *state* is omnipresent in specifications to capture real-life scenarios, where e.g. the (expected) states of participants in the applications, such as the credit of a client for on-line shopping, or the booking number for a transaction, play a critical role in specifications.

Then a specification language for dynamic (run-time) observing becomes not trivial when *states* and asynchrony are both considered. A basic issue in the *semantics* of a specification language in the presence of asynchronous communication arises. The issue makes naturally written specifications *semantically nonsensical*, thus posing a fundamental challenge to our endeavour to provide a consistent specification-verification framework.

Lets consider a general situation. When an observer (e.g. a trusted monitor) is located at an observee, the order of the observee's actions the observer sees is exactly the same as the one happening at the observee. However, when she sits outside the observee, e.g. remotely from the observee, the order of actions that she observes may not necessarily be the same as the one happening at the observee. I call the former kind of observation *synchronous*, and the latter *asynchronous*. Although the synchronous observation can capture more precisely the "actual" behaviour of the observee, in distributed systems, asynchronous observations are the norm and often a necessity. Can such a remote observer still check behaviours of processes against a non-trivial stateful specification? What are the classes of specifications that can be used for such purposes? Is there an algorithmic method to check if a specification is usable in the asynchronous environment?

Contributions. In the remainder, §2 illustrates the semantic issue through concrete examples. Starting from these motivating examples, this paper presents the following contributions:

1. Introduction of an intuitive, semantically well-founded protocol-centred specification method suitable for asynchronous stateful behaviour (called SP for stateful protocols), enriching [4] with set-based stateful operations (§2, §3).
2. Identification (first to our knowledge) of a semantic issue when specifying asynchronous interaction behaviour combined with updatable states (§2).
3. Formal analysis of the issue through asynchronous trace semantics, reaching several criteria for asynchronous verifiability of specifications (healthiness conditions [10]) including a decidable one admitting a rich set of specifications (§4).

Finally in §5, the practical implications of the theory are examined, and related work and conclude with further topics are discussed.

2 Motivating Examples

Before formally introducing the syntax and semantics of specifications, this section motivates key ideas through simple examples. Our specification language is based on multiparty session types [3, 12] with annotations by logical formulae, extending [4]

with local state(s). Our focus is on visible asynchronous interactions rather than internal actions. We treat three examples. The first motivates the use of state in protocol specifications through a typical business scenario. The second introduces the semantic issue in specifications in the presence of asynchrony through a simplest possible example. The third example illustrates the use of sets in specifications for asynchrony, linking the specification to the corresponding traces, adumbrating our semantic analysis later.

2.1 Using state(s) in protocol specifications

The first example motivates the use of state(s) in specifications. Consider the following purchase-invoice scenario:

(step 1) Buyer sends a *product name* (denoted by *PName*) to Seller, then Seller replies with its *price*, and Buyer decides to purchase (then go to step 2) or not (then terminate). We assume shipping is done independently.
(step 2) Seller sends the Buyer an *invoice* for the purchased product.

This scenario can be realised as a single protocol [4, 8, 12] between Buyer and Seller, consisting of a series of a few message exchanges. The scenario can also be realised using *two* protocols, one for each step. This form has a merit in flexibility: for example, when Buyer and Seller finish step 1, both can terminate that transaction. Then an invoice may be issued any time later. Example 1 presents protocols with logical annotations following this second framework.

Example 1 (SP for a cross-session Purchase-and-Invoice scenario).

$$
\begin{aligned}
G_{\mathsf{pcs}} = {} & B \to S : \mathit{Request}(\mathit{PName} : \mathsf{string}). \\
& S \to B : \mathit{Confirm}(\mathit{PNameConf} : \mathsf{string}, \mathit{Price} : \mathsf{int}) \\
& \qquad \langle \mathit{PNameConf} = \mathit{PName} \ \wedge \ \mathit{Price} \geq 0; \varepsilon \rangle \\
& \qquad \langle \mathsf{true}; \varepsilon \rangle. \\
& B \to S : \{ \mathit{OK}(\mathit{UserID} : \mathsf{int}) \langle \mathit{UserID} \neq 0; \varepsilon \rangle \langle \mathsf{true}; \varepsilon \rangle. \\
& \qquad S \to B : (\mathit{PNo} : \mathsf{int}) \\
& \qquad\quad \langle \mathit{PNo} \notin \mathsf{dom}(\mathbf{PLog}); \\
& \qquad\qquad \mathbf{PLog} := \mathbf{PLog} \cup \{ \mathit{PNo} \mapsto (\mathit{UserID}, \mathit{PName}, \mathit{Price}) \} \rangle \\
& \qquad\quad \langle \mathsf{true}; \varepsilon \rangle. \\
& \qquad \mathsf{end} \\
& \quad \mathit{KO}().\mathsf{end} \} \\
G_{\mathsf{ivc}} = {} & S \to B : (\mathit{PNo} : \mathsf{string}, \mathit{Invoice} : \mathsf{int}) \\
& \qquad \langle \mathit{PNo} \in \mathsf{dom}(\mathbf{PLog}) \ \wedge \mathit{Invoice} = \mathbf{PLog}(\mathit{PNo}) \rangle \langle \mathsf{true}; \varepsilon \rangle. \mathsf{end}
\end{aligned}
$$

Above G_{pcs} and G_{ivc} denote the global *stateful protocols*, or SPs from now on for short, corresponding to Steps 1 and 2. Each specifies the behaviour which the participants, S (denoting seller) and B (denoting buyer), should realise at each session. $\langle ...; ... \rangle \langle ...; ... \rangle$ are the obligations for sender (the former) and receiver (the latter), respectively. Note that $\langle \mathsf{true}; \varepsilon \rangle$ means no obligation and no state change. We will formally introduce their syntax in Section 3.3. In this example, the state of S, represented by the field **PLog** (the Purchase Log, which we consider to be a key-value store, mapping distinct keys to values), links the two protocols. Both specifications can be read intuitively. First, in G_{pcs},

1. B first sends a request (*Request* is an operator name), with the message value *PName* of type string, which is a product name.
2. S confirms by sending the same product name and its price, where the latter should be a non-negative integer as annotated.
3. If B says *OK* by sending its identity, then (in practice, after authenticating the identity) S sends back a *fresh* purchase number *PNo* which should be fresh, i.e. it should not be in the domain of **PLog**. As a result, this new key and the corresponding information is added to **PLog**. On the other hand, if B says *KO* (not *OK*), then the conversation terminates.

As may be seen from this example, our protocol specifications use a local state to record an abstraction of preceding interactions across sessions, and to constrain future behaviours using this record. Its ultimate aim is to specify visible behaviours of a process: thus the stipulated state does not (have to) come from an actual state of a process, i.e. we may call it a "ghost state" following JML [1].

2.2 Synchrony v.s. asynchrony for stateul specification

The next example illustrates the central topic of this paper, asynchrony in specifications, showing how a specification can be "too synchronous" for asynchronous observations. We focus on a part of the previous example. The purchase number allocator S will, upon a request from a buyer B at each session, issue a booking number incrementing the previously issued one: so S issues e.g. 1, 2, 3, ... in a sequence of sessions. Figure 2 (a) shows a protocol G_{sync} for such interactions, specifying a simple behaviour that the participants, S and B, should realise at each session. $\mathbf{c}$ is a local state of S, denoting the next booking number.

Example 2 (SPs for purchase number allocator: synchronous v.s. asynchronous).

(a) synchronous spec

$$\begin{array}{l} G_{\mathsf{sync}} = B \to S : \mathrm{req}(\varepsilon). \\ \qquad S \to B : \mathrm{ans}(x : \mathsf{int}) \\ \qquad\quad \langle x = \mathbf{c};\ \mathbf{c} := \mathbf{c}+1 \rangle . \langle \mathsf{true}; \varepsilon \rangle . \\ \qquad \mathsf{end} \end{array}$$

(b) asynchronous spec

$$\begin{array}{l} G_{\mathsf{async}} = B \to S : \mathrm{req}(\varepsilon). \\ \qquad S \to B : \mathrm{ans}(x : \mathsf{int}) \\ \qquad\quad \langle x \notin \mathbf{c};\ \mathbf{c} := \mathbf{c} \cup x \rangle . \langle \mathsf{true}; \varepsilon \rangle . \\ \qquad \mathsf{end} \end{array}$$

In the first line of G_{sync}, B (for buyer) requests S (for seller) a purchase number by sending $\mathrm{req}(\varepsilon)$, where ε means there is no message value in this request. In the second line, an integer x is sent from S to B, for which $\langle x = \mathbf{c}; \mathbf{c} := \mathbf{c}+1 \rangle$ specifies the *obligation* for S, while no obligation i.e. $\langle \mathsf{true}; \varepsilon \rangle$ for B. The first part "$x = \mathbf{c}$" says that x should be equal to $\mathbf{c}$. The second part "$\mathbf{c} := \mathbf{c}+1$" says that, after sending, S will increase $\mathbf{c}$ by 1, which constrains further behaviours of S in later sessions.

G_{sync} is an example of a SP which makes sense synchronously but *not* asynchronously. It seems an intuitively sensible specification: however, if a remote observer is far away, even if S *actually* sends the series of booking numbers $1,2,3,4,\ldots$ in this order, they may arrive at the observer as e.g. $2,4,1,3,\ldots$, under the assumption that the order of messages belonging to *distinct* sessions may not be preserved, which is a practical assumption. In particular, note this remote observer will consider S as being *ill-behaved*

with respect to G_{sync}: the correctness for S (which is synchronous) and the correctness for its observer (which is asynchronous) are not congruent.

As a remedy, we present G_{async} in Example 2 (b), which is intended for asynchronous observation. We now use the *set* of booking numbers: $\mathbf{c}$, whose type is a set of integers, corresponds to **PLog** in Example 1. The new specification just says, in brief, that "S always sends a fresh number". If the behaviour of S satisfies this condition at S, then even though messages from S may arrive out-of-order, the remote observer can verify that they are correct w.r.t. G_{async}, so that the actions of S and their asynchronous observation by a remote observer coincide. We shall later verify this statement formally.

2.3 Capturing causality using sets in stateful specification

While G_{async} gives a reasonable specification, it is not a strongest possible specification if our target is a server that issues booking numbers incrementally based on the previous number. For example, if the same buyer sequentially repeats a series of request-reply sessions, that buyer (and an observer sitting in-between) will observe $1,2,3,4$, but this point is not captured by G_{async}.

Example 3 (A refinement of G_{async}).

$$\begin{array}{l} G_{\mathsf{assign}} = B \to S : \mathrm{req}(\varepsilon)\langle \mathsf{true}; \varepsilon\rangle\langle \mathsf{true}; \mathbf{t} := \mathbf{t}+1, \mathbf{c} := \mathbf{c} \uplus \mathbf{t}\rangle. \\ \qquad\quad S \to B : \mathrm{ans}(x : \mathsf{int})\langle x \in \mathbf{c};\ \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle\langle \mathsf{true}; \varepsilon\rangle. \\ \qquad\quad \mathsf{end} \end{array}$$

G_{assign} in Example 3 is a refinement of G_{async} in §2.2 so that, while still being suitable for asynchronous observations, can capture a stronger causal constraint. It uses two states: $\mathbf{t}$, a counter, and $\mathbf{c}$, a collection of valid numbers to be issued. $\mathbf{t}$ and $\mathbf{c}$ are incremented when receiving a request, while the sent value is taken off from $\mathbf{c}$.

The intuition behind the construction is as follows:

1. If S receives n requests, then (assuming the server issues the booking numbers starting from 1) as a whole the numbers which can be issued are among $\{1,2,..,n\}$.
2. If S issues a number from this set, the remaining numbers are what it can issue.

As we shall see later, the specification has the property that, if S behaves well w.r.t. G_{assign}, then a remote observer also finds it well-behaved.

To understand G_{assign} as a specification, consider two sessions following the protocol, s_1 and s_2. Assume the initial states are $\mathbf{t} \mapsto 0$ and $\mathbf{c} \mapsto \{\}$. Then G_{assign} says the traces in Figure 1 are valid ones (we list the traces together with step-by-step state change: (I,II,III) are categories each stipulating how states will change).
Above, $s_1[B,S]?\mathrm{req}(\varepsilon)$ denotes an input ? from B to S at session s_1 carrying a req-message without value; $s_1[S,B]!\mathrm{ans}(1)$ is an output ! from S to B at s_1 carrying a ans-message with value 1. (I) and (II) are the traces where a remote observer observes that two consecutive inputs have arrived first. Note that, even if S may have indeed outputted immediately after the first input, we can have these traces due to asynchrony. Even then, unlike G_{async}, the observer is always sure, by G_{assign}, that the returned values should be no more than 2, i.e. it is either 1 or 2. In (III), the observer observes the second request only after the answer to the first request. Note the request-answer order in each session

Traces of permitted actions with the corresponding state change

cases	1st	2nd	3rd	4th
(I) actions	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_1[S,B]!\mathrm{ans}(1)$	$s_2[S,B]!\mathrm{ans}(2)$
	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_1[S,B]!\mathrm{ans}(1)$	$s_2[S,B]!\mathrm{ans}(2)$
	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_2[S,B]!\mathrm{ans}(1)$	$s_1[S,B]!\mathrm{ans}(2)$
	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_2[S,B]!\mathrm{ans}(1)$	$s_1[S,B]!\mathrm{ans}(2)$
(I) states	$\mathbf{t}\mapsto 1,\mathbf{c}\mapsto\{1\}$	$\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{1,2\}$	$\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{2\}$	$\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{\}$
(II) actions	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_1[S,B]!\mathrm{ans}(2)$	$s_2[S,B]!\mathrm{ans}(1)$
	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_1[S,B]!\mathrm{ans}(2)$	$s_2[S,B]!\mathrm{ans}(1)$
	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_2[S,B]!\mathrm{ans}(2)$	$s_1[S,B]!\mathrm{ans}(1)$
	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_2[S,B]!\mathrm{ans}(2)$	$s_1[S,B]!\mathrm{ans}(1)$
(II) states	$\mathbf{t}\mapsto 1,\mathbf{c}\mapsto\{1\}$	$\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{1,2\}$	$\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{1\}$	$\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{\}$
(III) actions	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_1[S,B]!\mathrm{ans}(1)$	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_2[S,B]!\mathrm{ans}(2)$
	$s_2[B,S]?\mathrm{req}(\varepsilon)$	$s_2[S,B]!\mathrm{ans}(1)$	$s_1[B,S]?\mathrm{req}(\varepsilon)$	$s_1[S,B]!\mathrm{ans}(2)$
(III) states	$\mathbf{t}\mapsto 1,\mathbf{c}\mapsto\{1\}$	$\mathbf{t}\mapsto 1,\mathbf{c}\mapsto\{\}$	$\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{2\}$	$\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{\}$

Fig. 1. The valid traces of synchronous and asynchronous interactions w.r.t. G_{assign}

is preserved because without the request, its answer cannot occur (which is Lamport's ordering [14]: more generally, we assume the order of two messages is preserved inside each session, as we shall formalise later). Unlike G_{async}, the observer can expect, based on G_{assign}, that the first answer is surely 1; and the second is surely 2.

The above example shows how we can represent causality while still (intuitively) keeping the asynchronous nature of specifications. It also shows how traces of actions can be used to give extensional meaning of specifications. This observation is given a formal account in §3.3 later.

2.4 Analyses of capturing causality: G_{assign}

Continue with section 2.3, from G_{assign} we have learnt that an observer can be more insightful if she has more information (which are initially established by input actions):

Remark 4. An observer (remote/close) can expect and reasonably guess the behaviours of her observee based on the messages she has observed. Note the messages delivered by observees can represent the behaviours of observees.

Based on G_{assign}, in which the assigned number to each session (established at each request) is asked to be increased with the coming requests one by one, the following properties say that a remote observer can expect the next outputted value based on what she observes. Let a sequence of actions, called *trace*, denoted by $\mathtt{s}$. Let $\mathtt{I}(\mathtt{s})$ be the

sequence of input actions of $\mathtt{s}$, $\mathtt{O}(\mathtt{s})$ be the sequence of output actions of $\mathtt{s}$, and $\mathtt{V}(\mathtt{s})$ be the sequence of values carried in $\mathtt{s}$. Let $\mathsf{num}(\mathtt{s})$ be the length of $\mathtt{s}$, and $\mathtt{prefix}(\mathtt{s})$ be the set of prefixes of $\mathtt{s}$. Moreover, let $\mathrm{v}(\ell)$ be the value carried by action ℓ, and w_1 be the initial value of state $\mathbf{f} \in G_{\mathsf{assign}}$, while w_2 be the initial value of state $\mathbf{c} \in G_{\mathsf{assign}}$.

For every trace w.r.t role S (representing server) satisfying $\upharpoonright G_{\mathsf{assign}}S$, it should have the following properties:

$\mathfrak{tq}0$. binding conventions (defined in Definition 14).
$\mathfrak{tq}1$. $\mathsf{num}(\mathtt{I}(\mathtt{s})) \geq \mathsf{num}(\mathtt{O}(\mathtt{s}))$;
$\mathfrak{tq}2$. $\forall \ell \in \mathtt{s} \wedge \ell = s[S,B]!l(v) \subset \exists \ell' \in \mathtt{s}, \ell' = s[B,S]?l(v)$.
$\mathfrak{tq}3$. $\forall \ell, \ell' \in \mathtt{O}(\mathtt{s}), \ell \neq \ell'$, then $\mathrm{v}(\ell) \neq \mathrm{v}(\ell')$;
$\mathfrak{tq}4$. $\forall \ell \in \mathtt{O}(\mathtt{s}'), \mathtt{s}' \in \mathtt{prefix}(\mathtt{s}), \mathsf{w} \leq \mathrm{v}(\ell) \leq \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{w}$.

The first 3 properties ($\mathfrak{tq}0$, $\mathfrak{tq}1$ and $\mathfrak{tq}2$) ensures the trace satisfies the interaction structure defined in G_{assign}. $\mathfrak{tq}$ 3 ensures that every assigned (outputted) value (i.e. $\ell \in \mathtt{O}, (\mathrm{v}(\ell))$ is distinct. And $\mathfrak{tq}4$ says that, an outputted value carried by an output action should be smaller than or equal to the number of inputs which positioning ahead of it. $\mathfrak{tq}3$ and $\mathfrak{tq}4$ together imply that, for a given trace $\ell_1 ... \ell_n$, the next outputted value has a fixed range: this range can be measured by knowing how many requests from buyers have happened and how many assignments to buyers have done.

For example, when an observer observes a trace $s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[B,S]?\mathrm{req}(\varepsilon) \cdot s_3[B,S]?\mathrm{req}(\varepsilon)$, if the next action is $s_2[S,B]!\mathrm{ans}(v)$, then $v \in \{1,2,3\}$; assume $v = 3$. After action $s_2[S,B]!\mathrm{ans}(3)$, if the upcoming one is $s_1[S,B]!\mathrm{ans}(v')$, then $v' \in \{1,2\}$ because 3 has been assigned; assume $v' = 1$. When a new request comes, says $s_4[B,S]?\mathrm{req}(\varepsilon)$, currently the overall actions starting from the initial is

$$s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[B,S]?\mathrm{req}(\varepsilon) \cdot s_3[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[S,B]!\mathrm{ans}(3) \cdot s_1[S,B]!\mathrm{ans}(1) \cdot s_4[B,S]?\mathrm{req}(\varepsilon)$$

the next output action of it, say $s_n[S,B]!\mathrm{ans}(v'')$, $n \in \{3,4\}$ and $v'' \in \{2,4\}$.

Moreover, the properties are useful for verifying a trace step by step. For example, assume the following trace is an initial trace: $s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,B]!\mathrm{asn}(2) \cdot s_2[B,S]?\mathrm{req}(\varepsilon)$ is not right immediately at $s_1[S,B]!\mathrm{asn}(2)$ because the only possible value is 1.

3 The language of stateful specifications

3.1 Syntax of protocols and specifications.

Figure 2 summarises the grammar of global stateful protocols (SPs) $(G, \ldots)$, which specify the interaction structure of a session from a global viewpoint; and local SPs $(T, \ldots)$ which specify protocols for endpoints, to be projected from G. Their syntax enriches [4] with local states update and rich operations on them: by adding simple state update, we obtain a rich class of stateful specifications.

A SP uses a state consisting of zero or more *field*s. A field gets read in a *predicate* A and gets read and written in an *update* E. Note that, one important assumption here is: for a session whose scope is G, except the update of state(s) defined in E at each interaction, the state(s) will not be changed by any other update rules outside of G. To allow state(s) being updated by other update rules outside the scope of a session

$$
\begin{array}{ll}
S ::= \mathsf{nat} \mid \mathsf{bool} \mid \mathsf{string} & A ::= \mathsf{true} \mid \mathsf{false} \mid e_1 = e_2 \mid e_1 > e_2 \\
\quad \mid S_1 \times S_2 \mid \mathsf{set}(S) \mid \mathsf{map}(S_1, S_2) & \quad \mid e_1 \in e_2 \mid A_1 \wedge A_2 \mid \neg A \\
e ::= x \mid v \mid \mathbf{f} \mid op(e_1, ..., e_n) & E ::= \varepsilon \mid E, \mathbf{f} := e
\end{array}
$$

$$
\begin{array}{lll}
G ::= \mathtt{p} \to \mathtt{q} : \{l_i(x_i : S_i)\langle A_i; E_i\rangle\langle A_i'; E_i'\rangle.G_i\}_{i \in I} & & \text{G-cm} \\
\quad \mid \; G_1 \mid G_2, \; role(G_1) \cap role(G_2) = \emptyset & & \text{G-par} \\
\quad \mid \; \mathsf{end} & & \text{G-end}
\end{array}
$$

$$
\begin{array}{lll}
T ::= \mathtt{p}!\{l_i(x_i : S_i)\langle A_i; E_i\rangle.T_i\}_{i \in I} & & \text{L-sel} \\
\quad \mid \; \mathtt{p}?\{l_i(x_i : S_i)\langle A_i; E_i\rangle.T_i\}_{i \in I} & & \text{L-bra} \\
\quad \mid \; \mathsf{end} & & \text{L-end}
\end{array}
$$

Fig. 2. The grammar of stateful protocols

is not discussed in this thesis. We call $\langle A; E\rangle$ *obligation*. We use updates instead of post-conditions for usability in runtime verification. $(S, \ldots)$ are sorts (data types), and $(e, \ldots)$ are expressions, where $op(e_1, ..., e_n)$ is the operation op on parameters $e_1, ..., e_n$. We use data types such as product $S_1 \times S_2$, set $\mathsf{set}(S)$ and (finite) function $\mathsf{map}(S_1, S_2)$. Sets and functions play important roles in asynchronous specifications. In expressions, x is a variable, v is a value, $\mathbf{f}$ is a (mutable) field. In E, $\mathbf{f} := v$ updates $\mathbf{f}$ with the value v. The grammar of G and T is simplified for distilled presentation. G and T are the bases for the endpoint specifications. G specifies, globally, the capability of an endpoint (represented by a shared channel)l while T specifies the *obligation* for every action coming from endpoints. The recursion of G and T can be omitted because, while a recursion is finite, we can unfold the finitely-recursive *obligations* of the body to several continuous obligations, each of which is defined by T. All results are preserved.

In G, $\mathtt{p} \to \mathtt{q}$ describes the communication (interaction) from sender $\mathtt{p}$ to receiver $\mathtt{q}$, while $\mathtt{p}!$ and $\mathtt{p}?$ are endpoint actions for output (to $\mathtt{p}$) and input (from $\mathtt{p}$). In $l_i(x_i : S_i)$, l_i is the label for a branch, when l_j is chosen, the interaction variable is x_j, and S_j is its type. In G-cm, the first obligation $\langle A; E\rangle$ is for the sender, indicating a sender should guarantee that its message x satisfies A and as a result E is done; the second obligation $\langle A'; E'\rangle$ is for the receiver, indicating it can expect a message x to satisfy A' and as a result E' is done. Rule G-par particularly asks that $role(G_1) \cap role(G_2) = \emptyset$, which means that no role is shared by G_1 and G_2, where $role(G)$ denotes the set of roles in G.

G-par provides SP language flexibility, and benefits programmers to write tasks in a more readable structure. However, with the permutation rules for stateful specifications defined in 3.5 it shows that generally, in *asynchronous environment*, $G_1 \mid ... \mid G_n$ in parallel (G-par) can be represented by one wrapped G (G-cm), which will be illustrated in Section 3.5. There I will have a detailed discussion for the reason why $role(G_1) \cap role(G_2) = \emptyset$ is asked in rule G-par. This restriction not only prevents deadlock, but also simplifies the grammar of local protocols: there is no need to have a structure $T_1 \mid T_2$ for local SP.

Rule L-sel is for sender's behaviours, while rule L-bra is for receiver's behaviours. Parallel composition specifies two interactions in parallel, while end denotes the end of interactions.

As a notational convention, if an obligation is trivial (i.e. the predicate is true and the update is ε) then it is omitted. Further, if either the predicate or the update is trivial in an obligation, then it is omitted.

3.2 Projection from global SP to local SP.

Global protocols are useful to capture overall interaction scenarios, while a local protocol specifies exactly what the endpoint should do. They are linked by *endpoint projection*.

Assume $\mathtt{p},\mathtt{q},\mathtt{r} \in G$ and $\mathtt{p} \neq \mathtt{q}$. $\mathtt{var}(G) \upharpoonright \mathtt{p}$ is the set of variables that $\mathtt{p}$ knows in G. Based on the projection defined in [4], the projection $\mathrm{Proj}(G,\ A,\ \mathtt{r})$ is inductively defined as:

$$\mathrm{Proj}(\mathtt{p} \to \mathtt{q} : \{l_i(x_i : S_i)\langle A_i\ ;\ E_i\rangle\langle A_i'\ ;\ E_i'\rangle.G_i\}_{i\in I},\ A^{pj},\ \mathtt{r}) =$$
$$\begin{cases} \mathtt{q}!\{l_i(x_i : S_i)\langle A_i\ ;\ E_i\ \rangle.G_i^{pj}\}_{i\in I} & \text{if } \mathtt{r} = \mathtt{p} \neq \mathtt{q}, \\ \mathtt{p}?\{l_i(x_i : S_i)\langle\ A_i'\ ;\ E_i'\rangle.G_i^{pj}\}_{i\in I} & \text{if } \mathtt{r} = \mathtt{q} \neq \mathtt{p}, \\ G_1^{pj} & \text{if } \mathtt{r} \neq \mathtt{q} \neq \mathtt{p} \text{ and } \forall\, i,j, \mathrm{Proj}(G_i^{pj},A^{pj},\mathtt{r}) = \mathrm{Proj}(G_j^{pj},A^{pj},\mathtt{r}) \end{cases}$$
$$\text{with } G_i^{pj} = \mathrm{Proj}(G_i,A^{pj},\mathtt{r})$$

$$\mathrm{Proj}(G_1 \mid G_2,\ A^{pj},\ \mathtt{r}) = \begin{cases} \mathrm{Proj}(G_i,\ A^{pj},\mathtt{r}) & \text{if } \mathtt{r} \in G_i \text{ and } \mathtt{r} \notin G_j\ i,j \in \{1,2\}, \\ \mathsf{end} & \text{else} \end{cases}$$
$$\mathsf{end} \upharpoonright \mathtt{r} = \mathsf{end}$$

When a side condition does not hold the projection $\mathrm{Proj}(G,A,\mathtt{r})$ is undefined. The projection of G onto $\mathtt{r}$, written in $G \upharpoonright \mathtt{r}$, is defined by $\mathrm{Proj}(G,\mathsf{true},\mathtt{r})$.

We illustrate the idea by the following example.

Example 5 (endpoint projection). The local SPs projected from G_{assign} are:

$$G_{\mathsf{assign}} \upharpoonright S = T_S = B?\mathrm{req}(\varepsilon)\langle \mathsf{true}; \mathbf{t} := \mathbf{t}+\mathbf{1}\ \ \mathbf{c} := \mathbf{c} \cup \mathbf{t}\rangle.B!\mathrm{ans}(x : \mathsf{int})\langle x \in \mathbf{c}\ ;\ \mathbf{c} := \mathbf{c} \setminus x\rangle.\mathsf{end}$$
$$G_{\mathsf{assign}} \upharpoonright B = T_B = S!\mathrm{req}(\varepsilon).S?\mathrm{ans}(x : \mathsf{int}).\mathsf{end}$$

3.3 Specifications.

A *specification* is a triple $\Theta ::= \langle \Gamma;\Delta;D\rangle$ which gives a behavioural specification of a local process (endpoint) as its interface. Γ, Δ and D, separated by ";" in Θ, are given by:

$$\Gamma ::= \emptyset \mid \Gamma, a : \mathtt{I}(G[\mathtt{p}]) \mid \Gamma, a : \mathtt{O}(G[\mathtt{p}]) \mid \Gamma, \mathbf{f} : S \qquad \Delta ::= \emptyset \mid \Delta, s[\mathtt{p}] : T \qquad D ::= \emptyset \mid D, \mathbf{f} \mapsto v$$

Above, $\mathtt{I}$ (resp. $\mathtt{O}$) is a mode denoting input (resp. output) capability. Γ, *shared environment*, describes several information: the permitted behaviour at each shared channel; and the type of each field. When a process has $a : \mathtt{I}(G[\mathtt{p}])$, it can *accept* invitations via a shared channel a to play the role $\mathtt{p}$ following what (the $\mathtt{p}$-projection of) G specifies;

while $a:\mathtt{0}(G[\mathtt{p}])$ is its dual. In Δ, *session environment*, $s[\mathtt{p}]:T$ describes the session behaviour (T) in a session s as $\mathtt{p}$. D is a set of (ghost) states of a local process (endpoint): the states in $D \in \Theta$ belong to an endpoint participant in a session. Each D is a map from fields to values, storing a range of data structure. In formulae, a field $\mathbf{f}$ itself represents its current value.

Example 6. Based on G_{assign} in Example 3 and its local SPs in Example 5, we give a local specification Θ_{ass} for server, playing role S, and Θ_{B_1} and Θ_{B_2} for two buyers B_1 and B_2, each playing role B in G_{assign}, assuming there are two ongoing sessions s_1 and s_2.

$T_{\mathsf{ass}} = B?\mathrm{req}(\varepsilon)\langle \mathsf{true}; \mathbf{t} := \mathbf{t}+\mathbf{1}\ \ \mathbf{c} := \mathbf{c} \cup \mathbf{t}\rangle.T'_{\mathsf{ass}},\ T'_{\mathsf{ass}} = B!\mathrm{ans}(x:\mathsf{int})\langle x \in \mathbf{c}\ ;\ \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle.\mathsf{end}$
$\Theta_{\mathsf{ass}} = \langle \Gamma'_{\mathsf{ass}}, \mathtt{server} : \mathtt{I}(G_{\mathsf{ass}}[S]);\ \Delta'_{\mathsf{ass}}, s_1[S] : T_{\mathsf{ass}}, s_2[S] : T_{\mathsf{ass}};\ D'_{\mathsf{ass}}, \mathbf{t} \mapsto 0, \mathbf{c} \mapsto \{\}\rangle$

$T_B = S!\mathrm{req}(\varepsilon).T'_B,\ \ T'_B = S?\mathrm{ans}(x:\mathsf{int}).\mathsf{end},\ \ \Theta_{B_1} = \langle \Gamma'_{B_1}, \mathtt{b}_1 : \mathtt{0}(G_B[B]);\ \Delta'_{B_1},\ s_1[B] : T_B;\ D_{B_1}\rangle$
$\Theta_{B_2} = \langle \Gamma'_{B_2}, \mathtt{b}_2 : \mathtt{0}(G_B[B]);\ \Delta'_{B_2}, s_2[B] : T_B;\ D_{B_2}\rangle$

The data storage in Θ_{ass} is $D'_{\mathsf{ass}}, \mathbf{t}, \mathbf{c}$. In this protocol, no state in D'_{ass} is used. Similarly, no state in D_{B_1} or D_{B_2} is used. Although we do not illustrate the whole procedures of session establishment (by using rules [ACC] and [REQ] defined in Figure 3), it shows that buyers B_1 and B_2 are the session inviters requesting S to join session s_1 and s_2.

3.4 Semantics of specifications.

In Figure 3, we present the semantics of specifications as a (deterministic [1]) labelled transition system (LTS), of the form $\Theta \xrightarrow{\ell} \Theta'$, which intuitively means Θ as a specification *allows* a process to do an action ℓ, and demands the resulting process to conform to Θ'.

Definition 7.

$$\Theta \xrightarrow{\ell_1} \Theta_1 \xrightarrow{\ell_2} \Theta_2 .... \xrightarrow{\ell_k} \Theta_k ... \stackrel{\text{def}}{=} \Theta \xrightarrow{\ell_1 \cdot \ell_2 ... \ell_k} \Theta_k ...$$

Definition 8. We say $\Theta' \in \mathtt{trans}(\Theta)$ if $\exists \mathtt{s} = \ell_1 \cdot \ell_2 ... \ell_k$ such that $\Theta \xrightarrow{\mathtt{s}} \Theta'$.

For actions labels, we use $\overline{a}(s[\mathtt{p}] : G)$ for sending an invitation when s is fresh to the sender, and use $\overline{a}\langle s[\mathtt{p}] : G\rangle$ for sending an invitation when s is not new to the sender, $a(s[\mathtt{p}] : G)$ for accepting an invitation when s is fresh to the receiver (which is the only case we consider), and $s[\mathtt{p},\mathtt{q}]!l(v)$ and $s[\mathtt{p},\mathtt{q}]?l(v)$ for sending and receiving in a session. We do not use τ since it is irrelevant in the present work (because, in brief, τ is always possible and has no effects on specifications).

In Figure 3, the only difference between rule [REQ-INI] and [REQ] is the former is used when s is fresh, which means that, when the first request happens to the sender to ask someone for playing role $\mathtt{p}_j$ in a fresh s, we use [REQ-INI], where the round parenthesis indicates it is a binding occurrence and we record all capabilities except the passed one in the linear typing environment; otherwise we use [REQ]. [REQ] says that,

[1] If $\Theta \xrightarrow{\ell} \Theta'$ and $\Theta \xrightarrow{\ell} \Theta''$, then $\Theta' = \Theta''$.

$$[\text{REQ-INI}]\quad \frac{s \notin \mathsf{dom}(\Delta),\ role(G) = \{\mathtt{p}_i\}_{i\in I}}{\langle \Gamma, a : \mathtt{0}(G[\mathtt{p}_j]); \Delta, \{s[\mathtt{p}_i] : G \upharpoonright \mathtt{p}_i\}_{i\in I}; D\rangle \xrightarrow{\overline{a}\langle s[\mathtt{p}_j]:G\rangle} \langle \Gamma, a : \mathtt{0}(G[\mathtt{p}_j]); \Delta, \{s[\mathtt{p}_i] : G \upharpoonright \mathtt{p}_i\}_{i\in I\setminus\{j\}}; D\rangle}$$

$$[\text{REQ}]\quad \frac{s \in \mathsf{dom}(\Delta),\ role(G) = \{\mathtt{p}_i\}_{i\in I}}{\langle \Gamma, a : \mathtt{0}(G[\mathtt{p}_j]); \Delta, s[\mathtt{p}_j] : G \upharpoonright \mathtt{p}_j; D\rangle \xrightarrow{\overline{a}\langle s[\mathtt{p}_j]:G\rangle} \langle \Gamma, a : \mathtt{0}(G[\mathtt{p}_j]); \Delta; D\rangle}$$

$$[\text{ACC}]\quad \frac{s \notin \mathsf{dom}(\Delta),\ T = G \upharpoonright \mathtt{q},\ field(T) \in D}{\langle \Gamma, a : \mathtt{I}(G[\mathtt{q}]); \Delta; D\rangle \xrightarrow{a(s[\mathtt{q}]:G)} \langle \Gamma, a : \mathtt{I}(G[\mathtt{q}]); \Delta, s[\mathtt{q}] : T; D\rangle}$$

$$[\text{SEL}]\quad \frac{T = \mathtt{q}!\{l_i(x_i : S_i)\langle A_i; E_i\rangle.T_i'\}_{i\in I},\ \Gamma \vdash v : S_j,\ \Gamma \models A_j\{v/x_j\},\ s \notin \mathsf{dom}(\Delta)}{\langle \Gamma; \Delta, s[\mathtt{p}] : T; D\rangle \xrightarrow{s[\mathtt{p},\mathtt{q}]!l_j(v)} \langle \Gamma; \Delta, s[\mathtt{p}] : T_j'\{v/x_j\}; D\,\mathtt{after}\,E_j\{v/x_j\}\rangle}$$

$$[\text{BRA}]\quad \frac{T = \mathtt{p}?\{l_i(x_i : S_i)\langle A_i; E_i\rangle.T_i'\}_{i\in I},\ \Gamma \vdash v : S_j,\ \Gamma \models A_j\{v/x_j\},\ s \notin \mathsf{dom}(\Delta))}{\langle \Gamma; \Delta, s[\mathtt{q}] : T; D\rangle \xrightarrow{s[\mathtt{p},\mathtt{q}]?l_j(v)} \langle \Gamma; \Delta, s[\mathtt{q}] : T_j'\{v/x_j\}; D\,\mathtt{after}\,E_j\{v/x_j\}\rangle}$$

$$[\text{PAR}]\quad \frac{\Theta_1 \xrightarrow{\ell} \Theta_2,\quad \mathsf{bn}(\ell) \cap \mathsf{n}(\Theta_3) = \emptyset}{\Theta_1, \Theta_3 \xrightarrow{\ell} \Theta_2, \Theta_3}$$

Fig. 3. Labelled Transition System for Specifications

when s is not new to the session environment, as Θ has an output channel a with G, (1) the target behaviour is permitted to send a request to ask someone to play role $\mathtt{p}_j$ in session s; and (2) after requesting, the capability at p_j is taken off. Rule [ACC] says that, if s is a new session to Θ, and all states declared in $G \upharpoonright \mathtt{q}$, $field(G \upharpoonright \mathtt{q})$, are in D, as Θ has an input channel a with G for accepting to play role $\mathtt{q}$, it accepts this request and plays session role $s[\mathtt{q}]$ specified by $G \upharpoonright \mathtt{q}$. Rule [SEL] is for sending a message inside a session. The premise says that, first, the type T should be a selection type; the passed value v has type S_j from the jth branch of T under Γ (note that, when v is a name, Γ is necessary to have the knowledge of its type, but it is not needed if v is a non-name value, like 3 or "*hello*" because its type is automatically known without Γ); and A_j after substitution is well-typed under Γ. In the conclusion, T_j' substitutes v for x_j and prepares for the next (incoming or outgoing) message, and the state is updated by $D\,\mathtt{after}\,E_j\{v/x_j\}$. To illustrate the updating of D by E_j, assume E_j is defined as $\mathbf{f} := \mathbf{f} \cup \{x_j\}$, and currently $\mathbf{f} \mapsto \{10\}$. After substituting 5 for x_j, D is updated to $\mathbf{f} \mapsto \{10, 5\}$. Rule [BRA] is a symmetric rule of [SEL]. Finally [PAR], where $\mathsf{bn}(\cdot)$ is the set of bound names and $\mathsf{n}(\cdot)$ is the set of names, says if Θ_1 and Θ_3 are composable, after action happens and Θ_1 becomes as Θ_2, they are still composable.

In this section, the trace of a local process is defined to describe a local process's run-time behaviour, and define the satisfaction criterion for formally describing a process satisfies a given specification synchronously/asynchronously. Note that, hereafter, when we say a trace, it always means a trace consisting of local actions inputting or outputting from a local process.

Definition 9 (trace). A *trace* $(\mathtt{s}, \mathtt{s}', \ldots)$ is a sequence of actions represented by

$$\mathtt{s} = \ell_1 \cdot \ell_2 \ldots \cdot \ell_n \ldots$$

in which each action is connected by " $\cdot$ ".

Note that, for every $\mathtt{s}$, we assume an accept/request action introducing a session, say s, binds the later occurrences of s.

Definition 10 (valid trace according to Θ). A trace $\mathtt{s}$ is valid according to Θ if there always exists Θ' such that $\Theta \xrightarrow{\mathtt{s}} \Theta'$.

3.5 Permutation of Stateful Protocols

We formalise the notion of minimum permutations for local Θ. Having permutation rules for Θ (see Definition 12) makes an observer (i.e. system monitor), who observes actions based on Θ, be able to capture the causality relation of actions when the order of actions is not preserved due to asynchrony. For example, in one session, when an endpoint, say $s[\mathtt{p}]$, waits for two input actions from different senders, say $s[\mathtt{q}_1]$ and $s[\mathtt{q}_2]$, the local Θ at role $s[\mathtt{p}]$ is

$$\mathtt{q}_1?l_1(x_1)\langle A_1;E_1\rangle\cdot\mathtt{q}_2?l_2(x_2)\langle A_2;E_2\rangle$$

then the order of actions may be $s[\mathtt{q}_1,\mathtt{p}]?l_1(v_1)\cdot s[\mathtt{q}_2,\mathtt{p}]?l_2(v_2)$ or $s[\mathtt{q}_2,\mathtt{p}]?l_1(v_2)\cdot s[\mathtt{q}_1,\mathtt{p}]?l_1(v_1)$ because there is no way to restrict that the first action should firstly happen and arrive earlier at $s[\mathtt{p}]$ then the second one.

The permutation rules for Θ has different purpose from the definition of commutativity (see Definition 58). The permutation rules for Θ are the *mechanisms* (or tools) that let an observer use them to properly observe actions, whose order is not preserved; while the attribute of commutativity of a Θ is used to validate whether the Θ is asynchronous verifiable or not.

Here we define stateful protocol context and local unit permutation. Assume that, whenever we write $\mathscr{T}[T]$, it always comes from a well-formed global stateful protocol (i.e. G). Recall that the well-formedness principles of global stateful protocol is introduced in section 3.8. If it is a local Θ containing the local SP projected from a non-well-formed global SP, it is not meaningful to explore the permutation mechanism when the original global stateful protocol is in a non-reasonable shape. For convenience, let an obligation $\langle A;E\rangle$ for sender be $\eta_!$, and an obligation $\langle A';E'\rangle$ for receiver be $\eta_?$.

Definition 11 (Local SP context). Let $\dagger ::= \{?,!\}$. A *local stateful-session type context* $\mathscr{T}[\,]$ is a local stateful protocol with a hole, whose grammar is defined as:

$$\mathscr{T}[\,] ::= [\,] \mid \mathtt{p}\dagger : \{l_1(x_1:S_1)\eta_{\dagger 1}.T_1,..,l_k(x_k:S_k)\eta_{\dagger k}.\mathscr{T}[\,],..,l_n(x_n:S_n)\eta_{\dagger n}.T_n\}$$

$\mathscr{T}[T]$ denotes the result of filling the hole in the local SP context $\mathscr{T}$ with local SP T. Note that $[\,]$ defines the identity function, i.e. $[T]=T$.

The causal order occurs when there are two consecutive sending (or receiving) actions with the same target (e.g. $\mathtt{q}!l(v).\mathtt{q}!l'(v')$ or $\mathtt{p}?l(x).\mathtt{p}?l'(x')$), or there is an input action followed by an output action.

Definition 12 (local unit permutation). The *unit permutation* of local SP is defined by the axioms below, assuming $p_1 \neq p_2$:

$$\begin{aligned}
&\mathscr{T}[\mathtt{p}_1!\{l_i(x_i:S_i)\eta_{!i}.\underline{\mathtt{p}_2}!\{l'_j(x'_j:S'_j)\eta'_{!j}.T_{ij}\}_{j\in I}\}_{i\in I}] \\
&\qquad\curvearrowright^1 \quad \mathscr{T}[\underline{\mathtt{p}_2}!\{l'_j(x'_j:S'_j)\eta'_{!j}.\mathtt{p}_1!\{l_i(x_i:S_i)\eta_{!i}.T_{ij}\}_{i\in I}\}_{j\in I}] \\
&\mathscr{T}[\mathtt{p}_1?\{l_i(x_i:S_i)\eta_{?i}.\underline{\mathtt{p}_2}?\{l'_j(x'_j:S'_j)\eta'_{?j}.T_{ij}\}_{j\in I}\}_{i\in I}] \\
&\qquad\curvearrowright^1 \quad \mathscr{T}[\underline{\mathtt{p}_2}?\{l'_j(x'_j:S'_j)\eta'_{?j}.\mathtt{p}_1?\{l_i(x_i:S_i)\eta_{?i}.T_{ij}\}_{i\in I}\}_{j\in I}] \\
&\mathscr{T}[\mathtt{p}_1!\{l_i(x_i:S_i)\eta_{!i}.\underline{\mathtt{p}_2}?\{l'_j(x'_j:S'_j)\eta'_{?j}.T_{ij}\}_{j\in I}\}_{i\in I}] \\
&\qquad\curvearrowright^1 \quad \mathscr{T}[\underline{\mathtt{p}_2}?\{l'_j(x'_j:S'_j)\eta'_{?j}.\mathtt{p}_1!\{l_i(x_i:S_i)\eta_{!i}.T_ij\}_{i\in I}\}_{j\in I}]
\end{aligned}$$

3.6 Permutation of Local Actions

Since Θ is a specification for a local endpoint, the actions concerned here are all *local* actions, which means that they are the actions happening at endpoints. Similarly, all permutations of actions are the permutation of local actions. Example below shows the relations between global actions and local actions.

Example 13 (global actions v.s. local actions). Assume a session s is given, in which the roles involving in it are $\mathtt{p}_1$ and $\mathtt{p}_2$. Assume globally, there are actions among session-roles $s[\mathtt{p}_1]$ and $s[\mathtt{p}_2]$:

$$\begin{aligned}
&s[\mathtt{p}_1,\mathtt{p}_2]!\langle\text{"How are you?"}\rangle \cdot s[\mathtt{p}_1,\mathtt{p}_2]?(\text{"How are you?"}) \cdot \\
&s[\mathtt{p}_2,\mathtt{p}_1]!\langle\text{"Fine. Thanks!"}\rangle \cdot s[\mathtt{p}_2,\mathtt{p}_1]?(\text{"Fine. Thanks!"})
\end{aligned}$$

with the scenario: An endpoint playing role $\mathtt{p}_1$ in session s, denoted by $s[\mathtt{p}_1]$, firstly sends a regard "How are you" to an endpoint playing role $\mathtt{p}_2$ in session s, denoted by $s[\mathtt{p}_2]$. After $s[\mathtt{p}_2]$ receives this regards, she replies $s[\mathtt{p}_1]$ with "Fine. Thanks!"; then $s[\mathtt{p}_1]$ receives this response. The sequence of actions can be observed by the following messages

$$s\langle\mathtt{p}_1,\mathtt{p}_2,\langle\text{"How are you?"}\rangle\rangle \cdot s\langle\mathtt{p}_2,\mathtt{p}_1,\langle\text{"Fine. Thanks!"}\rangle\rangle$$

This sequence of messages reflect the meaning of the sequence of actions.

Locally, the sequence of actions for session-role $s[\mathtt{p}_1]$ is

$$s[\mathtt{p}_1,\mathtt{p}_2]!\langle\text{"How are you?"}\rangle \cdot s[\mathtt{p}_2,\mathtt{p}_1]?(\text{"Fine. Thanks!"})$$

while the sequence of actions for session-role $s[\mathtt{p}_2]$ is

$$s[\mathtt{p}_1,\mathtt{p}_2]?(\text{"How are you?"}) \cdot s[\mathtt{p}_2,\mathtt{p}_1]!\langle\text{"Fine. Thanks!"}\rangle$$

A remote observer generally has more than one specifications (for different session-roles). For each session-role, the specification only specifies the local behaviours of that session-role.

To capture the causality among actions, below we define the relation of *dependency*:

Definition 14 (dependency). We define $\mathfrak{D}$, the dependency relation, such that $\ell_1 \mathfrak{D} \ell_2$ whenever

(a) ℓ_1 is either $a(k[\mathtt{p}] : G)$ or $a\langle k[\mathtt{p}] : G\rangle$, and $k[\mathtt{p}] \in \ell_2$ or $\mathsf{sbj}(\ell_2) = k[\mathtt{p}]$.
(b) ℓ_1 is $s[\mathtt{p}_1, \mathtt{p}_2]?l(x)$ and $x \in \ell_2$.

We say that ℓ_2 depends on ℓ_1 if $\ell_1 \mathfrak{D} \ell_2$.

The following examples illustrate the cases of dependency relation over action labels.

Example 15. Assume a sequence of actions $\ell_1 \cdot \ell_2$, where

$$\ell_1 = a\langle k[\mathtt{p}] : G\rangle,\ \ell_2 = \overline{b}\langle k[\mathtt{p}] : G\rangle.$$

Based on Definition 14.(a), ℓ_2 depends on ℓ_1.

Example 16. Assume a sequence of actions $\mathtt{s} = \mathtt{s}_0 \cdot \ell_1 \cdot \ell_2$, where

$$\ell_1 = a(k[\mathtt{p}] : G),\ \ell_2 = k[\mathtt{p}', \mathtt{p}]!l\langle v\rangle$$

ℓ_2 does not depend on ℓ_1 because, based on Definition 14.(a), $\mathsf{sbj}(\ell_2) = k[\mathtt{p}'] \neq k[\mathtt{p}]$.

Example 17. Assume a sequence of actions $\mathtt{s} = \mathtt{s}_0 \cdot \ell_1 \cdot \ell_2 \cdot \ell_3 \cdot \ell_4$, where

$$\ell_1 = a(k[\mathtt{p}] : G),\ \ell_2 = k[\mathtt{p}, \mathtt{q}]!l\langle v\rangle,\ \ell_3 = b\langle k'[\mathtt{q}] : G'\rangle,\ \ell_4 = k'[\mathtt{p}, \mathtt{q}]?l'(v')$$

ℓ_2 depends on ℓ_1 because, based on Definition 14.(a), $\mathsf{sbj}(\ell_2) = k[\mathtt{p}]$. Similarly, ℓ_4 depends on ℓ_3.

Definition 18 (suppress). For a sequence of actions $\ell_1 \cdot \ell_2$, we say ℓ_2 is suppressed by ℓ_1 if

1. $\ell_1 = s[\mathtt{p}_1, \mathtt{q}]?l(v)$ and $\ell_2 = s[\mathtt{q}, \mathtt{p}_2]!l'\langle v'\rangle$, or
2. $\ell_1 = s[\mathtt{p}, \mathtt{q}]!l\langle v\rangle$ and $\ell_2 = s[\mathtt{p}, \mathtt{q}]!l'\langle v'\rangle$, or
3. $\ell_1 = s[\mathtt{p}, \mathtt{q}]?l(v)$ and $\ell_1 = s[\mathtt{p}, \mathtt{q}]?l'(v')$.

Definition 19 (causality). We say ℓ_1 and ℓ_2 have causality relation if one of them depends or suppresses the other.

Definition 20 (legal unit permutation of local actions). Let $\ell_1 \cdot \ell_2$ be a trace. A permutation from $\ell_1 \cdot \ell_2$ to $\ell_2 \cdot \ell_1$ is *legal* if ℓ_1 and ℓ_2 have no causality relation.

We write $\mathtt{s} \curvearrowright \mathtt{s}'$ when $\mathtt{s}'$ is the result of applying zero or *more* legal unit permutations. In this case $\mathtt{s}'$ is *a permutation variant of* $\mathtt{s}$ and this permutation is called a *legal permutation*.

Definition 21 (valid unit permutation of actions according to Θ). We say $\mathtt{s}_1$ *is a permutation variant of* $\mathtt{s}_2$ *under* Θ, written $\Theta \vdash \mathtt{s}_1 \curvearrowright \mathtt{s}_2$, when $\mathtt{s}_1 \curvearrowright \mathtt{s}_2$, $\Theta \vdash \mathtt{s}_1$ and $\Theta \vdash \mathtt{s}_2$ hold.

Example 22 (legal permutations of actions). In Figure 1, all traces in (I) and (II) are permutation variants to each other. The traces in (III) can legally permute to any trace in (I) and (II), but not the converse.

Note that, it does not mean that a sequence of legally-permuted actions is always valid to a Θ. The following example explains this point.

Example 23. Let $\ell_1 = s[\mathtt{p},\mathtt{q}_1]!\mathrm{ans}\langle\varepsilon\rangle$ and $\ell_2 = s[\mathtt{p},\mathtt{q}_2]!\mathrm{ans}\langle\varepsilon\rangle$. Then $\ell_1\cdot\ell_2 \curvearrowright \ell_2\cdot\ell_1$. Assume that initial value of state $\mathbf{c}$ is 1, and that local SP for role $\mathtt{p}$ in session s is:

$$s[\mathtt{p}] : \mathtt{q}_1!\mathrm{ans}(\varepsilon)\langle \mathbf{c} = 1; \varepsilon\rangle \ .\ \mathtt{q}_2!\mathrm{ans}(\varepsilon)\langle \mathbf{c} = 1; \mathbf{c} := 0\rangle$$

Then $\Theta \vdash \ell_1\cdot\ell_2$, but $\Theta \nvdash \ell_2\cdot\ell_1$.

3.7 Satisfaction

The following simple definition of processes is enough for our purpose: we can readily use the π-calculus with session primitives and its weak (τ-abstracted) LTS to induce this abstract notion of processes.

Definition 24 (process). A *process* $(P, Q, ..)$ is a prefix-closed set of traces.

The following defines the notion of synchronous and asynchronous observables as the sets of traces observed by a synchronous observer (i.e. as it is) and by an asynchronous observer (i.e. up to legal permutations).

Definition 25 (synchronous and asynchronous observable).

1. $\mathsf{Obs}_s(P) \stackrel{\text{def}}{=} P$.
2. $\mathsf{Obs}_a(P)$ is the set of all legal permutation variants of the traces in P.

Firstly, recall the LTS rule of SP defined in Figure 3: a valid ℓ against a Θ is an action which can be verified by a LTS rule such that $\Theta \xrightarrow{\ell} \Theta'$.

Definition 26 ($\mathtt{trace}(\Theta)$: the set of valid traces of Θ). We define $\mathtt{trace}(\Theta)$ as the set of *valid traces* of Θ:

$$\mathtt{trace}(\Theta) = \{\mathtt{s} \mid \Theta \xrightarrow{\mathtt{s}} \Theta'\}$$

i.e. $\mathtt{trace}(\Theta)$ is the maximun set of traces such that $\forall \mathtt{s} \in \mathtt{trace}(\Theta), \Theta \vdash \mathtt{s}$.

Intuitively, a valid trace is a trace that Θ approves it.

Definition 27 (satisfaction up to observables). A process $\mathsf{Obs}_s(P)$ *synchronously satisfies* Θ, denoted $P \models_{\mathtt{sync}} \Theta$, when the following two conditions hold:

1. (output safety) $\mathsf{Obs}_s(P) \subset \mathtt{trace}(\Theta)$.

2.a (input consistency) Whenever $\mathtt{s} \in \mathsf{Obs}_s(P)$ and $\mathtt{s}\cdot\ell \in \mathtt{trace}(\Theta)$ where ℓ is an input, $\mathtt{s}\cdot\ell' \in \mathsf{Obs}_s(P)$ and ℓ' is an input with the same sender, label, and content type as ℓ, then $\mathtt{s}\cdot\ell \in \mathsf{Obs}_s(P)$.

A process P *asynchronously satisfies* Θ, denoted $P \models_{\mathtt{async}} \Theta$, if, after replacing each $\mathsf{Obs}_s(P)$ with $\mathsf{Obs}_a(P)$, it satisfies condition 1. above and the following condition:

2.b (input consistency) Whenever $\mathtt{s} \in \mathsf{Obs}_a(P)$ and $\mathtt{s} \cdot \ell \in \mathtt{trace}(\Theta)$ where ℓ is an input, then $\mathtt{s} \cdot \ell \in \mathsf{Obs}_a(P)$.

Note that, for synchronous process (2.a), it can accept a valid input only when it is ready to receive it; while for asynchronous process (2.b), it can accept and should accept a valid input. Intuitively, Definition 27 says that a process P synchronously (resp. asynchronously) satisfies Θ if, w.r.t. synchronous (resp. asynchronous) observables, P always does valid outputs as far as it receives valid inputs.

Example 28 (valid/invalid traces according to G_{assign}). We consider Θ_{assign} from Example 6 which uses the local SP, projected from G_{assign} (given in Section 2.3), for the server side. Then, for example,

$$s_2[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[S,B]!\mathrm{ans}\langle 1\rangle \cdot s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,B]!\mathrm{ans}\langle 2\rangle$$

is a valid trace of Θ_{assign}, but

$$s_2[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[S,B]!\mathrm{ans}\langle 2\rangle \cdot s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_1[B,S]!\mathrm{ans}\langle 1\rangle$$

is not its trace (violation is at the second step), because it is not permitted by a remote observer with Θ_{assign} when she observes it.

Lemma 29. $\mathtt{s} \in \mathsf{Obs}_s(P), P \models_{\mathtt{sync}} \Theta$ *implies* $\mathtt{s} \in \mathtt{trace}(\Theta)$.

Proof. Directly from Definition 27. ■

3.8 Well-formedness principles

Before introducing the well-formedness principles, the set of states of a predicate or an update is defined as follows.

Definition 30 (The set of states). $\mathit{field}(B)$ denotes the set of states (i.e. fields) names occurring in B. B can be a predicate (i.e. A) or an update (i.e. E).

Assume $\mathtt{p} \to \mathtt{q} : \{l_i(x_i : S_i)\langle A_i; E_i\rangle\langle A_i'; E_i'\rangle.G_i\}_{i\in I}$ is inside a context, with possibly preceding interactions. The following well-formedness principles, based on [4], stipulate consistency of global stateful protocols (i.e. SPs):

1. (a) $\forall i \in I, \mathit{field}(A_i') = \emptyset$; and (b) $\forall i \in I, A_i$ implies A_i'.
2. (history sensitivity) A_i and E_i only refer to interaction variables which $\mathtt{p}$, the sender, has sent or received before, as well as x_i. Similarly, A_i' and E_i' are for a receiver.
3. (temporal satisfiability) at each step, and for any state, there is always a branch i and a value x_i that satisfy A_i (hence A_i' at each step).

Principle (1-a) says that a predicate of a receiver is stateless. Two reasons for this requirement: first, if a receiving-side predicate relies on its own local state, then a sender may not be able to find a "proper" value to send; secondly, a sent message, which has been verified by a trusted observer (e.g. a system monitor), should not be judged invalid by the predicate at the receiver-side (or rejected by the receiver-side's system

monitor), otherwise, it is inconsistent because the judgement at the sender-side is not correct. Principle (1-b) says that, in every interaction, the predicate at sender always implies the predicate at the receiver: together with (1-a), it means that if a sender sends a message that satisfies the sender's predicate, then automatically the receiver's predicate is satisfied (the latter is however useful for the receiver to know what to expect).

Example 31 (sender's predicate always implies receiver's). Let state $\mathbf{f}_{\mathtt{q}}$ belong to role $\mathtt{q}$. Assume $\mathbf{f}_{\mathtt{q}} = 5$. Consider the following simple global specification which violates principle (1-a):

$$\mathtt{p} \to \mathtt{q} : (x : \mathsf{int})\langle \mathsf{true}\ ; \varepsilon \rangle \langle \mathbf{f}_{\mathtt{q}} < x\ ;\ \mathbf{f}_{\mathtt{q}} := x \rangle$$

When session participants apply this protocol for communication, if session-role $s[\mathtt{p}]$ sends a message with value less than 5, this action will be rejected and considered invalid by the receiver-side observer. However, it is a valid action according to the sender-side observer who knows nothing about the predicate at the receiver side. Principle (1-a) prevents this situation because all system monitors should be considered as having the same governance ability; which means, whenever a sender-side monitor approves an action according to the specification, the receiver-side monitor should also approve it because the monitor at the sender side is trusted.

For the similar reason, principle (1-b), which implies (1-a), ensures that whenever the sender-side observer approves an action, the receiver-side agrees with it. The following global specification violates principle (1-b):

$$\mathtt{p} \to \mathtt{q} : (x : \mathsf{int})\langle x > 10\ ; \varepsilon \rangle \langle x > 20\ ;\ \varepsilon \rangle$$

This specification is not reasonable because, if the sender sends a value of x in the range from 11 to 20, from the sender's viewpoint, she obeys the specification $x > 10$; however, she will be rejected by the receiver because she does not know that the receiver asks a value more than 20.

Principles 2 and 3 are based on the well-formedness rules in [4]. The following examples illustrate the principles 2 and 3, respectively.

Example 32 (history sensitivity). Let state $\mathbf{f}_{\mathtt{q}}$ belong to role $\mathtt{q}$. The following global specification violates history sensitivity

$$\begin{array}{l} \mathtt{p} \to \mathtt{q} : (x : \mathsf{int})\langle x > 10\ ; \varepsilon \rangle \langle x > 5\ ;\ \mathbf{f}_{\mathtt{q}} := x \rangle. \\ \mathtt{r} \to \mathtt{q} : (y : \mathsf{int})\langle y > x\ ; \varepsilon \rangle \langle y > x\ ;\ \mathbf{f}_{\mathtt{q}} := y \rangle \end{array}$$

because $\mathtt{r}$ does not know variable x, which is only shared by roles $\mathtt{p}$ and $\mathtt{q}$.

It can be revised to the global specification below

$$\begin{array}{l} \mathtt{p} \to \mathtt{q} : (x : \mathsf{int})\langle x > 10\ ; \varepsilon \rangle \langle x > 5\ ;\ \mathbf{f}_{\mathtt{q}} := x \rangle. \\ \mathtt{q} \to \mathtt{r} : (x). \\ \mathtt{r} \to \mathtt{q} : (y : \mathsf{int})\langle y > x\ ; \varepsilon \rangle \langle y > x\ ;\ \mathbf{f}_{\mathtt{q}} := y \rangle \end{array}$$

to respect history sensitivity.

Example 33 (temporal satisfiability). Let state $\mathbf{f}_{\mathtt{len}}$ belong to role $\mathtt{q}$. The global specification below violates temporal satisfiability

$$\begin{array}{l}\mathtt{p} \to \mathtt{q} : \mathrm{len}(x : \mathsf{int})\langle x > 5 \ ; \ \mathbf{f}_{\mathtt{len}} := x\rangle\langle x > 0 \ ; \ \varepsilon\rangle. \\ \mathtt{q} \to \mathtt{r} : \{\mathrm{walk}(y : \mathsf{int})\langle x < y < 10 \ ; \ \varepsilon\rangle\langle x < y < 30 \ ; \ \varepsilon\rangle, \\ \qquad\quad \mathrm{run}(y : \mathsf{int})\langle x < y < 50 \ ; \ \varepsilon\rangle\langle x < y < 60 \ ; \ \varepsilon\rangle\}\end{array}$$

because, when x is bigger than 50, which is actually possible since the predicate at role $\mathtt{p}$ only asks $x > 5$, neither branch "walk" or "run" can be selected for sending a proper y from role $\mathtt{q}$ to role $\mathtt{r}$ to satisfy the predicates specified at role $\mathtt{q}$ and $\mathtt{r}$ (although the predicate at role $\mathtt{q}$ implies the predicate at role $\mathtt{r}$).

A revised one that respects temoral satisfiability can be

$$\begin{array}{l}\mathtt{p} \to \mathtt{q} : \mathrm{len}(x : \mathsf{int})\langle 49 > x > 5 \ ; \ \mathbf{f}_{\mathtt{len}} := x\rangle\langle 49 > x > 0 \ ; \ \varepsilon\rangle. \\ \mathtt{q} \to \mathtt{r} : \{\mathrm{walk}(y : \mathsf{int})\langle x < y < 10 \ ; \ \varepsilon\rangle\langle x < y < 30 \ ; \ \varepsilon\rangle, \\ \qquad\quad \mathrm{run}(y : \mathsf{int})\langle x < y < 50 \ ; \ \varepsilon\rangle\langle x < y < 60 \ ; \ \varepsilon\rangle\}\end{array}$$

where $(49 > x > 5)$ specifies the possible values of x from 6 to 48. Even when a value in the range of 8 to 48 is sent from role $\mathtt{p}$ to role $\mathtt{q}$, $\mathtt{q}$ can select branch "run" to send y in the range of 9 to 49 to satify the predicate $x < y < 50$.

All examples treated in this paper are easily well-formed. *Henceforth we assume all global SPs we treat are well-formed.*

4 Theory of asynchronous specifications

4.1 Asynchronously verifiable specifications

We say Θ is *asynchronous* if it is suitable for a remote observer to verify the behaviours of a process. In this case, we do not want the conformance of a trace to change depending on an accidental reordering due to asynchrony: i.e. we want its validity to be robust w.r.t. legal permutations. In the followings, the specifications introduced in Section 2.2 are again used.

Definition 34 (asynchronously verifiable specification). We say Θ is *asynchronously verifiable* or simply *asynchronous* when $\mathtt{s} \in \mathtt{trace}(\Theta)$ and $\mathtt{s} \frown \mathtt{s}'$ imply $\mathtt{s}' \in \mathtt{trace}(\Theta)$.

To check violation of asynchrony of a specification, we only have to find a single acceptable trace whose permutation is not acceptable.

Example 35. Let T_{sync} be the local SP at server, projected from G_{sync}.

$$\Theta_{\mathsf{sync}} = \langle \Gamma'_{\mathsf{sync}}, a : \mathtt{I}(G_{\mathsf{sync}}[S]); \Delta'_{\mathsf{sync}}, \{s_i[S] : T_{\mathsf{sync}}\}_{i\in I}; D'_{\mathsf{sync}}, \mathbf{c}\rangle$$

where I is the set of indexes for the sessions using G_{sync}. This specification is *not* asynchronous because Θ_{sync} can accept trace

$$s_1[C,S]?\mathrm{req}(\varepsilon)\cdot s_1[S,C]!\mathrm{ans}(1)\cdot s_2[C,S]?\mathrm{req}(\varepsilon)s_2[S,C]!\mathrm{ans}(2)$$

but cannot accept the trace below which is permuted from the trace above

$$s_1[C,S]?\mathrm{req}(\varepsilon)\cdot s_2[C,S]?\mathrm{req}(\varepsilon)\cdot s_2[S,C]!\mathrm{ans}(2)\cdot s_1[S,C]!\mathrm{ans}(1)$$

On the other hand, checking asynchrony by Definition 34 means we should verify the property for all traces, which are usually infinitely many. Later we shall find methods by which we can approve the asynchrony of, for example, Θ_{assign} and all the corresponding specifications that use $G_{\mathsf{pcs}}/G_{\mathsf{ivc}}$ and G_{async}.

The following characterisation says that, if a synchronous observer recognises that P conforms to Θ, i.e. if P conforms to Θ synchronously, then an asynchronous observer will also do the same.

Proposition 36. *Θ is asynchronous iff, for each P, $P \models_{\mathtt{sync}} \Theta$ implies $P \models_{\mathtt{async}} \Theta$.*

Proof. For ($\Rightarrow$), assume $P \models_{\mathtt{sync}} \Theta$ and Θ is asynchronous. By definition, $P \models_{\mathtt{sync}} \Theta$ means $\mathsf{Obs}_s(P) \subset \mathtt{trace}(\Theta)$. Since Θ is asynchronous, by Definition 34, all legal permutation variants of $\mathtt{trace}(\Theta)$ are in $\mathtt{trace}(\Theta)$. This implies all legal permutation variants of $\mathsf{Obs}_s(P)$, which is $\mathsf{Obs}_a(P)$, are in $\mathtt{trace}(\Theta)$. Thus $P \models_{\mathtt{async}} \Theta$ as required.

For ($\Leftarrow$), since $\forall P, P \models_{\mathtt{sync}} \Theta$ implies $P \models_{\mathtt{async}} \Theta$, we know

$$\forall P, \forall \mathtt{s} \in \mathsf{Obs}_s(P) \subset \mathtt{trace}(\Theta) \text{ and } \mathtt{s} \curvearrowright \mathtt{s}' \text{ implies } \mathtt{s}' \in \mathsf{Obs}_a(P) \subset \mathtt{trace}(\Theta).$$

By Definition 34, Θ is asynchronous. □

The additional lemmas for proving Propositions 46 and 47 are in Appendix B. Let $\mathtt{I}(\mathtt{s})$ be the sequence of input actions of $\mathtt{s}$, $\mathtt{O}(\mathtt{s})$ be the sequence of output actions of $\mathtt{s}$, and $\mathtt{V}(\mathtt{s})$ be the sequence of values carried in $\mathtt{s}$. Let $\mathsf{num}(\mathtt{s})$ be the length of $\mathtt{s}$, and $\mathtt{prefix}(\mathtt{s})$ be the set of prefixes of $\mathtt{s}$. Let $\mathtt{ses}(\mathtt{s})$ be the set of sessions occuring in trace $\mathtt{s}$, and, similarly, $\mathtt{ses}(\Theta)$ be the set of sessions involving in Θ, i.e. those sessions obeying Θ. Note that, the size of set $\mathtt{ses}(\mathtt{s})$ is increasing when new sessions are created by a local process; similarly, the size of set $\mathtt{ses}(\Theta)$ is increasing when new sessions obeying Θ are created. Moreover, let $\mathtt{v}(\ell)$ be the value carried by action ℓ.

When a state is declared in a local specification, since a specification can dynamically verify sequence of actions during runtime, the state may be changed with the incoming or outgoing actions according to the update rules defined in the specification. Therefore, the current value of a state in a specification should be defined with actions of a process because these actions may make the value of state change.

Definition 37. Define $\mathtt{field}(\Theta)$ as the set of states declaired in Θ. If $\mathbf{c} \in \mathtt{field}(\Theta)$, then $\mathbf{c}$ is a state declaired in Θ.

Definition 38 (Current value of a state with respect to Θ and actions). Let $\mathbf{f} \in \mathtt{field}(\Theta)$ and $\mathtt{val}(\mathbf{f}, \Theta')$ be the value of state $\mathbf{f}$ when the configuration of specification is Θ'. Define the current value of state $\mathbf{f}$ as

$$\mathtt{current}(\mathbf{f}, \Theta, \mathtt{s}) = \mathtt{val}(\mathbf{f}, \Theta') \text{ where } \Theta \xrightarrow{\mathtt{s}} \Theta'.$$

Based on this definition, $\mathtt{current}(\mathbf{f}, \Theta, \mathtt{s})$ is the current value of state $\mathbf{f}$ according to Θ, which specifies the initial value of $\mathbf{f}$, when actions in $\mathtt{s}$ have happened.

Definition 39 (Initial value of a state with respect to Θ). Define $\mathtt{current}(\mathbf{f}, \Theta, \emptyset)$ be the initial value of state $\mathbf{f}$ which is declaired in Θ.

The initial value of a state should have been declaired in Θ. For example, as $\mathbf{f} \mapsto \mathsf{w}$ is declaired in Θ, we have $\mathtt{current}(\mathbf{f}, \Theta, \emptyset) = \mathsf{w}$.

Lemma 40. Assume $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$ and $\mathbf{c} \notin \Gamma, \Delta$, and $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \emptyset) = \mathsf{w}$. Assume every session s_i guided by G_{sync} has been established. Θ_{sync} is defined as below:

$$\begin{array}{l}\Theta_{\mathsf{sync}} = \langle \Gamma, \mathtt{ser} : \mathtt{I}(G_{\mathsf{sync}}[S]) \ ; \\ \qquad \Delta, \{s_i[S] : C?\mathrm{req}(\varepsilon)\langle \mathsf{true} \ ; \varepsilon\rangle . C!\mathrm{ans}(x : \mathsf{int})\langle x = \mathbf{c} \ ; \ \mathbf{c} := \mathbf{c} + 1\rangle\}_{i \in I} \ ; \\ \qquad \mathbf{c} \mapsto \mathsf{w}\rangle\end{array}$$

where I is the set of indexes of sessions. If $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, then for any $\mathtt{s}$ resulting from permutations $\mathtt{s}' \curvearrowright \mathtt{s}$, $\mathtt{s}$ satisfies the followings:

cond 1. If session $s \in \mathtt{ses}(\mathtt{s})$, then session $s \in \mathtt{ses}(\Theta_{\mathsf{sync}})$.
cond 2. $\forall \mathtt{s}'' \in \mathtt{prefix}(\mathtt{s}), \mathsf{num}(\mathtt{I}(\mathtt{s}'')) \geq \mathsf{num}(\mathtt{O}(\mathtt{s}''))$;
cond 3. If $\mathtt{s}_0 \cdot \ell \in \mathtt{s} \ \wedge \ \ell = s[S,B]!\mathrm{ans}(v)$, then $\exists \ \ell' \in \mathtt{s}_0, \ \ell' = s[B,S]?\mathrm{req}(\varepsilon)$.
cond 4. $\forall \ell, \ell' \in \mathtt{O}(\mathtt{s}), \ell \neq \ell'$, implies $\mathtt{v}(\ell) \neq \mathtt{v}(\ell')$;
cond 5. $\forall \ell \in \mathtt{O}(\mathtt{s}''), \mathtt{s}'' \in \mathtt{prefix}(\mathtt{s}), \mathsf{w} \leq \mathtt{v}(\ell) < \mathsf{num}(\mathtt{I}(\mathtt{s}'')) + \mathsf{w}$.

Proposition 41. Assume $\mathbf{c}', \mathbf{log} \in \mathtt{field}(\Theta_{\mathsf{assign}})$ and $\mathbf{c}', \mathbf{log} \notin \Gamma, \Delta$, and $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \emptyset) = \mathsf{w}' = \mathsf{w} - 1$, $\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \emptyset) = \{\ \}$. Assume every session s_i guided by G_{assign} has been established. Θ_{assign} is defined below:

$$\begin{array}{l}\Theta_{\mathsf{assign}} = \langle \Gamma, \mathtt{ser} : \mathtt{I}(G_{\mathsf{assign}}[S]) \ ; \\ \qquad \Delta, \{s_i[S] : C?\mathrm{req}(\varepsilon)\langle \mathsf{true} \ ; \ \mathbf{c}' := \mathbf{c}' + 1, \mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}\rangle . \\ \qquad\qquad C!\mathrm{ans}(x : \mathsf{int})\langle x \in \mathbf{log} \ ; \ \mathbf{log} := \mathbf{log} \setminus \{x\}\rangle\}_{i \in I} \ ; \\ \qquad \mathbf{c}' \mapsto \mathsf{w}', \mathbf{log} \mapsto \{\ \}\rangle\end{array}$$

where I is the set of indexes of sessions. $\mathtt{s}$ satisfies all conditions listed in Lemma 40, *if and only if* $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$.

Definition 42 (The strongest asynchronous specification). Let Θ_1 be the strongest asynchronous specificaiton w.r.t Θ_2 if

1. Θ_1 is asynchronous, and
2. $\mathtt{s} \in \Theta_1$ if and only if there exists $\mathtt{s}' \in \Theta_2$ such that $\mathtt{s}' \curvearrowright \mathtt{s}$.

Proposition 43. Θ_{assign} *is the strongest specification w.r.t* Θ_{sync}.

Proof. Based on Definition 42, it is proved directly from Lemma 40 and Proposition 41.□

Lemma 44. If $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$, then $\forall \mathtt{s} \in \mathsf{Obs}_a(P)$ implies $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$

Proof. Firslty, $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$ means that, for any $\mathtt{s}' \in \mathsf{Obs}_s(P)$, we have $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, and for any $\mathtt{s} \in \mathsf{Obs}_a(P)$, $\mathtt{s}$ is the result of legal permutations from some $\mathtt{s}' \in \mathsf{Obs}_s(P)$. In other words, for any $\mathtt{s} \in \mathsf{Obs}_a(P)$, there exists $\mathtt{s}'$ such that $\mathtt{s}' \in \mathsf{Obs}_s(P)$ and $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, then $\mathtt{s}$ can be reached from $\mathtt{s}' \curvearrowright \mathtt{s}$. Based on Lemma 40, $\mathtt{s}$ satisfies conditions listed there. Since for any $\mathtt{s}$ satisfies these conditions, by Lemma 41, it implies $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$, thus $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$. □

Definition 45 (input-output alternating sequence). Let ℓ_i^{in} be the ith input action and ℓ_i^{out} be the ith output action. $\mathtt{s}$ is called an input-output alternating sequence if it is in such a shape

$$\mathtt{s} = \ell_1^{in} \cdot \ell_1^{out} \cdot \ell_2^{in} \cdot \ell_2^{out} .....$$

where ℓ_1^{in} is the first action and the sequence of actions will end up with ℓ_k^{in} or $\ell_k^{in} \cdot \ell_k^{out}$ for some $k \geq 1$.

Proposition 46. *$P \models_{\mathtt{sync}} \Theta_{\mathsf{assign}}$ then $P \models_{\mathtt{async}} \Theta_{\mathsf{assign}}$.*

Proof. It can be proved by proving that Θ_{assign} is asynchronous through proving it is commutative with the definitions and propositions introduced later. Another direct proof without proving that Θ_{assign} is asynchronous is provided in Appendix B.

Proposition 47. Assume $\mathtt{s}$ is an input-output alternating sequence. If $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$, then $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$.

Proof. Please see Appendix B.

Note that, although Proposition 47 says that G_{assign} is an asynchronous verifiable specification that can realise synchronous behaviours w.r.t Θ_{sync}, the other way round is not true. The following example illustrates this point.

Example 48. Assume the initial value of state $\mathbf{c} \in \Theta_{\mathsf{assign}}$ is empty set (i.e. $\mathbf{c} \mapsto \{\}$) and the initial value of state $\mathbf{t}$ is 1. Also, assume the initial value of $\mathbf{c} \in \Theta_{\mathsf{sync}}$ is 1.

Consider a trace $\mathtt{s}$:

$$s_1[C,S]?\mathrm{req}(\varepsilon) \cdot s_2[C,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,C]!\mathrm{ans}(2)$$

then $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$ but $\mathtt{s} \notin \mathsf{Obs}_a(P), P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$ because there does not exist $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ such that $\mathtt{s}' \curvearrowright \mathtt{s}$.

The next result says that asynchronous verifiability is consistent with the asynchronous trace equivalence. Below let $P \approx_{\mathtt{async}} Q$ mean $\mathsf{Obs}_a(P) = \mathsf{Obs}_a(Q)$. In [13], we have shown how $\approx_{\mathtt{async}}$ (but not its synchronous counterpart) can be used for non-trivial optimising transformation.

Proposition 49. *If $P \approx_{\mathtt{async}} Q$ and $P \models_{\mathtt{async}} \Theta$ then $Q \models_{\mathtt{async}} \Theta$.*

Proof. Assume $P \approx_{\mathtt{async}} Q$ and $P \models_{\mathtt{async}} \Theta$. We first show $\mathsf{Obs}_a(Q) \subset \mathtt{trace}(\Theta)$. By assumption we have:

1. If $\mathtt{s} \in \mathsf{Obs}_a(Q)$ then $\mathtt{s} \in \mathsf{Obs}_a(P)$ based on $P \approx_{\mathtt{async}} Q$.
2. If $\mathtt{s} \in \mathsf{Obs}_a(P)$ then $\mathtt{s} \in \mathtt{trace}(\Theta)$ based on $P \models_{\mathtt{async}} \Theta$.

Hence if $\mathtt{s} \in \mathsf{Obs}_a(Q)$ then $\mathtt{s} \in \mathtt{trace}(\Theta)$, as required.

We next show input consistency. Suppose $\mathtt{s} \in \mathsf{Obs}_a(Q)$ and $\mathtt{s} \cdot \ell \in \mathtt{trace}(\Theta)$ where ℓ is input. $P \approx_{\mathtt{async}} Q$ implies $\mathtt{s} \in \mathsf{Obs}_a(P)$, and $P \models_{\mathtt{async}} \Theta$ implies $\mathtt{s} \cdot \ell \in \mathsf{Obs}_a(P)$. Again by $P \approx_{\mathtt{async}} Q$, we obtain $\mathtt{s} \cdot \ell \in \mathsf{Obs}_a(Q)$ as required. This concludes the proof. □

4.2 Asynchrony in specifications through commutativity

A basic issue in Definition 34 and its characterisation in Proposition 36 is that they do not directly mention the (intensional) structure of specifications. Thus it does not offer engineers insights as to how one may design her/his specifications. Extending the usage of the term in [10], we may call a criteria for specifications which a designer can use for ensuring robustness w.r.t. asynchrony, *healthiness condition*. The following definition is a first step towards such a criteria.

Definition 50 (confluence). Θ is *confluent* if, whenever $\Theta \xrightarrow{\mathtt{s}} \Theta'$, if $\Theta' \xrightarrow{\ell_1\ell_2} \Theta''$ and $\ell_2\ell_1 \frown \ell_1\ell_2$, then $\Theta' \xrightarrow{\ell_2\ell_1} \Theta''$ again.

I.e. the specification accepts the same sequence of values regardless of legal permutations and the resulting states are the same. Immediately confluence means asynchrony.

Lemma 51. Θ is asynchronous iff $\mathtt{s}\cdot\ell_1\cdot\ell_2 \in \mathtt{trace}(\Theta)$ and $\ell_1\cdot\ell_2 \frown \ell_2\cdot\ell_1$ imply $\mathtt{s}\cdot\ell_2\cdot\ell_1 \in \mathtt{trace}(\Theta)$ for each $\mathtt{s}$, ℓ_1 and ℓ_2.

Proof. For ($\Rightarrow$), it is trivial by Definitions 34 and 50. For ($\Leftarrow$), we want to prove that all cases should satisfy Definition 34, thus Θ is asynchronous. The reasonings are as follows:

1. If $\mathtt{s} = \emptyset$, which is not permutable, then $\forall \ell_1\cdot\ell_2 \in \mathtt{trace}(\Theta), \ell_1\cdot\ell_2 \frown \ell_2\cdot\ell_1 \in \mathtt{trace}(\Theta)$ satisfies Definition 34.

2. If $\mathtt{s} \neq \emptyset$, several cases are analysed below:
   (1) If $\mathtt{s} \not\frown$, which means $\mathtt{s}$ is not permutable to any other sequence of actions,
      (a) when neither ℓ_1 nor ℓ_2 can permute with $\mathtt{s}$, then the only permutable actions in $\mathtt{s}\cdot\ell_1\cdot\ell_2 \in \mathtt{trace}(\Theta)$ is $\ell_1\cdot\ell_2 \frown \ell_2\cdot\ell_1$, and $\mathtt{s}\cdot\ell_2\cdot\ell_1 \in \mathtt{trace}(\Theta)$. It satisfies Definition 34.

      (b) when there exists $\ell'_1,..,\ell'_j \in \mathtt{s}$ such that ℓ_2 (or ℓ_1) is permutable to $\ell'_1,..,\ell'_j$. Note that, since permutations are done one by one through legal unit permutation, $\mathtt{s}$ should be in the following shape:

$$\mathtt{s} = \mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell'_j,$$

      where $\mathtt{s}_0$ is not permutable at all (i.e. it means that $\mathtt{s}_0$ is empty, or no actions in $\mathtt{s}_0$ are permutable to each other and no action in $\mathtt{s}_0$ is permutable to any action of $\ell'_1,..,\ell'_j$.) Assume that is ℓ_2 can permute with $\ell'_1,..,\ell'_{j-1}$. Then

$$\begin{array}{l}\mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell'_j\cdot\ell_1\cdot\ell_2 \frown \\ \mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell'_j\cdot\ell_2\cdot\ell_1 \frown \\ \mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell_2\cdot\ell'_j\cdot\ell_1\end{array}$$

      Let $\mathtt{s}_0\cdot\ell'_1..\ell'_{j-1}\cdot\ell_2 = \mathtt{s}'\cdot\ell_2$ It is either
      (i) ℓ_2 can permute with some actions in $\ell'_1..\ell'_{j-1}$ then it is still under the analysis of case 2.(1)(b), or

(ii) ℓ_2 has permuted to all permutable actions in $\mathtt{s}'$ until $\mathtt{s}_0$, which is not permutable at all. Then it is case 2.(1)(a) that satisfies Definition 34.

Similarly, if $\ell_2 \cdot \ell_1 \curvearrowright \ell_1 \cdot \ell_2$ and ℓ_1 can permute with some actions in $\mathtt{s}$, it is again under the analysis of case 2.(1)(b).

(2) If there exists $\mathtt{s}'$ such that $\mathtt{s} \curvearrowright \mathtt{s}'$, $\mathtt{s}$ should be in the following shape:

$$\mathtt{s} = \mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2 \cdot \mathtt{s}'_2,$$

where $\ell''_1 \cdot \ell''_2 \curvearrowright \ell''_2 \cdot \ell''_1$ and no actions in $\mathtt{s}'_2$ are permutable to each other and no action in $\mathtt{s}'_2$ is permutable to any action in $\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2$. Then consider

$$\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2 \cdot \mathtt{s}'_2 \ell_1 \cdot \ell_2 \curvearrowright \mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2 \cdot \mathtt{s}'_2 \ell_2 \cdot \ell_1$$

It is either

(a) ℓ_2 is not permutable to $\mathtt{s}'_2$, then we only consider the part $\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2$, since $\mathtt{s} \cdot \ell_1 \cdot \ell_2 \in \mathtt{trace}(\Theta)$ implies $\mathtt{s} \in \mathtt{trace}(\Theta)$, which again implies $\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2 \in \mathtt{trace}(\Theta)$, so that $\mathtt{s}'_1 \cdot \ell''_2 \cdot \ell''_1 \in \mathtt{trace}(\Theta)$, and this case (i.e. $\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2$) is under the analysis of case 2.(1)(b).

(b) ℓ_2 is permutable to $\mathtt{s}'_1 \cdot \ell''_1 \cdot \ell''_2 \cdot \mathtt{s}'_2$. Then it is again in case 2.(1)(b).

Since all situations in 2.(1)(b) will finally go to 2.(1)(b)(ii), which means when the permutation terminates, it satisfies Definition 34. Overall, all possible cases satisfy Definition 34, thus Θ is asynchronous. □

Proposition 52. *If Θ is confluent then it is asynchronous.*

Proof. Directly from Definition 50 and Lemma 51. □

Note that the other way round is not true. The example below illustrates this point.

Example 53. Assume the initial values of states **c** and **t** are both 0, and the local SP for role $\mathtt{p}$ in session s is:

$$s[\mathtt{p}] : \mathtt{q}_1!\mathrm{ans}(\varepsilon)\langle \mathsf{true}; \mathbf{c} = 50\rangle \ . \ \mathtt{q}_2!\mathrm{ans}(\varepsilon)\langle \mathsf{true}; \mathbf{t} = \mathbf{c} + 10\rangle$$

Let $\ell_1 = s[\mathtt{p}, \mathtt{q}_1]!\mathrm{ans}(\varepsilon)$ and $\ell_2 = s[\mathtt{p}, \mathtt{q}_2]!\mathrm{ans}(\varepsilon)$. Then $\ell_1 \cdot \ell_2$ makes state **t** be 60, but $\ell_2 \cdot \ell_1$ makes state **t** be 10.

We can easily find a specification which is not confluent (for example, if a specification just does the same counting as G_{sync}). To check confluence, we still need to consider all possible transition derivatives of Θ. However we can observe that, in such a derivative, *the obligations used to check confluence are already present in Θ*. This suggests we only have to look at the obligations occurring in Θ and check their commutativity w.r.t. their legal unit permutations. This method demands designers to look at only Θ, so that it clearly helps her/his design process. The method treats a predicate and an update in an obligation as functions (operations) on state, as follows. Let $\dagger \in \{?, !\}$.

Definition 54 (predicate/update functions). Let $\xi \stackrel{\text{def}}{=} \mathtt{r} \dagger l(x:S)\langle A;E\rangle$ with the associated state D whose domain is $\mathbf{f}_1,...,\mathbf{f}_n$. W.l.o.g. we regard E to be a simultaneous substitution of the form $\mathbf{f}_1 := e_1,...,\mathbf{f}_n := e_n$. Then we define:

$$\mathtt{pred}(\xi) \stackrel{\text{def}}{=} \lambda x,\mathbf{f}_1,...,\mathbf{f}_n.A \qquad \mathtt{upd}(\xi) \stackrel{\text{def}}{=} \lambda x,\mathbf{f}_1,...,\mathbf{f}_n.\langle e_1,..,e_n\rangle$$

We call $\mathtt{pred}(\xi)$ (resp. $\mathtt{upd}(\xi)$) the *predicate function* (resp. *update function*) of ξ.

Example 55. Below we project G_{sync} and G_{assign} (all from §2) to the server. For simplicity we assume its local state only consists of those fields specified in global SP.

$$G_{\mathsf{sync}} \upharpoonright S = B?\mathrm{req}(\varepsilon)\langle \mathsf{true};\varepsilon\rangle \;.\; B!\mathrm{ans}(x:\ \mathsf{int})\langle x = \mathbf{c}\ ;\ \mathbf{c} := \mathbf{c}+1\rangle$$

$$G_{\mathsf{assign}} \upharpoonright S = B?\mathrm{req}(\varepsilon)\langle \mathsf{true};\mathbf{t} := \mathbf{t}+1\ \ \mathbf{c} := \mathbf{c}\cup\{\mathbf{t}\}\rangle \;.\; B!\mathrm{ans}(x:\ \mathsf{int})\langle x \in \mathbf{c}\ ;\ \mathbf{c} := \mathbf{c}\setminus\{x\}\rangle$$

Then the following table gives the functions induced by obligations in these local types.

	input	output
$G_{\mathsf{sync}} \upharpoonright S$	$\xi_0 \stackrel{\text{def}}{=} B?\mathrm{req}(\varepsilon)\langle \mathsf{true};\varepsilon\rangle$	$\xi_1 \stackrel{\text{def}}{=} B!\mathrm{ans}(x:\mathsf{int})\langle x=\mathbf{c};\mathbf{c}:=\mathbf{c}+1\rangle$
	$\mathtt{pred}(\xi_0) \stackrel{\text{def}}{=} \lambda\varepsilon,\mathbf{c}.(\mathsf{true})$	$\mathtt{pred}(\xi_1) \stackrel{\text{def}}{=} \lambda x,\mathbf{c}.(x=\mathbf{c})$
	$\mathtt{upd}(\xi_0) \stackrel{\text{def}}{=} \lambda\varepsilon,\mathbf{c}.\langle\varepsilon\rangle$	$\mathtt{upd}(\xi_1) \stackrel{\text{def}}{=} \lambda x,\mathbf{c}.\langle\mathbf{c}+1\rangle$
$G_{\mathsf{assign}} \upharpoonright S$	$\xi_2 \stackrel{\text{def}}{=} B?\mathrm{req}(\varepsilon)\langle \mathsf{true};\mathbf{t}:=\mathbf{t}+1\ \ \mathbf{c}:=\mathbf{c}\cup\{\mathbf{t}\}\rangle$	$\xi_3 \stackrel{\text{def}}{=} B!\mathrm{ans}(x:\ \mathsf{int})\langle x\in\mathbf{c}\ ;\ \mathbf{c}:=\mathbf{c}\setminus\{x\}\rangle$
	$\mathtt{pred}(\xi_2) \stackrel{\text{def}}{=} \lambda\varepsilon,\mathbf{c}.(\mathsf{true})$	$\mathtt{pred}(\xi_3) \stackrel{\text{def}}{=} \lambda x,\mathbf{c}.(x\in\mathbf{c})$
	$\mathtt{upd}(\xi_2) \stackrel{\text{def}}{=} \lambda\varepsilon,\mathbf{c}.\langle\mathbf{t}+1\ \ \mathbf{c}\cup\{\mathbf{t}\}\rangle$	$\mathtt{upd}(\xi_3) \stackrel{\text{def}}{=} \lambda x,\mathbf{c}.\langle\mathbf{c}\setminus\{x\}\rangle$

Once we can treat obligations as operations on state, we can define their commutativity. Since the commutativity we need is asymmetric (corresponding to asymmetric permutations induced by asynchrony, cf. Definition 20), we define semi-commutativity, which plays a key role in validating specifications later. A precursor of the following construction in a different setting is found in [7] (see §5 for discussions).

Definition 56 (semi-commutativity). Assume w.l.o.g., ξ_i and ξ_j use $\mathbf{f}$ as the field. Then we say ξ_i *commutes over* ξ_j if, for any message values v_i and v_j (for ξ_i and ξ_j), and any initial state w (for $\mathbf{f}$), the following conditions hold: If $\mathtt{pred}(\xi_i)(v_i,w)$ and $\mathtt{pred}(\xi_j)(v_j,\mathtt{upd}(\xi_i)(v_i,w))$ are both true, then

1. $\mathtt{pred}(\xi_j)(v_j,w)$ and $\mathtt{pred}(\xi_i)(v_i,\mathtt{upd}(\xi_j)(v_j,w))$ are both true.
2. $\mathtt{upd}(\xi_j)(v_j,\mathtt{upd}(\xi_i)(v_i,w)) = \mathtt{upd}(\xi_i)(v_i,\mathtt{upd}(\xi_j)(v_j,w))$.

If ξ_i commutes over ξ_j and vice versa, then we say ξ_i *and* ξ_j *are commutative*.

Note that the *if* statement says that it should satisfy the conditions above for *any* kind of initial values. I will explain the reason with Example 63 after Definition 58 and Proposition 61 are introduced.

Example 57. We show ξ_1 in Example 55 does not commute over itself. Let $\mathbf{f}=\mathbf{c}$. Then $\mathtt{pred}(\xi_1)(1,1)$, $\mathtt{pred}(\xi_1)(2,\mathtt{upd}(\xi_1)(1,1))$ and $\mathtt{pred}(\xi_1)(2,2)$ are all true, however $\mathtt{pred}(\xi_1)(2,1) = \mathsf{false}$. Similarly, ξ_0 does not commute over ξ_1 (however ξ_0,ξ_0 are commutative).

Using this notion, the healthiness condition for asynchronous specification can be concisely stated as follows. Below we say an obligation is *usable in* Θ if it occurs in a local SPin Θ or in the projection of a global SPin Θ to its potentially local role, where by "potentially local" we mean that the role has a potential to be played locally (e.g. for the global SPcarried by an input shared channel type (i.e. $a(s[\mathtt{p}] : G)$), only the specified role is potentially local).

Definition 58 (commutativity). Given Θ, let $\xi_1, .., \xi_n$ be all the obligations usable in Θ. Then we say Θ is *commutative* if the following conditions hold:

1. For (possibly identical) ξ_1' and ξ_2' from $\{\xi_1, .., \xi_n\}$, if both are inputs or both are outputs, then ξ_1' and ξ_2' are commutative.
2. For distinct ξ_1' and ξ_2' from $\{\xi_1, .., \xi_n\}$, if ξ_1' is an output and ξ_2' is an input then ξ_1' commutes over ξ_2'.

Possibly identical obligations means that it can be two obligations which are exactly the same, e.g. analysing ξ, ξ. In particular when there is only one obligation in Θ, this obligation, say ξ, should be analysed by itself ξ, ξ to know whether it commutes over itself.

Note that, based on Definition 56, Definition 58 states that, Θ is commutative if, for *any kind of initial values* in Θ, it satisfies the conditions given above. It is very important that it is for *any* initial value. Even though, assume a specification Θ is given with initial value w, Θ satisfies all conditions above with w but not with another initial value, say $w' \neq w$, then Θ is not commutative. I.e. Θ is action confluent when all obligations used in the specifications for the target process commute over each other up to legal permutations.

Before proving "Θ is commutative implies Θ is confluent", the following lemma is introduced.

Lemma 59. Assume $\Theta \xrightarrow{\mathtt{s}} \Theta'$ for some $\mathtt{s}$. If Θ is commutative then Θ' is commutative.

Proof. For ($\Rightarrow$), because $\Theta' \subset \Theta$, thus

1. for any obligation $\xi \in \Theta'$, $\xi \in \Theta'$ implies $\xi \in \Theta$, where Θ is commutative so that ξ itself is commutative.

2. for any two obligations $\xi, \xi' \in \Theta$ but $\xi' \notin \Theta'$ (an obligation may be finished since the conversation is finished. See Figure 3, the LTS of specification), since ξ itself should be commutative due to $\xi \in \Theta$ and Θ is commutative, ξ is still commutative in Θ' even it is alone.

Thus Θ' is commutative with new values of states resulting from $\Theta \xrightarrow{\mathtt{s}} \Theta'$ if Θ is commutative.

Note that the other way round (i.e. $\Leftarrow$) is not true because, while Θ' is derived from Θ, Θ can have more obligations than what Θ' has. Thus Θ' is commutative cannot imply that Θ is commutative.

Remark 60. Definitions 54 and 56 imply every obligation ξ corresponds to a valid action, and vise versa.

Now we can easily show

Proposition 61 (Commutativity implies confluence). *If Θ is commutative then it is confluent (hence asynchronous).*

Proof. Assume that Θ is commutative is given, let $\Theta \xrightarrow{\mathtt{s}} \Theta'$ for some $\mathtt{s}$. Assume $\Theta' \xrightarrow{\ell_1 \cdot \ell_2} \Theta''$ for some ℓ_1 and ℓ_2, and $\ell_1 \cdot \ell_2 \frown \ell_2 \cdot \ell_1$. We will show that $\Theta' \xrightarrow{\ell_2 \cdot \ell_1} \Theta'''$ and $\Theta'' = \Theta'''$. Assume the value carried by ℓ_i is v_i, $i = \{1,2\}$. Since Θ' is commutative based on Lemma 59:

1. When both of ℓ_1 and ℓ_2 are for output (resp. for input), these actions correspond to (possibly identical) obligations ξ_1 and ξ_2 from Θ', which are both for outputs (resp. for inputs). Assume the vector of values of states $\tilde{\mathbf{f}} \in \xi_1 \cup \xi_2$ is $\tilde{w}$, then because Θ' is commutative,

   $$\mathtt{pred}(\xi_1)(v_1, \tilde{w}) = \mathtt{pred}(\xi_2)(v_2, \mathtt{upd}(\xi_1)(v_1, \tilde{w})) = \mathsf{true}$$

   implies

   $$\mathtt{pred}(\xi_2)(v_2, \tilde{w}) = \mathtt{pred}(\xi_1)(v_1, \mathtt{upd}(\xi_2)(v_2, \tilde{w})) = \mathsf{true}$$

   which means $\Theta' \xrightarrow{\ell_2 \cdot \ell_1} \Theta'''$ for some Θ''', and moreover,

   $$\mathtt{upd}(\xi_1)(v_1, \mathtt{upd}(\xi_2)(v_2, \tilde{w})) = \mathtt{upd}(\xi_2)(v_2, \mathtt{upd}(\xi_1)(v_1, \tilde{w})) = \tilde{w'}$$

   which means $\ell_1 \cdot \ell_2$ and $\ell_2 \cdot \ell_1$ consume the same obligations and they reach the same value of states $\tilde{w'}$, so that we have $\Theta'' = \Theta'''$.

2. When ℓ_1 is for output and ℓ_2 is for input, these actions correspond to distinct obligations ξ_1 and ξ_2 from Θ', which are separately for output and for input. Assume the vector of values of states $\tilde{\mathbf{f}} \in \xi_1 \cup \xi_2$ is $\tilde{w}$, then

   $$\mathtt{pred}(\xi_1)(v_1, \tilde{w}) = \mathtt{pred}(\xi_2)(v_2, \mathtt{upd}(\xi_1)(v_1, \tilde{w})) = \mathsf{true}$$

   implies

   $$\mathtt{pred}(\xi_2)(v_2, \tilde{w}) = \mathtt{pred}(\xi_1)(v_1, \mathtt{upd}(\xi_2)(v_2, \tilde{w})) = \mathsf{true}$$

   which means $\Theta' \xrightarrow{\ell_2 \cdot \ell_1} \Theta'''$ for some Θ''', and moreover,

   $$\mathtt{upd}(\xi_1)(v_1, \mathtt{upd}(\xi_2)(v_2, \tilde{w})) = \mathtt{upd}(\xi_2)(v_2, \mathtt{upd}(\xi_1)(v_1, \tilde{w})) = \tilde{w'}.$$

   which means $\ell_1 \cdot \ell_2$ and $\ell_2 \cdot \ell_1$ consume the same obligations and they reach the same value of states $\tilde{w'}$, so that we have $\Theta'' = \Theta'''$.

□

Note that the other way round is not true. The following example explains the reason.

Example 62. Assume Θ has the following local SP, called T. The initial value of $\mathbf{c} \in \Theta$ is 2, and $\mathbf{c}$ will only be updated by the actions defined in T. It shows that Θ is confluent but not commutative.

$$\begin{aligned} T = B!\{&\mathrm{ans}_1(\varepsilon)\langle \mathsf{true}\ ;\ \mathbf{c} := \mathbf{c}+1\rangle\ ,\\ &\mathrm{ans}_2(\varepsilon)\langle c < 2\ ;\ \mathbf{c} := \mathbf{c}+\mathbf{c}\rangle\}.\\ &\mathsf{end}\end{aligned}$$

It is confluent because based on Definition 50, we assume Θ starts from initial value of $\mathbf{c}$, which is 2. Whenever $\Theta \xrightarrow{\mathtt{s}} \Theta' \xrightarrow{\ell_1\cdot\ell_2} \Theta''$ and $\ell_1\cdot\ell_2 \frown \ell_2\cdot\ell_1$, $\Theta \xrightarrow{\mathtt{s}} \Theta' \xrightarrow{\ell_2\cdot\ell_1} \Theta''$ is true because ℓ_1 or ℓ_2 never be with branch ans_2; otherwise it becomes $\Theta' \not\xrightarrow{\ell_1\cdot\ell_2}$, which is not the case considered.

On the other hand, because Definition 58 asks that Θ is commutative if it satisfies the commutativity conditions for *any kind of initial value* of $\mathbf{c}$, we know that it does not satisfy the conditions immediately when the initial value of $\mathbf{c}$ is smaller than 2 (i.e., 1, 0, -1, -2, ...). So Θ is not commutative.

The main motivation to have property of *commutativity* for Θ is that commutativity provides a concise and more easily verifiable way to decide if Θ is confluent, which automatically implies it is asynchronous-verifiable.

Compare Definition 50 and Definition 58, readers may feel curious about why *commutativity* should hold for any possible initial value of a state $\mathbf{f}$. The reason is because, generally, we do not know what the *invariants* of a given Θ are. To ensure that a Θ is commutative, we need to know if it satisfies all conditions for all possible initial values.

The idea is illustrated in the following example:

Example 63. Consider a specification, Θ, having a state $\mathbf{c}$ with initial value 8 and the following local SP, called T. Assume T is for endpoint $s[\mathtt{q}]$ and $\mathbf{c}$ will only be updated by the actions defined in T.

$$\begin{aligned} T = {}&\mathtt{p}_1!\mathrm{req}_1(\varepsilon)\langle \mathbf{c} < 10; \varepsilon\rangle.\\ &\mathtt{p}_2!\mathrm{req}_2(\varepsilon)\langle \mathbf{c} < 10; \mathbf{c} := \mathbf{c}+1\rangle.\\ &\mathsf{end}\end{aligned}$$

If commutativity is defined only for a particular initial value of $\mathbf{c} \in \Theta$, which is 8, then Θ is commutative; however, Θ is *not confluent*. Let $\ell_1 = s[\mathtt{q},\mathtt{p}_1]!\mathrm{req}_1(\varepsilon)$ and $\ell_2 = s[\mathtt{q},\mathtt{p}_2]!\mathrm{req}_2(\varepsilon)$, when several actions in trace $\mathtt{s}$ update $\mathbf{c}$ and $\mathbf{c}$ becomes 9, and

$$\Theta \xrightarrow{\mathtt{s}} \Theta' \xrightarrow{\ell_1\cdot\ell_2} \Theta''$$

is achieved, $\ell_1\cdot\ell_2 \frown \ell_2\cdot\ell_1$, but $\Theta' \not\xrightarrow{\ell_2\cdot\ell_1}$.

This is because the value of $\mathbf{c}$ is changing as actions happen. If Definition 58 only considers for some particular initial value of $\mathbf{c}$, then

1. the checking of commutativity is, however, done by a snapshot for any possible pair $\xi_i, \xi_j \in \Theta$. It can not reflect the real situation of the changes of value of $\mathbf{c}$;

2. or, we know exactly what the invariant of Θ is; e.g. the initial value of $\mathbf{c}$ should be bigger than or equal to 10. With this invariant, Θ is confluent. But generally, to find out the invariant(s) of a specification is difficult, which will be a part of the future works.

Therefore, it should be defined as,

Remark 64. Whenever any possible snapshot is taken with any kind of initial value of $\mathbf{c}$, Θ satisfies the conditions defined in Definition 58 then Θ is commutative.

As mentioned in 2. the method of checking commutativity can be strengthened by adding an invariant (including correlation among states) in state and checking that invariant continues to hold at each step. But it is currently not in the scope of this paper. We can now show all our example specifications except the one induced by G_{sync} is asynchronous. Below, let Θ_{async}'s shared environment contains $a : I(G_{\mathsf{async}}[S])$, and let Θ_{async}'s data storage contains $\mathbf{c} \mapsto \{\}$.

Proposition 65. *Θ_{async} and Θ_{assign} at server are both commutative, hence asynchronous.*

Proof. Commutativity of operations in Θ_{async} is immediate. For Θ_{assign}, we use ξ_2 and ξ_3 from Example 55. Then ξ_3 easily commutes over itself, because if $\mathtt{pred}(\xi_3)(v,\mathbf{c}) = \mathsf{true}$ and $\mathtt{pred}(\xi_3)(v',\mathtt{upd}(\xi_3)(v,\mathbf{c})) = \mathtt{pred}(\xi_3)(v',\mathbf{c}\setminus\{v\}) = \mathsf{true}$, so that

$$\mathtt{pred}(\xi_3)(v',\mathbf{c}) = \mathsf{true}.$$

When $v \neq v'$, we clearly have

$$\mathtt{pred}(\xi_3)(v,\mathtt{upd}(\xi_3)(v',\mathbf{c})) = \mathtt{pred}(\xi_3)(v,\mathbf{c}\setminus\{v'\}) = \mathsf{true}.$$

When $v = v'$, due to $\mathtt{pred}(\xi_3)(v,\mathbf{c}) = \mathsf{true}$ and $\mathtt{pred}(\xi_3)(v',\mathbf{c}\setminus\{v\})$ which together imply that $\mathbf{c}$ contains at least two elements whose values are $v = v'$, we clearly have $\mathtt{pred}(\xi_3)(v,\mathbf{c}\setminus\{v'\}) = \mathsf{true}$. Similarly $\mathtt{upd}(\xi_3)(v',\mathtt{upd}(\xi_3)(v,\mathbf{c})) = \mathtt{upd}(\xi_3)(v',\mathbf{c}\setminus\{v\}) = \mathbf{c}\setminus\{v\}\setminus\{v'\}$ which is equal to $\mathtt{upd}(\xi_3)(v,\mathtt{upd}(\xi_3)(v',\mathbf{c})) = \mathtt{upd}(\xi_3)(v,\mathbf{c}\setminus\{v'\})) = \mathbf{c}\setminus\{v'\}\setminus\{v\}$. We can similarly check ξ_2 is commutative over itself, and ξ_3 commutes over ξ_2 (but not the converse), as required. □

We can similarly check a specification induced by G_{pcs} and G_{ivc} are commutative.

4.3 Additional properties for asynchronous specification

With Definition 58 and the observations of obligations, some general properties of obligations are found:

Proposition 66. For a specification Θ, consider any two obligations ξ_1 and ξ_2 such that:

1. both are for inputs, or
2. both are for outputs, or
3. the first obligation is for output and the second one is for input,

including the case when $\xi_1 = \xi_2$. If for any $i, j \in \{1,2\}$, $\mathit{field}(A_i) \wedge \mathit{field}(E_j) = \emptyset$ where A_i occurs in ξ_i and E_j occurs in ξ_j, then Θ is commutative and hence asynchronously verifiable.

Proof. Because $\mathit{field}(A_i) \wedge \mathit{field}(E_j) = \emptyset$ for any pair of i and j, it means whenever a state is checked in a predicate then it is not updated in any update rule, and whenever a state is updated in an update then it is not checked in any predicate. It is proved immediately from Proposition 61 based on Definition 58.

The following simple example explains Proposition 66:

Example 67. Consider there are two (possibly identical) obligations ξ_1 and ξ_2. Assume they are both for output with the following shapes:

$$\xi_1 = \mathtt{p}_1!l(x_1 : S_1)\langle A_1; E_1\rangle, \xi_2 = \mathtt{p}_2!l'(x_2 : S_2)\langle A_2; E_2\rangle$$

Assume $\mathbf{f} = 10$ is checked in A_1 and $\mathbf{f} := 20$ is updated in E_2. E_2 updates $\mathbf{f}$ to a single value 20 when action $\mathtt{p}_2!$ takes place. Assume $s[\mathtt{p}, \mathtt{p}_1]!l(x_1) \cdot s'[\mathtt{p}, \mathtt{p}_2]!l'(x_2)$ is valid approved by ξ_1 and ξ_2. However, no matter what kind of values of x_1 and x_2 are sent, action $\mathtt{p}_2!$ always affects A_1, which implies that, when the order of actions is permuted to $s'[\mathtt{p}, \mathtt{p}_2]!l'(x_2) \cdot s[\mathtt{p}, \mathtt{p}_1]!l(x_1)$, because E_2 will affect the predicate for action $\mathtt{p}_1!$, ξ_1 and ξ_2 may not be commutative.

The above observation can be similarly applied for the obligations ξ_1 and ξ_2 which are input-input pair, or output-input pair. Note that, Proposition 73 also consider the pairs ξ_1, ξ_1 and ξ_2, ξ_2 for the situations when the same actions happen in two different sessions.

With the similarly structure of Proposition 66, now the conditions when *set* is used in a specification is proposed as follows.

Definition 68. $\mathit{field}^{set}(\cdot)$ maps an udpate (i.e. E) or a predicate (i.e. A) to the set of fields which are with data type set.

Recall specifications Θ_{async} and Θ_{assign}. For Θ_{async}, when one obligation's predicate checks $x \notin \mathbf{c}$ and its update has $\mathbf{c} := \mathbf{c} \cup \{x\}$, this obligation itself is commutative; similarly, for Θ_{assign}, when one obligation's predicate checks $x \in \mathbf{c}$ and its update has $\mathbf{c} := \mathbf{c} \setminus \{x\}$, this obligation itself is commutative. The following example shows that these kinds of shapes can be applied for more than one obligations:

Example 69. Assume state $\mathbf{f}$, whose data type is set, exists in obligations ξ_1 and ξ_2 with the following shapes,

$$\xi_1 = \mathtt{p}_1!(x_1)\langle x_1 \notin \mathbf{f}\,;\, \mathbf{f} := \mathbf{f} \cup \{x_1\}\rangle$$
$$\xi_2 = \mathtt{p}_2!(x_2)\langle x_2 \notin \mathbf{f}\,;\, \mathbf{f} := \mathbf{f} \cup \{x_2\}\rangle$$

then these two obligations are commutative. The reason is simply analysed as follows. Assume the initial value of $\mathbf{f}$ is w, and there are two actions ℓ_1 carrying value $v_1 = 5$ and ℓ_2 carring value $v_2 = 3$. If $\mathtt{pred}(\xi_1)(5, \mathsf{w}) = \mathsf{true}$ and $\mathtt{pred}(\xi_2)(3, \mathtt{upd}(\xi_1)(5, \mathsf{w})) = \mathsf{true}$, we have $\mathtt{pred}(\xi_2)(3, \mathsf{w}) = \mathsf{true}$ and $\mathtt{pred}(\xi_1)(5, \mathtt{upd}(\xi_2)(3, \mathsf{w})) = \mathsf{true}$ and

$$\mathtt{upd}(\xi_2)(3, \mathtt{upd}(\xi_1)(5, \mathsf{w})) = \mathtt{upd}(\xi_1)(5, \mathtt{upd}(\xi_2)(3, \mathsf{w})).$$

Actually, for any $v_1 \neq v_2$, the above equations always hold. For $v_1 = v_2$, for example $v_1 = 5 = v_2$, when $\texttt{pred}(\xi_1)(5,\mathsf{w}) = \mathsf{true}$, we have $\texttt{pred}(\xi_2)(5,\texttt{upd}(\xi_1)(5,\mathsf{w})) = \mathsf{false}$. Therefore ξ_1 and ξ_2 are commutative to each other.

Example 70. Assume state $\mathbf{f}$, whose data type is set, exists in obligations ξ_1 and ξ_2 with the following shapes,

$$\xi_1 = \mathrm{p}_1!(x_1)\langle x_1 \in \mathbf{f}\ ;\ \mathbf{f} := \mathbf{f} \setminus \{x_1\}\rangle$$
$$\xi_2 = \mathrm{p}_2!(x_2)\langle x_2 \in \mathbf{f}\ ;\ \mathbf{f} := \mathbf{f} \setminus \{x_2\}\rangle$$

then these two obligations are commutative to each other. It can be similarly analysed as the analysis in Example 69.

The following example further shows that the shapes of Θ_{async} and Θ_{assign} can be mixed up if they are applied to different states with data type set.

Example 71. Assume states $\mathbf{f}_1$ and $\mathbf{f}_2$ have data type set. When $\mathbf{f}_1$ and $\mathbf{f}_2$ exist in obligations ξ_1 and ξ_2, separately, with the following shapes:

$$\xi_1 = \mathrm{p}_1!(x_1)\langle x_1 \in \mathbf{f}_1\ ;\ \mathbf{f}_1 := \mathbf{f}_1 \setminus \{x_1\}\rangle$$
$$\xi_2 = \mathrm{p}_2!(x_2)\langle x_2 \notin \mathbf{f}_2\ ;\ \mathbf{f}_2 := \mathbf{f}_2 \cup \{x_2\}\rangle$$

then these two obligations are commutative to each other.

When $\mathbf{f}_1$ and $\mathbf{f}_2$ exist in obligations ξ with the following shapes,

$$\xi = \mathrm{p}!(x_1,x_2)\langle x_1 \in \mathbf{f}_1, x_2 \notin \mathbf{f}_2\ ;\ \mathbf{f}_1 := \mathbf{f}_1 \setminus \{x_1\}, \mathbf{f}_2 := \mathbf{f}_2 \cup \{x_2\}\rangle$$

ξ itself is commutative. The reason is because $\mathbf{f}_1$ and $\mathbf{f}_2$ are independent.

These examples show that, for a state with data type set, when it is involved in one or more obligations, if the obligations' predicates and udpates have some particular corresponding shapes (i.e. $\langle x \in \mathbf{f}; \mathbf{f} := \mathbf{f} \setminus \{x\}\rangle$ or $\langle x \notin \mathbf{f}; \mathbf{f} := \mathbf{f} \cup \{x\}\rangle$), the obligations are commutative.

However, the following example shows that if the corresponding shapes of predicate and update are not in the same obligation, but separately in two different obligations, these two obligations may not be commutative.

Example 72. Assume state $\mathbf{f}$ whose data type is set only exists in two following obligations:

$$\xi_1 = \mathrm{p}_1!(x_1)\langle x_1 \in \mathbf{f}\ ;\ \varepsilon\rangle$$
$$\xi_2 = \mathrm{p}_2!(x_2)\langle \mathsf{true}\ ;\ \mathbf{f} := \mathbf{f} \cup \{x_2\}\rangle$$

Assume the initial value of $\mathbf{f}$ is w. Simply check when $x_1 = 5$ and $x_2 = 5$, then we have $\texttt{pred}(\xi_1)(5,\mathsf{w}) = \mathsf{true}$ and $\texttt{pred}(\xi_2)(5,\texttt{upd}(\xi_1)(5,\mathsf{w})) = \mathsf{true}$, but $\texttt{pred}(\xi_1)(5,\texttt{upd}(\xi_2)(5,\mathsf{w})) = \mathsf{false}$.

Proposition 73. For $i \in \{1,2\}$, assume $\mathbf{f} \in \mathit{field}^{set}(E_i)$ and $\mathbf{f} := e$ is defined in E_i, where E_i belongs to ξ_i. Assume except $\mathbf{f}$, the states in e will not be updated by ξ_i, and the input/output variables of ξ_i is $x_{i1},...,x_{ik},...,x_{in}$. For obligations ξ_1 and ξ_2, which are possibly identical, such that

1. both are for inputs, or
2. both are for outputs, or
3. the first obligation is for output and the second one is for input

whenever *one and only one* of the following structure holds for both ξ_i, $i \in \{1,2\}$,

1. when $e \stackrel{\text{def}}{=} \mathbf{f} \cup \{f_{i1}(x_{i1}),..,f_{ik}(x_{ik}),...,f_{in}(x_{in})\}$ and $\mathbf{f} \in A_i$, A_i checks whether
$$f_{i1}(x_{i1}),...,f_{ik}(x_{ik}),...,f_{in}(x_{in}) \notin \mathbf{f}.$$
2. when $e \stackrel{\text{def}}{=} \mathbf{f} \setminus \{f_{i1}(x_{i1}),...,f_{ik}(x_{ik}),...,f_{in}(x_{in})\}$ and $\mathbf{f} \in A_i$, A_i checks whether
$$f_{i1}(x_{i1}),...,f_{ik}(x_{ik}),...,f_{in}(x_{in}) \in \mathbf{f}.$$

these two obligations are commutative.

Proof. Assume every state of data type set occurs in the obligations ξ_1 and ξ_2 satisfies either 1 or 2.

1. For a state $\mathbf{f}$ that satisfies 1, assume its initial value is w and let $i, j \in \{1,2\}$. Let $\tilde{v}_i = (v_{i1},..,v_{ik},...,v_{in})$, which will be checked by obligation ξ_i, similarly let $\tilde{v_j} = (v'_{j1},..,v'_{jk},...,v'_{jn})$, which will be checked by obligation ξ_j. Let $f_{ik}(x_{ik})\{v_{ik}/x_{ik}\} = v''_{ik}$ and $f_{jk}(x_{jk})\{v'_j k/x_{jk}\} = v''_{jk}$ for any $k = 1,...,n$. If $\texttt{pred}(\xi_i)(\tilde{v}_i, \mathsf{w}) = \mathsf{true}$ and $\texttt{pred}(\xi_j)(\tilde{v_j}, \texttt{upd}(\xi_i)(\tilde{v}_i, \mathsf{w})) = \mathsf{true}$, it means that
$$v''_{i1},...,v''_{ik},...,v''_{in} \notin \mathsf{w}$$
and $v''_{j1},...,v''_{jk},...,v''_{jn} \notin \mathsf{w} \cup \{v''_{i1},...,v''_{ik},...,v''_{in}\}$. They together imply, for any $k = 1,..,n$ and $m = 1,..,n$, $v''_{jk} \neq v''_{im}$, and also $v''_{j1},...,v''_{jk},...,v''_{jn} \notin \mathsf{w}$.

   Therefore, $\texttt{pred}(\xi_j)(\tilde{v_j}, \mathsf{w}) = \mathsf{true}$ due to $v''_{j1},...,v''_{jk},...,v''_{jn} \notin \mathsf{w}$, and
$$\texttt{pred}(\xi_i)(\tilde{v}_i, \texttt{upd}(\xi_j)(\tilde{v_j}, \mathsf{w})) = \texttt{pred}(\xi_i)(\tilde{v}_i, \mathsf{w} \cup \{v''_{j1},...,v''_{jk},...,v''_{jn}\}) = \mathsf{true}$$
   because $\texttt{pred}(\xi_i)(\tilde{v}_i, \mathsf{w}) = \mathsf{true}$ and for any $k = 1,..,n$ and $m = 1,..,n$, $v''_{jk} \neq v''_{im}$, which implies $v''_{i1},...,v''_{ik},...,v''_{in} \notin \mathsf{w} \cup \{v''_{j1},...,v''_{jk},...,v''_{jn}\}$.

   Moreover, because
$$\begin{aligned}&\texttt{upd}(\xi_i)(\tilde{v}_i, \texttt{upd}(\xi_j)(\tilde{v_j}, \mathsf{w}))\\ &= \texttt{upd}(\xi_i)(\tilde{v}_i, \mathsf{w} \cup \{v''_{j1},...,v''_{jk},...,v''_{jn}\})\\ &= \mathsf{w} \cup \{v''_{j1},...,v''_{jk},...,v''_{jn}\} \cup \{v''_{i1},...,v''_{ik},...,v''_{in}\}\end{aligned}$$
   and
$$\begin{aligned}&\texttt{upd}(\xi_j)(\tilde{v_j}, \texttt{upd}(\xi_i)(\tilde{v}_i, \mathsf{w}))\\ &= \texttt{upd}(\xi_j)(\tilde{v_j}, \mathsf{w} \cup \{v''_{i1},...,v''_{ik},...,v''_{in}\})\\ &= \mathsf{w} \cup \{v''_{i1},...,v''_{ik},...,v''_{in}\} \cup \{v''_{j1},...,v''_{jk},...,v''_{jn}\}\end{aligned}$$
   so that $\texttt{upd}(\xi_i)(\tilde{v}_i, \texttt{upd}(\xi_j)(\tilde{v_j}, \mathsf{w})) = \texttt{upd}(\xi_j)(\tilde{v_j}, \texttt{upd}(\xi_i)(\tilde{v}_i, \mathsf{w}))$.

2. For a state $\mathbf{f}$ that satisfies 2, with the similar settings, if $\texttt{pred}(\xi_i)(\tilde{v_i}, \mathsf{w}) = \mathsf{true}$ and $\texttt{pred}(\xi_j)(\tilde{v_j}, \texttt{upd}(\xi_i)(\tilde{v_i}, \mathsf{w})) = \mathsf{true}$, it means that

$$v''_{i1}, ..., v''_{ik}, ..., v''_{in} \in \mathsf{w}$$

and $v''_{j1}, ..., v''_{jk}, ..., v''_{jn} \in \mathsf{w} \setminus \{v''_{i1}, ..., v''_{ik}, ..., v''_{in}\}$. The later one implies,

$$v''_{j1}, ..., v''_{jk}, ..., v''_{jn} \in \mathsf{w}$$

thus $\texttt{pred}(\xi_j)(\tilde{v_j}, \mathsf{w}) = \mathsf{true}$. For any v''_{ik}, $k = 1, ..., n$,
(a) if $v''_{ik} \neq v''_{jm}$ for any $m = 1, ..., n$, then $v''_{ik} \in \mathsf{w} \setminus \{v''_{j1}, ..., v''_{jk}, ..., v''_{jn}\}$;
(b) if there exists $v''_{ik} = v''_{jm}$ for some k and m such that $k, m \in \{1, ..., n\}$, because $v''_{i1}, ..., v''_{ik}, ..., v''_{in} \in \mathsf{w}$ and $v''_{j1}, ..., v''_{jk}, ..., v''_{jn} \in \mathsf{w} \setminus \{v''_{i1}, ..., v''_{ik}, ..., v''_{in}\}$ together imply w contains at least two common elements whose values are $v''_{ik} = v''_{jm}$, so that (I) it is either $\mathbf{f}$ is a multiset and $v''_{ik} \in \mathsf{w} \setminus \{v''_{j1}, ..., v''_{jk}, ..., v''_{jn}\}$, or (II) this case never happens.

In summary, we have

$$\texttt{pred}(\xi_i)(\tilde{v_i}, \texttt{upd}(\xi_j)(\tilde{v_j}, \mathsf{w})) = \texttt{pred}(\xi_i)(\tilde{v_i}, \mathsf{w} \setminus \{v''_{j1}, ..., v''_{jk}, ..., v''_{jn}\}) = \mathsf{true}.$$

Moreover, because

$$\begin{aligned}
&\texttt{upd}(\xi_i)(\tilde{v_i}, \texttt{upd}(\xi_j)(\tilde{v_j}, \mathsf{w})) \\
&= \texttt{upd}(\xi_i)(\tilde{v_i}, \mathsf{w} \setminus \{v''_{j1}, ..., v''_{jk}, ..., v''_{jn}\}) \\
&= \mathsf{w} \setminus \{v''_{j1}, ..., v''_{jk}, ..., v''_{jn}\} \setminus \{v''_{i1}, ..., v''_{ik}, ..., v''_{in}\}
\end{aligned}$$

and

$$\begin{aligned}
&\texttt{upd}(\xi_j)(\tilde{v_j}, \texttt{upd}(\xi_i)(\tilde{v_i}, \mathsf{w})) \\
&= \texttt{upd}(\xi_j)(\tilde{v_j}, \mathsf{w} \setminus \{v''_{i1}, ..., v''_{ik}, ..., v''_{in}\}) \\
&= \mathsf{w} \setminus \{v''_{i1}, ..., v''_{ik}, ..., v''_{in}\} \setminus \{v''_{j1}, ..., v''_{jk}, ..., v''_{jn}\}
\end{aligned}$$

so that $\texttt{upd}(\xi_i)(\tilde{v_i}, \texttt{upd}(\xi_j)(\tilde{v_j}, \mathsf{w})) = \texttt{upd}(\xi_j)(\tilde{v_j}, \texttt{upd}(\xi_i)(\tilde{v_i}, \mathsf{w}))$.

Example 74. Consider there are two (possibly identical) obligations ξ_1 and ξ_2. Assume they are both for output with the following shapes:

$$\xi_1 = \mathtt{p}_1!l(x_1 : S_1)\langle x_1 + 3 \notin \mathbf{c}; E_1\rangle, \xi_2 = \mathtt{p}_2!l'(x_2 : S_2)\langle A_2; \mathbf{c} \cup \{x_2 + 3\}\rangle.$$

when action $\mathtt{p}_2!$ takes place, E_2 updates $\mathbf{f}$ with set operations: $\mathbf{f} := \mathbf{f} \cup \{x_2 + 3\}$. and checks $x_1 + 3 \notin \mathbf{c}$ since $\mathbf{c}$ will be updated by $x_2 + 3$. When $s[\mathtt{p}, \mathtt{p}_1]!l(x_1) \cdot s'[\mathtt{p}, \mathtt{p}_2]!l'(x_2)$ is a valid sequence of actions according to the specification and is permuted to $s[\mathtt{p}, \mathtt{p}_2]!l'(x_2) \cdot s[\mathtt{p}, \mathtt{p}_1]!l(x_1)$, the sequence of obligations $\xi_2.\xi_1$ can still prove the permuted actions.

Every output-output, input-input, output-input pair ξ_i, ξ_j in G_{async} or G_{assign} satisfies Proposition 73. Note that, in Proposition 73, although e_k can be a constant, say v_k, example below shows generally it is not the case.

Example 75. Assume two obligations are

$$\xi_1 = \mathtt{p}!(x)\langle v \notin \mathbf{f}; \mathbf{f} \cup \{v'\}\rangle, \xi_2 = \mathtt{q}!(y)\langle v' \notin \mathbf{f}; \mathbf{f} \cup \{v\}\rangle$$

which is vacuously satisfies Proposition 73 because there is no sequence of actions $\ell \cdot \ell'$ satisfies these obligations (or, we can say it does not even satisfy the assumption in Definition 56).

Compare the definition of commutativity and the permutation rules of Θ, as explained in the beginning of Section 3.5, they have different usage:

Remark 76. Attribute of commutativity is used for developers to know whether a Θ is asynchronous, while the permutation rules (defined in Definition 12) are used for an observer (e.g. system monitor) to have to observe the incoming and outgoing non-order-preserved actions.

The valuation of commutativity is essentially satisfiability of a formula whose free variables are universally quantified. Thus if the logic (for predicates) we use for our specification language is decidable, commutativity is decidable. In particular:

Proposition 77. *With the SP language given in §3 restricting operations on integers to be the constant* 0*, the addition and the subtraction, then the commutativity of specifications is decidable.*

Proof. By [19], a logic with sets, products and integers with additions is decidable, which subsumes the formulae used in Definition 56. □

We discuss practical implications of these results in the next section.

5 Related Work and Further Topics

Practical implications of the Theory The characterisation results in §4 offer not only a decision procedure for a rich subset of specifications, but also a basic insight on the design methodology for asynchronous specifications. In particular it sheds light on the use of operations on sets in our examples in §2. Because checking commutativity solely relies on the obligations occurring in protocols, adding the recursion to the syntax:

$$G \ ::= \ \ldots \mid \mu X.G \mid X \qquad T \ ::= \ \ldots \mid \mu X.T \mid X$$

does not change the nature of commutativity checking nor the resulting guarantee.

If Θ is asynchronous and a process behaves properly w.r.t. Θ synchronously, an asynchronous observer will also judge the induced (permuted) trace to be proper w.r.t. Θ. It is however easy to see that the converse is *not* true: consider a server that violates Θ_{assign} by responding 2 to the first request, 1 to the second, but these are delayed by asynchrony, leading to a valid trace when they arrive at the remote observer. A key consistency property is that any further legal permutation of this valid trace is again valid. For example, if a system monitor for the server is sitting between Client and Server, and if this monitor observes a valid trace of Server against the specification she has, Client will observe no worse behaviour. This monotonicity gives a basis for an application of the presented framework such as runtime monitoring.

Related works and further topics The semantic differences between synchronous and asynchronous communications have been studied for several decades: early works include [2, 6, 9, 11]. The permutations associated with asynchronous communication used in Definition 20 are noted in these works (and implicit in such work as [14]). Their more explicit presentation in the categorical setting is found in [18]. There is also a study in component validation based on asynchronous histories such as [17]. In spite of these precursors and close technical connection, the existing works (except [15] which however focuses on synchronous specifications and proof rules for their verifications) may not have pointed out the concrete semantic issues which stateful behavioural specifications and asynchronous observables can induce, and how this issue can be resolved through the interplay between synchronous and asynchronous semantics.

As observed in §4.2, a close analogue of commutativity of operations used for our characterisation result (Definition 56) appears in [7], where the authors study a method for checking commutativity (called *diamond connectivity*) of operations with pre-conditions in object-oriented programs, with a view to preventing the simultaneous issuance of these operations when they are not commutative. They translate the original model of methods in OCL to Alloy, which is analysed through simulation by Alloy Analyser. They do not (aim to) determine a class of specifications suitable for asynchronously communicating processes. In contrast, our aim is to stipulate a general class of specifications for communicating processes suitable for asynchronous observations, and identify its subclass amenable for automatic verification. Following this principle, we use a semi-commutativity to capture asymmetry in asynchronous communications: as seen in the Proposition 65 (the proofs are in Appendix), we crucially use this semi-commutativity when verifying G_{assign} is asynchronous.

Among further topics, we are currently exploring and analysing concrete forms of asynchronously verifiable specifications with different structures, informed by use cases from [16] as well as our theory, with a view to their usage in monitoring. One of the challenges is to find a solid (asynchronous) specification framework for inherently conflicting operations, such as two consecutive and overwriting updates on the same datum.

Acknowledgements We thank the reviewers for their valuable comments and our colleagues in Mobility Reading Group for discussions. This work is supported by Ocean Observatories Initiative [16] and EPSRC grants EP/F002114/1 and EP/G015481/1.

References

1. The Java Modeling Language (JML) homepage. http://www.jmlspecs.org/.
2. Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. In *Proc. CONCUR'96*, 1996.
3. Lorenzo Bettini et al. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
4. Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *LNCS*, pages 162–176, 2010.
5. Tzu chun Chen and Kohei Honda. Specifying stateful asynchronous properties for distributed programs. In *Concur2012*, LNCS. Springer, 2012. To appear.

6. Frank S. de Boer, Joost N. Kok, Catuscia Palamidessi, and Jan J. M. M. Rutten. The failure of failures in a paradigm for asynchronous communication. In *CONCUR*, pages 111–126, 1991.
7. Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. In *ISSTA'04*, pages 165–174, New York, NY, USA, 2004. ACM.
8. Tzu-Chun Chen et al. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC'11*, LNCS. Springer, 2012. To appear.
9. Jifeng He, Mark Josephs, and Tony Hoare. A theory of synchrony and asynchrony. In *Programming Concepts and Methods*, IFIP, pages 459–478, 1990.
10. C.A.R. Hoare and H. Jifeng. *Unifying theories of programming*. Prentice Hall series in computer science. Prentice Hall, 1998.
11. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *LNCS*, pages 133–147, 1991.
12. Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
13. Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *ECOOP'10*, volume 6183 of *LNCS*, pages 329–353. Springer-Verlag, 2010.
14. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, July 1978.
15. A multiparty multi-session logic. `http://www.cs.le.ac.uk/people/lb148/StatefulAssertions/main-long.pdf`.
16. Ocean Observatories Initiative (OOI). `http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/`.
17. Olaf Owe, Martin Steffen, and Arild B. Torjusen. Model Testing Asynchronously Communicating Objects using Modulo AC Rewriting. *ENCS*, 264(3):69–84, 2010.
18. Peter Selinger. First-order axioms for asynchrony. In *CONCUR*, pages 376–390, 1997.
19. Calogero G. Zarba. Combining sets with integers. In *FroCos*, pages 103–116, 2002.

A Examining G_{assign}

Based on §2.3, consider two requests from two sessions s_1 and s_2. Θ_{assign}, defined in Example 6 (page 10), is the local specification at server. Figure 4 lists the changes of Θ_{assign} as an action happens. Θ_{assign} is shown that it is actually a monitor. Because $\mathtt{server} : \mathtt{I}(G_{\mathsf{assign}}[S])$ does not change by actions, here we neglect it for simplicity. Note that Θ_{assign} detects (II) is an invalid trace at the second $s_2[S] : T'_{\mathsf{assign}}$.

B Appendix: Proofs and Additional Theorems

Lemma 78. For any $\mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, we always have

$$\mathsf{num}(\mathtt{O}(\mathtt{s}')) \leq \mathsf{num}(\mathtt{I}(\mathtt{s}'))$$

Proof. Simply by the definition of Θ_{sync} which specifies that inputs should always happen before outputs in the same session. Thus, in every possible prefix of $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, the number of inputs should be more than the number of outputs.□

action	1st	2nd	3rd	4th
(I) valid	$s_2[C,S]?\mathrm{req}(\varepsilon)$	$s_2[S,C]!\mathrm{ans}(1)$	$s_1[C,S]?\mathrm{req}(\varepsilon)$	$s_1[S,C]!\mathrm{ans}(2)$
(I)Θ_{assign}	$s_1[S]:T_{\mathsf{assign}}$, $s_2[S]:T'_{\mathsf{assign}}$; $\mathbf{t}\mapsto 1,\mathbf{c}\mapsto\{1\}$	$s_1[S]:T_{\mathsf{assign}}$, $s_2[S]:\mathsf{end}$; $\mathbf{t}\mapsto 1,\mathbf{c}\mapsto\{\}$	$s_1[S]:T'_{\mathsf{assign}}$, $s_2[S]:\mathsf{end}$; $\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{2\}$	$s_1[S]:\mathsf{end}$, $s_2[S]:\mathsf{end}$ $\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{\}$
(II) invalid	$s_2[C,S]?\mathrm{req}(\varepsilon)$	$s_2[S,C]!\mathrm{ans}(2)$	$s_1[C,S]?\mathrm{req}(\varepsilon)$	$s_1[C,S]!\mathrm{ans}(1)$
(II)Θ_{assign}	$s_1[S]:T_{\mathsf{assign}}$, $s_2[S]:T'_{\mathsf{assign}}$; $\mathbf{t}\mapsto 1,\mathbf{c}\mapsto\{1\}$	$s_1[S]:T_{\mathsf{assign}}$, $s_2[S]:T'_{\mathsf{assign}}$; $\mathbf{t}\mapsto 1,\mathbf{c}\mapsto\{1\}$	$s_1[S]:T'_{\mathsf{assign}}$, $s_2[S]:T'_{\mathsf{assign}}$; $\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{1,2\}$	$s_1[S]:\mathsf{end}$, $s_2[S]:T'_{\mathsf{assign}}$; $\mathbf{t}\mapsto 2,\mathbf{c}\mapsto\{2\}$

Fig. 4. The valid trace v.s. invalid trace w.r.t. G_{assign}

Lemma 79. Assume $\mathtt{s}\in\mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathbf{c}\in\mathtt{field}(\Theta_{\mathsf{sync}})$. Assume $\mathtt{current}(\mathbf{c},\Theta_{\mathsf{sync}},\emptyset)=\mathsf{w}$. Then $\forall\mathtt{s}'\in\mathtt{prefix}(\mathtt{s})$, we always have

$$\mathtt{current}(\mathbf{c},\Theta_{\mathsf{sync}},\mathtt{s}')=\mathsf{num}(\mathtt{O}(\mathtt{s}'))+\mathsf{w}$$

Proof. By induction, when $\mathsf{num}(\mathtt{O}(\mathtt{s}'))=0$, which means there is no output action, $0+\mathsf{w}=\mathtt{current}(\mathbf{c},\Theta_{\mathsf{sync}},\mathtt{s}')$. It is correct because no output action in $\mathtt{s}'$ means no update $\mathbf{c}:=\mathbf{c}+1$ applied according to Θ_{sync}.

Let Θ'_{sync} be the configuration from $\Theta_{\mathsf{sync}}\xrightarrow{\mathtt{s}'}\Theta'_{\mathsf{sync}}$. Assume $\mathsf{num}(\mathtt{O}(\mathtt{s}'))=k$ so that $\mathtt{current}(\mathbf{c},\Theta_{\mathsf{sync}},\mathtt{s}')=k+\mathsf{w}$, which means $\mathtt{current}(\mathbf{c},\Theta_{\mathsf{sync}},\mathtt{s}')=\mathtt{val}(\mathbf{c},\Theta'_{\mathsf{sync}})=k+\mathsf{w}$. For $\mathtt{s}'\cdot\ell$,

1. when ℓ is an input, $\mathsf{num}(\mathtt{O}(\mathtt{s}'\cdot\ell))=\mathsf{num}(\mathtt{O}(\mathtt{s}'))=k$, and according to Θ_{sync}, state $\mathbf{c}$ is *not* updated, which means

$$\mathtt{current}(\mathbf{c},\Theta_{\mathsf{sync}},\mathtt{s}'\cdot\ell)=\mathtt{current}(\mathbf{c},\Theta_{\mathsf{sync}},\mathtt{s}')=k+\mathsf{w}=\mathsf{num}(\mathtt{O}(\mathtt{s}'))+\mathsf{w}$$

2. when ℓ is an output, $\mathsf{num}(\mathtt{O}(\mathtt{s}'\cdot\ell))=\mathsf{num}(\mathtt{O}(\mathtt{s}'))+1=k+1$. According to Θ_{sync}, state $\mathbf{c}$ is updated by $\mathbf{c}:=\mathbf{c}+1$. Therefore, let Θ''_{sync} be the configuration from $\Theta'_{\mathsf{sync}}\xrightarrow{\ell}\Theta''_{\mathsf{sync}}$, we have

$$\mathtt{current}(\mathbf{c},\Theta_{\mathsf{sync}},\mathtt{s}'\cdot\ell)=\mathtt{val}(\mathbf{c},\Theta''_{\mathsf{sync}})=\mathtt{val}(\mathbf{c},\Theta'_{\mathsf{sync}})+1=(k+\mathsf{w})+1$$

which means

$$\mathtt{current}(\mathbf{c},\Theta_{\mathsf{sync}},\mathtt{s}'\cdot\ell)=(k+1)+\mathsf{w}=\mathsf{num}(\mathtt{O}(\mathtt{s}'\cdot\ell))+\mathsf{w}.$$

By induction, the statement is proved.□

Lemma 80. Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$. For any $\mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s})$.

Proof. According to Θ_{sync}, $\mathbf{c}$ is updated by $\mathbf{c} := \mathbf{c} + 1$ when an output happens, which means the value of $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}'')$ depends on the number of outputs happening in $\mathtt{s}''$. Because the number of outputs in $\mathtt{s}'$ is smaller than or equal to the number of outpus in $\mathtt{s}$, we always have $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s})$. □

Lemma 81. Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$. Then for any $\ell \in \mathtt{O}(\mathtt{s}'), \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{v}(\ell) < \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$.

Proof. By Lemma 80, we always have $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$ whenever $\mathtt{s}_0$ is a prefix of $\mathtt{s}'$ because $\mathsf{num}(\mathtt{s}_0) \leq \mathsf{num}(\mathtt{s}')$.

Let ℓ^* be the last output in $\mathtt{s}' = \mathtt{s}_0 \cdot \ell^* \cdot \mathtt{s}_1$, which means there is no output in $\mathtt{s}_1$. Then, according to Θ_{sync}, $\mathtt{v}(\ell^*) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0)$. This fact implies two things:

(a) for any other $\ell \in \mathtt{O}(\mathtt{s}')$, we know $\ell \in \mathtt{s}_0 = \mathtt{s}_0' \cdot \ell \cdot \mathtt{s}_0''$ because ℓ^* is the last output. According to Θ_{sync}, $\mathtt{v}(\ell) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0') \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) = \mathtt{v}(\ell^*)$.

(b) $\mathtt{v}(\ell^*) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0) < \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}_0 \cdot \ell^*) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$, so that
$$\mathtt{v}(\ell^*) < \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$$

Together with (a) and (b), for any $\ell \in \mathtt{s}', \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{v}(\ell) < \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$.
□

Lemma 82. Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$. Assume $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \emptyset) = \mathsf{w}$. $\forall \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, we have

$$\mathsf{w} \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') \leq \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{w}.$$

Proof. By Lemma 79, $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') = \mathsf{num}(\mathtt{O}(\mathtt{s}')) + \mathsf{w}$ so that $\mathsf{w} \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}')$. Moreover, by Lemma 78 which states that $\mathsf{num}(\mathtt{O}(\mathtt{s}')) \leq \mathsf{num}(\mathtt{I}(\mathtt{s}'))$, we have

$$\forall \mathtt{s}' \in \mathtt{prefix}(\mathtt{s}), \mathsf{w} \leq \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}') = \mathsf{num}(\mathtt{O}(\mathtt{s}')) + \mathsf{w} \leq \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{w}.$$

Then the proof is done. □

Lemma 83. Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$. Assume $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \emptyset) = \mathsf{w}$. $\forall \ell \in \mathtt{O}(\mathtt{s}'), \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, we always have $\mathsf{w} \leq \mathtt{v}(\ell) < \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{w}$.

Proof. Immediately from Lemmas 81 and 82. □

Lemma 84. Assume $\mathtt{s} = \mathtt{s}_1 \cdot \ell$ and $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$. If ℓ can permute with $\mathtt{s}_1$ such that $\mathtt{s}_1 \cdot \ell \curvearrowright \mathtt{s}_1' \cdot \ell'$, then $\mathsf{num}(\mathtt{I}(\mathtt{s}_1')) > \mathsf{num}(\mathtt{O}(\mathtt{s}_1'))$.

Proof. When $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, only when ℓ is input then it can permute with an output in $\mathtt{s}_1$. Therefore, $\mathsf{num}(\mathtt{I}(\mathtt{s}_1')) > \mathsf{num}(\mathtt{O}(\mathtt{s}_1'))$. □

Lemma 40 . Assume $\mathbf{c} \in \mathtt{field}(\Theta_{\mathsf{sync}})$ and $\mathbf{c} \notin \Gamma, \Delta$, and $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \emptyset) = \mathsf{w}$. Assume every session s_i guided by G_{sync} has been established. Θ_{sync} is defined as below:

$$\begin{aligned}\Theta_{\mathsf{sync}} = \langle & \Gamma, \mathtt{ser} : \mathtt{I}(G_{\mathsf{sync}}[S]) \ ; \\ & \Delta, \{s_i[S] : C?\mathsf{req}(\varepsilon)\langle \mathsf{true} \ ; \varepsilon\rangle.C!\mathsf{ans}(x : \mathsf{int})\langle x = \mathbf{c} \ ; \ \mathbf{c} := \mathbf{c} + 1\rangle\}_{i \in I} \ ; \\ & \mathbf{c} \mapsto \mathsf{w}\rangle\end{aligned}$$

where I is the index of sessions. If $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, then for any $\mathtt{s}$ resulting from permutations $\mathtt{s}' \curvearrowright \mathtt{s}$, $\mathtt{s}$ satisfies the followings:

cond 1. If $s \in \mathtt{ses}(\mathtt{s})$, then $s \in \mathtt{ses}(\Theta_{\mathsf{sync}})$.
cond 2. $\forall \mathtt{s}'' \in \mathtt{prefix}(\mathtt{s}), \mathsf{num}(\mathtt{I}(\mathtt{s}'')) \geq \mathsf{num}(\mathtt{O}(\mathtt{s}''))$;
cond 3. If $\mathtt{s}_0 \cdot \ell \in \mathtt{s} \ \wedge \ \ell = s[S,B]!\mathsf{ans}(v)$, then $\exists \ \ell' \in \mathtt{s}_0, \ \ell' = s[B,S]?\mathsf{req}(\varepsilon)$.
cond 4. $\forall \ell, \ell' \in \mathtt{O}(\mathtt{s}), \ell \neq \ell'$, then $\mathtt{v}(\ell) \neq \mathtt{v}(\ell')$;
cond 5. $\forall \ell \in \mathtt{O}(\mathtt{s}''), \mathtt{s}'' \in \mathtt{prefix}(\mathtt{s}), \mathsf{w} \leq \mathtt{v}(\ell) < \mathsf{num}(\mathtt{I}(\mathtt{s}'')) + \mathsf{w}$.

Proof. First of all, for any $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, $\mathtt{s}'$ satisfies all these conditions: condition 0 is satisfied because $\mathtt{s}'$ is valid according to Θ_{sync} which means, for any $s \in \mathtt{ses}(\mathtt{s}')$, Θ_{sync} should include this session so that it can verify the actions involving this session. Conditions 1,2, 3 are satisfied simply by the definition of Θ_{sync} and condition 4 is satisfied by Lemmas 83. So the main proof below is to show, when $\mathtt{s}'$ satisfies these five conditions and $\mathtt{s}' \curvearrowright \mathtt{s}$, $\mathtt{s}$ satisfies all these conditions.

For cond 1. Since any permutation of $\mathtt{s}'$ does not add or subtract any session from $\mathtt{s}'$, any trace $\mathtt{s}$ permuted from $\mathtt{s}' \curvearrowright \mathtt{s}$ has the sessions as same as $\mathtt{s}'$ does. We have any $s \in \mathtt{ses}(\mathtt{s})$ implies $s \in \mathtt{ses}(\mathtt{s}')$, which again implies $s \in \mathtt{ses}(\Theta_{\mathsf{sync}})$.

For cond 2. By the permutation rules defined in Definition 20,

(a) for any action sequence $\ell \cdot \ell' \in \mathtt{s}'$, where ℓ is output and ℓ' is input, if they are permutable and are permuted by doing $\ell \cdot \ell' \curvearrowright \ell' \cdot \ell$, so that make $\mathtt{s}' \curvearrowright \mathtt{s}$, $\mathtt{s}$ still satisfies cond 2. because the number of inputs in every $\mathtt{s}'' \in \mathtt{prefix}(\mathtt{s})$ is more than the number of inputs in every $\mathtt{s}''' \in \mathtt{prefix}(\mathtt{s}')$.

(b) for any action sequence $\ell \cdot \ell' \in \mathtt{s}'$, which are both inputs (resp. outputs), after permutations $\ell \cdot \ell' \curvearrowright \ell' \cdot \ell$, which leads to $\mathtt{s}' \curvearrowright \mathtt{s}$, the number of inputs or the number of outputs in every $\mathtt{s}'' \in \mathtt{prefix}(\mathtt{s})$ is as same as that in every $\mathtt{s}''' \in \mathtt{prefix}(\mathtt{s}')$.

For cond 3. Based on $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, since the permutation of $\mathtt{s}' \curvearrowright \mathtt{s}$ only moves input actions ahead of output actions in $\mathtt{s}$, $\mathtt{s}$ still satisfies cond 3.

For cond 4. Since $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, every outputted value of a trace in $\mathtt{s}'$ is unique. Then for every $\mathtt{s}$ permuted from $\mathtt{s}' \curvearrowright \mathtt{s}$, every outputted value of $\mathtt{s}$ is still unique.

For cond 5. Based on $\mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ and $\mathtt{s}'$ satisfies cond 5, when $\mathtt{s}$ is the result from permutation $\mathtt{s}' \curvearrowright \mathtt{s}$, because the valid permutations only make inputs move ahead of outputs, or two inputs (resp. two outputs) permute to each other, overall the

permutations only increase the number of input actions ahead of output actions in $\mathtt{s}$, while the value of any output action is not changed. By Lemma 84, $\mathtt{s}$ satisfies cond 5. □

Remark 85. Based on Lemma 40, some trace $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{async}})$ may not satisfy these conditions because $\mathtt{trace}(\Theta_{\mathsf{async}})$ is much bigger than the set

$$\{\mathtt{s} \mid \mathtt{s}' \in \mathtt{trace}(\Theta_{\mathsf{sync}}), \mathtt{s}' \curvearrowright \mathtt{s}\}.$$

Definition 86 (contiguous series). For a set $\{v_0, ..., v_n\}$ of numbers, where v_0 is the minimum element and v_n is the maximum element. If , except v_n, any $v \in \{v_0, ..., v_n\}$, $\exists v' \in \{v_0, ..., v_n\}$ such that $v' = v + 1$, then the set is called a contiguous series.

Lemma 87. The states $\mathbf{c}'$ and $\mathbf{log}$ in Θ_{assign} repectively have the following attributes:

1. For state $\mathbf{c}'$, for any $\mathtt{s}$,
   (a) when ℓ is an input, $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}\cdot\ell) = \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}) + 1$.
   (b) when ℓ is an output, $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}\cdot\ell) = \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s})$.

2. For state $\mathbf{log}$, for any $\mathtt{s}$,
   (a) when ℓ is an input, $\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathtt{s}\cdot\ell) = \mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathtt{s}) \cup \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}\cdot\ell)$.
   (b) when ℓ is an output, $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}\cdot\ell) = \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}) \setminus \{\mathtt{v}(\ell)\}$.

Proposition 41 . Assume $\mathbf{c}', \mathbf{log} \in \mathtt{field}(\Theta_{\mathsf{assign}})$ and $\mathbf{c}', \mathbf{log} \notin \Gamma, \Delta$, and $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \emptyset) = \mathsf{w}' = \mathsf{w} - 1$, $\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \emptyset) = \{\ \}$. Assume every session s_i guided by G_{assign} has been established. Θ_{assign} is defined below:

$$\begin{aligned}\Theta_{\mathsf{assign}} = \langle \Gamma, \mathtt{ser} : \mathtt{I}(G_{\mathsf{assign}}[S])\ ;\ & \\ \Delta, \{s_i[S] : C?\mathrm{req}(\varepsilon)\langle \mathsf{true}\ ;\ \mathbf{c}' := \mathbf{c}' + 1, \mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}\rangle. & \\ C!\mathrm{ans}(x : \mathsf{int})\langle x \in \mathbf{log}\ ;\ \mathbf{log} := \mathbf{log} \setminus \{x\}\rangle\}_{i \in I}\ ; & \\ \mathbf{c}' \mapsto \mathsf{w}', \mathbf{log} \mapsto \{\ \}\rangle & \end{aligned}$$

where I is the index of sessions. $\mathtt{s}$ satisfies all conditions listed in Lemma 40, *if and only if* $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$.

Proof. (For $\Rightarrow$): Assume $\mathtt{s}$ satisfies all conditions listed in Lemma 40. We only need to make sure that every output action in $\mathtt{s}$ always satisfies the predicate at output obligation (i.e. $x \in \mathbf{log}$). The reasonings are as follows:

(1) Based on cond 1, 1, and 2, for any session $s \in \mathtt{ses}(\mathtt{s})$, its actions are either $s[C,S]?\mathrm{req}(\varepsilon)\cdot s[S,C]!\mathrm{ans}(v)$ or $s[C,S]?\mathrm{req}(\varepsilon)$. Thus the sequence of actions satisfy that Θ_{assign} defines, for a session, its input action should happen before its ouput.

(2) Although an input is anyway valid because of the predicate true at input obligation, with the update $\mathbf{c}' := \mathbf{c}' + 1$ and $\mathbf{log} := \mathbf{log} \cup \mathbf{c}'$, it affects the elements of set $\mathbf{log}$ and thus may affect the judgement of predicate $x \in \mathbf{log}$ when replacing x with the value of the upcoming output action.

For the effects caused by inputs, the statement, we have, for $\mathtt{s}$ satisfying the five conditions:

Claim. $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}) = \mathsf{num}(\mathtt{I}(\mathtt{s})) + \mathsf{w}'$.

This is because $\mathbf{c}'$ is a counter at input obligation whose predicate is true, any output action does not update $\mathbf{c}'$ and, for any $\mathtt{s}$, $\mathbf{c}'$ will be updated whenever an input happens. Note *it does not matter* whether $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$ or not (at this point this has not been proved).

This statement can be proved similarly as the proof in Lemma 79. In Lemma 79, the statement is $\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathtt{s}) = \mathsf{num}(\mathtt{O}(\mathtt{s})) + \mathsf{w}'$ because $\mathbf{c}$ is a counter at output obligation in Θ_{sync}, while $\mathbf{c}'$ is a counter at input obligation in Θ_{assign}. Remember, the predicate in Θ_{assign} for any input is true and any output action cannot update $\mathbf{c}'$. We only need to replace every $\mathbf{c}$ with $\mathbf{c}'$ and Θ_{sync} with Θ_{assign}, and change $\mathsf{num}(\mathtt{O}(\mathtt{s}))$ to $\mathsf{num}(\mathtt{I}(\mathtt{s}))$, other steps of the proof are the same.

(3) For any output action ℓ in $\mathtt{s}$, since more than one inputs may have happened before ℓ according to cond 2 and cond 3, to satisfy the predicate $x \in \mathbf{log}$ when replacing x with $\mathtt{v}(\ell)$, the followings should be considered:

For any $\mathtt{s}$, let $\mathtt{s} = \mathtt{s}_0 \cdot \ell \cdot \mathtt{s}_1$, where $\mathtt{s}_0$ only has inputs so that ℓ is the first output.

(a) Up to $\mathtt{s}_0$, Θ_{assign} accepts $\mathtt{s}_0$ because it only has inputs. According to updates $\mathbf{c}' := \mathbf{c}' + 1$ and $\mathbf{log} := \mathbf{log} \cup \mathbf{c}'$ at input obligation, where $\mathbf{c}'$ has initial value w', and $\mathbf{log}$ has initial value $\{\ \}$, the minimum (first) element in $\mathbf{log}$ is $\mathsf{w}' + 1 = \mathsf{w}$, we therefore have,

$$\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathtt{s}_0) = \{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{s}_0)\} = \{\mathsf{w}, ..., (\mathsf{w} - 1) + \mathsf{num}(\mathtt{s}_0)\}$$

Thus for any $\mathtt{s}'' \in \mathtt{prefix}(\mathtt{s}_0)$,

$$\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}'' \cdot \ell') = \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}'') + 1$$

whenever ℓ' is an input. Thus we have $\{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{s}_0)\}$ is a contiguous set with minimun $\mathsf{w}' + 1$ and maximum $\mathsf{w}' + \mathsf{num}(\mathtt{s}_0)$.

(b) Up to $\mathtt{s}_0 \cdot \ell$, by cond 5,

$$\mathsf{w} \leq \mathtt{v}(\ell) < \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \ell)) + \mathsf{w} = \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{w} = \mathsf{num}(\mathtt{s}_0) + \mathsf{w}$$

which implies $\mathtt{v}(\ell) \in \mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathtt{s}_0)$ so that Θ_{assign} accepts $\mathtt{s}_0 \cdot \ell$. Note that, for any $\mathtt{s}$, $\mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \ell)) = \mathsf{num}(\mathtt{I}(\mathtt{s}_0))$ whenever ℓ is an output.

(c) Assume for any $\mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, $\mathtt{s}' = \mathtt{s}_f \cdot \ell$ where $\mathtt{s}_f$ may contain inputs or at most one output or both. Assume Θ_{assign} accepts $\mathtt{s}_f$ and let $\{\ell \mid \ell \in \mathtt{O}(\mathtt{s}_f)\} = \mathtt{out}(\mathtt{s}_f)$ be the set of output *values* happening in $\mathtt{s}_f$. So, up to $\mathtt{s}_f$, we have

$$\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathtt{s}_f) = \{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_f))\} \setminus \mathtt{out}(\mathtt{s}_f)$$

where $\{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_f))\}$ is a contiguous series.

Firstly, it needs to prove that $\{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_f))\}$ is a contiguous series. Assume $\mathtt{s}_f = \mathtt{s}_0 \cdot \mathtt{s}'_f$ where $\mathtt{s}_0$ only contains inputs, then it is straightforward that $\{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0))\}$ is a contiguous series simply from (a). The following claims that, if a set is a contiguous series, then after adding new elements to it because of inputs, the resulting set is still a contiguous series.

Claim. For $\mathtt{s}_0 \cdot \mathtt{s}'_1 = \mathtt{s}', \mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$ where $\mathtt{s}_0$ only contains inputs, and let $\mathtt{out}(\mathtt{s}'_1)$ be the set of output actions happening in $\mathtt{s}'_1$. If $\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathtt{s}_0) = \{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0))\}$, then

$$\begin{aligned}
&\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathtt{s}_0 \cdot \mathtt{s}'_1) \\
&= \{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0))\} \cup \\
&\quad \{\mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1))\} \setminus \mathtt{out}(\mathtt{s}'_1) \\
&= \{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1))\} \setminus \mathtt{out}(\mathtt{s}'_1) \\
&= \{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \mathtt{s}'_1))\} \setminus \mathtt{out}(\mathtt{s}'_1)
\end{aligned}$$

and the set $\{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \mathtt{s}'_1))\}$ is a contiguous series.

This claim can be proved by the following reasoning: according to Θ_{assign}, once an input happens, it updates $\mathbf{c}' := \mathbf{c}' + 1$ and $\mathbf{log} := \mathbf{log} \cup \mathbf{c}'$. The new elements added by the inputs in $\mathtt{s}'_1$ are in the range from $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}_0 \cdot \ell)$, where ℓ is the first input of $\mathtt{s}'_1$, to $\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}_0 \cdot \mathtt{s}'_1)$ where

$$\begin{aligned}
&\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}_0 \cdot \ell) \\
&= \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}_0) + 1 \\
&= \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + 1
\end{aligned}$$

and

$$\begin{aligned}
&\mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}_0 \cdot \mathtt{s}'_1) \\
&= \mathtt{current}(\mathbf{c}', \Theta_{\mathsf{assign}}, \mathtt{s}_0) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1)) \\
&= \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1))
\end{aligned}$$

Because $\mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + \mathsf{num}(\mathtt{I}(\mathtt{s}'_1)) = \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \mathtt{s}'_1))$, the set of new elements added to $\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathtt{s}_0)$ is

$$\{\mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0)) + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathtt{s}_0 \cdot \mathtt{s}'_1))\}$$

which is a contiguous series set by the same reasoning of (a). Because the minimum element of set $\{\mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0)) + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0 \cdot \mathsf{s}_1'))\}$, which is $\mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0)) + 1$, is one more bigger than the maximum element of set $\{\mathsf{w}', ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0))\}$, which is $\mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0))$, the union of these two contiguous sets is still a contiguous series set. Moreover,

$$\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathsf{s}_0 \cdot \mathsf{s}_1') = \{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0 \cdot \mathsf{s}_1'))\} \setminus \mathtt{out}(\mathsf{s}_1')$$

simply according to the updates $\mathbf{c}' := \mathbf{c}' + 1$ and $\mathbf{log} := \mathbf{log} + \mathbf{c}'$ for every input, and the update $\mathbf{log} := \mathbf{log}\{\mathtt{v}(\ell)\}$ for every output ℓ.

Continue to prove that $\mathsf{s}_f \cdot \ell$ is anyway accepted by Θ_{assign}. When ℓ is an input, Θ_{assign} accepts $\mathsf{s}_f \cdot \ell$ because Θ_{assign} accepts any input. When ℓ is an output, then by cond 5, $\mathsf{w} \leq \mathtt{v}(\ell) < \mathsf{w} + \mathsf{num}(\mathtt{I}(\mathsf{s}_f \cdot \ell)) = \mathsf{w} + \mathsf{num}(\mathtt{I}(\mathsf{s}_f))$, we know that $\mathtt{v}(\ell) \in \{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_f))\}$ because it is a contiguous series set and $\mathsf{w}' = \mathsf{w} - 1$.

Further, by cond 4 we know $\mathtt{v}(\ell)$ is different from any value of actions in $\mathtt{out}(\mathsf{s}_f)$. They together imply $\mathtt{v}(\ell)$ is in

$$\mathtt{current}(\mathbf{log}, \Theta_{\mathsf{assign}}, \mathsf{s}_f) = \{\mathsf{w}' + 1, ..., \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_f))\} \setminus \mathtt{out}(\mathsf{s}_f)$$

Therefore, by induction, every output in $\mathtt{s}$ is accepted by Θ_{assign}, which means every $\mathtt{s}$ satisfies all conditions in Lemma 40, $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$.

(For $\Leftarrow$): Assume $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$ is given. The reasonings are as follows:

1. $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$ means that the (fee) sessions declared in $\mathtt{s}$ has been validly established (as $s_i, i \in I$), which implies $\mathtt{s}$ satisfies cond 1.

2. By the definition of Θ_{assign}, if $\mathtt{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$, then the inputs should always happen before the outputs in the same session, thus $\mathtt{s}$ satisfies cond 2, and cond 3.

3. **log** is updated while an input happens by updating $\mathbf{log} := \mathbf{log} \cup \{\mathbf{c}'\}$; therefore, for any v_1 and v_2 in **log**, v_1 is always different from v_2 simply because whenever an input happens, $\mathbf{c}'$ updates to a different value (i.e. $\mathbf{c}' := \mathbf{c}' + 1$) and this new value is added to set **log** (i.e. $\mathbf{log} := \mathbf{log} \cup \mathbf{c}'$). Since every outputted value v in $\mathtt{s}$ should be selected from set **log** through verifying the predicate $v \in \mathbf{log}$, and, when outputting v, **log** is updated by $\mathbf{log} := \mathbf{log} \setminus \{v\}$, they together ensure that for any outputs ℓ and ℓ' in $\mathtt{s}$, $\mathtt{v}(\ell) \neq \mathtt{v}(\ell')$. Thus $\mathtt{s}$ satisfies condition 3.

4. By the definition of Θ_{assign}, the initial value of $\mathbf{c}'$ is w' and $\mathbf{c}'$ is updated by increasing one when an input happens. Therefore, for every $\mathtt{s}' \in \mathtt{prefix}(\mathtt{s})$, again, $\mathtt{current}(\mathbf{c}', \mathsf{w}', \mathtt{s}')$ defined before is used to represent the lastest value of $\mathbf{c}'$ when the actions in $\mathtt{s}'$ have happened. Then we know

$$\mathtt{current}(\mathbf{c}', \mathsf{w}', \mathtt{s}') = \mathsf{num}(\mathtt{I}(\mathtt{s}')) + \mathsf{w}'$$

because the lastest value of $\mathbf{c}'$ is the summation of the number of inputs happening in s' plus the starting value w'. Since for every outputted value v in $\mathsf{s}' \in \mathtt{prefix}(\mathsf{s})$, there exists an input belonging to the same session of v ahead of it, and it is selected from set **log** by verifying through predicate $v \in \mathbf{log}$, v is always in the range from $\mathsf{w}'+1$, the minimum element in **log** from $\mathbf{c}' := \mathbf{c}'+1$ and $\mathbf{log} := \mathbf{log} \cup \mathbf{c}'$, to $\mathsf{num}(\mathtt{I}(\mathsf{s}'))+\mathsf{w}'$, the maximum element in **log**. Therefore we have

$$\mathsf{w}'+1 \leq v \leq \mathtt{current}(\mathbf{c}', \mathsf{w}', \mathsf{s}') = \mathsf{num}(\mathtt{I}(\mathsf{s}'))+\mathsf{w}',$$

which implies

$$\mathsf{w} \leq v \leq \mathsf{num}(\mathtt{I}(\mathsf{s}'))+\mathsf{w}' < \mathsf{num}(\mathtt{I}(\mathsf{s}'))+\mathsf{w}$$

due to $\mathsf{w}' = \mathsf{w}-1$. So that s satisfies cond 5.

5. Therefore, $\forall \mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$, s satisfies the conditions defined in Lemma 40.□

Proposition 46. $P \models_{\mathtt{sync}} \Theta_{\mathsf{assign}}$ then $P \models_{\mathtt{async}} \Theta_{\mathsf{assign}}$.

Proof. It can be proved simply by proving that Θ_{assign} is commutative, or proved by the following reasoning. Let ℓ^{in} represent an input action, and ℓ^{out} represent an output action. For every $\mathsf{s} \in \mathsf{Obs}_s(P)$, in which $P \models_{\mathtt{sync}} \Theta_{\mathsf{assign}}$, according to Θ_{assign}, it is an input-output alternating sequence with the following shape

$$\mathsf{s} = \ell_1^{in} \cdot \ell_1^{out} \cdot \ell_2^{in} \cdot \ell_2^{out} .....$$

which will end up with ℓ_k^{in} or $\ell_k^{in} \cdot \ell_k^{out}$ for some $k \neq 1$, and satisfies all conditions listed in Lemma 40. Since the permutations only move inputs ahead of outputs, no condition is affected by the permutations. Thus $\forall \mathsf{s} \curvearrowright \mathsf{s}', \mathsf{s}' \in \mathtt{trace}(\Theta_{\mathsf{assign}})$ i.e. $\mathsf{s}' \in \mathsf{Obs}_a(P), P \models_{\mathtt{async}} \Theta_{\mathsf{assign}}$.□

Lemma 88. Let ℓ_i^{out} be the ith output action and ℓ_i^{in} be the ith input action. $\mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$ if and only if

1. $\ell_1^{out} = \mathsf{w}$, and
2. for any $\ell_i^{out} \in \mathsf{s}$, there exist ℓ_i^{in} positioning before ℓ_i^{out}, and
3. for any $\ell_i^{out}, \ell_{i-1}^{out} \in \mathsf{s}$, $\mathtt{v}(\ell_i^{out}) = \mathtt{v}(\ell_{i-1}^{out})+1$.

Proof. For ($\Rightarrow$), according to Θ_{sync} and Lemma 79, for any $\mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, let $\mathsf{s} = \mathsf{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathsf{s}_0' \cdot \ell_i^{out} \cdot \mathsf{s}_1$, where s_0' only contains input actions because ℓ_i^{out} is the next output action after ℓ_{i-1}^{out}. The three conditions are proved as followings.

1. For the value of the first output according to Θ_{sync}, let s_0 contains only inputs and $i = 2$, then we have $\mathtt{v}(\ell_1^{out}) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathsf{s}_0) = \mathsf{num}(\mathtt{O}(\mathsf{s}_0))+\mathsf{w} = \mathsf{w}$.

2. According to Θ_{sync}, whenever $\mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$, there always exists ℓ_{i-1}^{in} in s_0 and ℓ_i^{in} in s_0'.

3. Moreover we have

$$\mathrm{v}(\ell_{i-1}^{out}) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathsf{s}_0) = \mathsf{num}(\mathtt{O}(\mathsf{s}_0)) + \mathsf{w}$$

and

$$\mathrm{v}(\ell_{i}^{out}) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathsf{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathsf{s}_0') = \mathsf{num}(\mathtt{O}(\mathsf{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathsf{s}_0')) + \mathsf{w}$$

where

$$\mathsf{num}(\mathtt{O}(\mathsf{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathsf{s}_0')) + \mathsf{w} = \mathsf{num}(\mathtt{O}(\mathsf{s}_0)) + 1 + \mathsf{w}$$

therefore we have $\mathrm{v}(\ell_i^{out}) = \mathrm{v}(\ell_{i-1}^{out}) + 1$ for any ℓ_i^{out} and ℓ_{i-1}^{out} in such s.

For ($\Leftarrow$), the prove is done by induction. Assume $\mathsf{s}_0'' \in \mathtt{prefix}(\mathsf{s})$, $\mathsf{s}_0'' \in \mathtt{trace}(\Theta_{\mathsf{sync}})$. We can always find such s_0'' by letting s_0'' be a trace which only contains input actions. Consider $\mathsf{s}_0'' \cdot \ell_1^{out}$, where ℓ_1^{out} is the first output so that s_0'' contains only inputs,

$$\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathsf{s}_0'') = \mathsf{num}(\mathtt{O}(\mathsf{s}_0'')) + \mathsf{w} = \mathsf{w} = \mathrm{v}(\ell_1^{out})$$

which means that $\mathsf{s}_0 \cdot \ell_1^{out}$ is valid to Θ_{sync} because the outputed value is equal to the current value of state $\mathbf{c}$.

Let $\mathsf{s}_0'' = \mathsf{s}_0 \cdot \ell_{i-1}^{out}$ where s_0 may contain some inputs and outpus or only inputs. Without loss of generality, assume s_0'' is valid according to Θ_{sync}. It means

$$\mathrm{v}(\ell_{i-1}^{out}) = \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathsf{s}_0) = \mathsf{num}(\mathtt{O}(\mathsf{s}_0)) + \mathsf{w}$$

Consider trace $\mathsf{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathsf{s}_0' \cdot \ell_i^{out}$, where ℓ_i^{out} is the next output of ℓ_{i-1}^{out} and s_0' only contains inputs. Because Θ_{sync} always approves input actions, it implies that $\mathsf{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathsf{s}_0'$ is valid according to Θ_{sync}. Because for every $\mathrm{v}(\ell_i^{out}), \mathrm{v}(\ell_{i-1}^{out}) \in \mathsf{s}$, we always have $\mathrm{v}(\ell_i^{out}) = \mathrm{v}(\ell_{i-1}^{out}) + 1$, therefore, we have the following equations:

$$\begin{aligned}
\mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathsf{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathsf{s}_0') &= \mathsf{num}(\mathtt{O}(\mathsf{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathsf{s}_0')) + \mathsf{w} \\
&= \mathsf{num}(\mathtt{O}(\mathsf{s}_0)) + 1 + \mathsf{w} \\
&= \mathtt{current}(\mathbf{c}, \Theta_{\mathsf{sync}}, \mathsf{s}_0) + 1 \\
&= \mathrm{v}(\ell_{i-1}^{out}) + 1 = \mathrm{v}(\ell_i^{out})
\end{aligned}$$

which means the outputed value is equal to the current value of state $\mathbf{c}$, so that $\mathsf{s}_0 \cdot \ell_{i-1}^{out} \cdot \mathsf{s}_0' \cdot \ell_i^{out}$ is valid according to Θ_{sync}. By Induction, for any s satisfying these three conditions, we know $\mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$.□

Proposition 47 . Assume s is an input-output alternating sequence. If $\mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$, then $\mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$.

Proof. It is proved by induction. $\mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$ if and only if s satisfies all conditions listed in Lemma 40 gives the following reasonings:

1. For the first output action ℓ_1^{out}, by cond 5, $\mathsf{w} \leq \mathrm{v}(\ell_1^{out}) < \mathsf{num}(\mathtt{I}(\ell_1^{in} \cdot \ell_1^{out})) + \mathsf{w}$. Due to $\mathsf{w}' = \mathsf{w} - 1$ and according to Θ_{assign}, we have

$$\mathrm{v}(\ell_1^{out}) \in \{\mathsf{w}' + 1\} = \{\mathsf{w}\}$$

which means $\mathrm{v}(\ell_1^{out}) = \mathsf{w}$.

2. For the second output action ℓ_2^{out}, by cond 5, $\mathsf{w} \leq \mathrm{v}(\ell_2^{out}) < \mathsf{num}(\mathtt{I}(\ell_1^{in} \cdot \ell_1^{out} \cdot \ell_2^{in} \cdot \ell_2^{out})) + \mathsf{w}$. Due to $\mathsf{w}' = \mathsf{w} - 1$ and according to Θ_{assign}, we have

$$\begin{aligned}\mathrm{v}(\ell_2^{out}) \in \{\mathsf{w}' + 1, \mathsf{w}' + \mathsf{num}(\mathtt{I}(\ell_1^{in} \cdot \ell_1^{out} \cdot \ell_2^{in} \cdot \ell_2^{out}))\} \setminus \{\mathsf{w}' + 1\} \\ \{\mathsf{w}' + 1, \mathsf{w}' + 2\} \setminus \{\mathsf{w}' + 1\}\end{aligned}$$

which means $\mathrm{v}(\ell_2^{out}) = \mathsf{w}' + 2 = \mathsf{w} + 1$.

3. Let $\mathsf{s}_0 = \ell_1^{in} \cdot \ell_1^{out} \ldots \ell_{i-1}^{in} \cdot \ell_{i-1}^{out}$. Assume for the ith output action ℓ_i^{out} we have

$$\mathrm{v}(\ell_i^{out}) \in \{\mathsf{w}' + 1, \ldots \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0 \cdot \ell_i^{in} \cdot \ell_i^{out}))\} \setminus \{\mathsf{w}' + 1, \ldots, \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0))\}$$

so that

$$\mathrm{v}(\ell_i^{out}) \in \{\mathsf{w}' + 1, \ldots, \mathsf{w}' + (i-1), \mathsf{w}' + i\} \setminus \{\mathsf{w}' + 1, \ldots, \mathsf{w}' + (i-1)\}$$

which means $\mathrm{v}(\ell_i^{out}) = \mathsf{w}' + i = \mathsf{w} + (i-1)$.

Then for the $(i+1)$th output action ℓ_{i+1}^{out}, by cond 5, $\mathsf{w} \leq \mathrm{v}(\ell_{i+1}^{out}) < \mathsf{num}(\mathtt{I}(\mathsf{s}_0 \cdot \ell_i^{in} \cdot \ell_i^{out} \cdot \ell_{i+1}^{in} \cdot \ell_{i+1}^{out})) + \mathsf{w}$. Due to $\mathsf{w}' = \mathsf{w} - 1$ and according to Θ_{assign}, we have

$$\begin{aligned}\mathrm{v}(\ell_{i+1}^{out}) \in \{\mathsf{w}' + 1, \ldots \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0 \cdot \ell_i^{in} \cdot \ell_i^{out} \cdot \ell_{i+1}^{in} \cdot \ell_{i+1}^{out}))\} \\ \setminus \{\mathsf{w}' + 1, \ldots, \mathsf{w}' + \mathsf{num}(\mathtt{I}(\mathsf{s}_0 \cdot \ell_i^{in} \cdot \ell_i^{out}))\}\end{aligned}$$

so that

$$\mathrm{v}(\ell_{i+1}^{out}) \in \{\mathsf{w}' + 1, \ldots, \mathsf{w}' + i, \mathsf{w}' + i + 1\} \setminus \{\mathsf{w}' + 1, \ldots, \mathsf{w}' + i\}$$

therefore we have $\mathrm{v}(\ell_{i+1}^{out}) = \mathsf{w}' + (i+1) = \mathsf{w} + i$.

Finally we have $\mathrm{v}(\ell_{i+1}^{out}) = \mathrm{v}(\ell_i^{out}) + 1$. Therefore, by Lemma 88, $\mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{sync}})$. □

Note that, although Proposition 41 says that the set of traces satisfying conditions listed in Lemma 40 is equal to $\mathtt{trace}(\Theta_{\mathsf{assign}})$, and Proposition 47 says that G_{assign} is an asynchronously verifiable specification realising synchronous behaviours w.r.t Θ_{sync}, $\mathsf{s} \in \mathtt{trace}(\Theta_{\mathsf{assign}})$ does not imply $\mathsf{s} \in \mathsf{Obs}_a(P)$ such that $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$. It is simply because $\mathtt{trace}(\Theta_{\mathsf{assign}})$ is bigger than $\mathsf{Obs}_a(P)$ in which $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$. The following example illustrates this point.

Example 89. Consider a sequence of actions below:

$$s_1[C,S]?\mathrm{req}(\varepsilon) \cdot s_2[C,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,C]!\mathrm{ans}(v)$$

if it is a trace in $\mathsf{Obs}_a(P)$ such that $P \models_{\mathtt{sync}} \Theta_{\mathsf{sync}}$, then v can only be 1; however if it is a trace in $\mathtt{trace}(\Theta_{\mathsf{assign}})$, v can be either 1 or 2.

C Proofs for SPs' commutativity

Here we illustrate the main obligations in the SPs introduced in examples (see § 2 and § 2.4). Note that, only the state(s) updated at every input/output action appear(s) in the obligations.

Proof (G_{sync} is not asynchronous). Assume $\xi^{\mathtt{i}}_{\mathsf{sync}}$ (for input) and $\xi^{\circ}_{\mathsf{sync}}$ (for output) are the notations for the obligations in G_{sync}.

$$\begin{array}{rcl} \xi^{\mathtt{i}}_{\mathsf{sync}} & \stackrel{\text{def}}{=} & B?\mathrm{req}(\varepsilon)\langle \mathsf{true}; \varepsilon\rangle \\ \xi^{\circ}_{\mathsf{sync}} & \stackrel{\text{def}}{=} & B!\mathrm{ans}(x:\mathsf{int})\langle x = \mathbf{c}; \mathbf{c} := \mathbf{c}+1\rangle \end{array}$$

$$\begin{array}{rcl} \mathtt{pred}(\xi^{\mathtt{i}}_{\mathsf{sync}}) & \stackrel{\text{def}}{=} & \lambda\varepsilon,\mathbf{c}.(\mathsf{true}) \\ \mathtt{pred}(\xi^{\circ}_{\mathsf{sync}}) & \stackrel{\text{def}}{=} & \lambda x,\mathbf{c}.(x=\mathbf{c}) \\ \mathtt{upd}(\xi^{\mathtt{i}}_{\mathsf{sync}}) & \stackrel{\text{def}}{=} & \lambda\varepsilon,\mathbf{c}.\langle\varepsilon\rangle \\ \mathtt{upd}(\xi^{\circ}_{\mathsf{sync}}) & \stackrel{\text{def}}{=} & \lambda x,\mathbf{c}.\langle\mathbf{c}+1\rangle \end{array}$$

Then for $\xi^{\circ}_{\mathsf{sync}}, \xi^{\circ}_{\mathsf{sync}}$, we start from $\mathbf{c}$'s current value is $w = 1$:

$$\mathbf{c} = w = 1.$$

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{sync}})(1,1) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{sync}})(2,\mathtt{upd}(\xi^{\circ}_{\mathsf{sync}})(1,1)) = \mathtt{pred}(\xi^{\circ}_{\mathsf{sync}})(2,2) = \mathsf{true}.$$

Then we have

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{sync}})(2,1) = \mathsf{false}.$$

Thus condition 1. in Definition 56 does not hold. Similarly, $\xi^{\mathtt{i}}_{\mathsf{sync}}$ and $\xi^{\circ}_{\mathsf{sync}}$ are not commutative. □

Proof (G_{assign} is asynchronous). Assume $\xi^{\mathtt{i}}_{\mathsf{ass}}$ and $\xi^{\circ}_{\mathsf{ass}}$ are the notations for obligations in G_{assign}.

$$\begin{array}{rcl} \xi^{\mathtt{i}}_{\mathsf{ass}} & \stackrel{\text{def}}{=} & B?\mathrm{req}(\varepsilon)\langle \mathsf{true}; \mathbf{t} := \mathbf{t}+1 \;\; \mathbf{c} := \mathbf{c} \cup \{\mathbf{t}\}\rangle \\ \xi^{\circ}_{\mathsf{ass}} & \stackrel{\text{def}}{=} & B!\mathrm{ans}(x:\;\mathsf{int})\langle x \in \mathbf{c}\;;\; \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle \end{array}$$

$$\begin{array}{rcl} \mathtt{pred}(\xi^{\mathtt{i}}_{\mathsf{ass}}) & \stackrel{\text{def}}{=} & \lambda\varepsilon,\mathbf{t},\mathbf{c}.(\mathsf{true}) \\ \mathtt{pred}(\xi^{\circ}_{\mathsf{ass}}) & \stackrel{\text{def}}{=} & \lambda x,\mathbf{t},\mathbf{c}.(x \in \mathbf{c}) \\ \mathtt{upd}(\xi^{\mathtt{i}}_{\mathsf{ass}}) & \stackrel{\text{def}}{=} & \lambda\varepsilon,\mathbf{t},\mathbf{c}.\langle\mathbf{t}+1 \;\; \mathbf{c}\cup\{\mathbf{t}\}\rangle \\ \mathtt{upd}(\xi^{\circ}_{\mathsf{ass}}) & \stackrel{\text{def}}{=} & \lambda x,\mathbf{t},\mathbf{c}.\langle\mathbf{c}\setminus\{x\}\rangle \end{array}$$

As for $\xi^{\circ}_{\mathsf{ass}}, \xi^{\circ}_{\mathsf{ass}}$, let $\mathbf{c} = w$, where w is a set. Assume

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{ass}})(v,w) = \mathsf{true}$$

and

$$\texttt{pred}(\xi^{\circ}_{\mathsf{ass}})(v', \texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v, w)) = \texttt{pred}(\xi^{\circ}_{\mathsf{ass}})(v', w \setminus \{v\}) = \mathsf{true}.$$

Then, $v' \neq v$, we have

$$\texttt{pred}(\xi^{\circ}_{\mathsf{ass}})(v', w) = \mathsf{true}$$

and

$$\texttt{pred}(\xi^{\circ}_{\mathsf{ass}})(v, \texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v', w)) = \texttt{pred}(\xi^{\circ}_{\mathsf{ass}})(v, w \setminus \{v'\}) = \mathsf{true}.$$

Moreover,

$$\texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v', \texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v, w)) = \texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v', w \setminus \{v\}) = w \setminus \{v\} \setminus \{v'\},$$

which is equal to

$$\texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v, \texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v', w)) = \texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v, w \setminus \{v'\})) = w \setminus \{v'\} \setminus \{v\}.$$

So that conditions 1. and 2. in Definition 56 both hold.

We can similarly check $\xi^{\mathtt{i}}_{\mathsf{ass}}, \xi^{\mathtt{i}}_{\mathsf{ass}}$. Let $\mathbf{t} = w_1$ and $\mathbf{c} = w_2$. w_1 is the current value of state $\mathbf{t}$, and w_2 is the current set of state $\mathbf{c}$. Condition 1. in Definition 56 holds immediately since the predicate is always true in $\xi^{\mathtt{i}}_{\mathsf{ass}}$. Condition 2. also holds immediately since the interaction variable is always ε.

As for $\xi^{\circ}_{\mathsf{ass}}, \xi^{\mathtt{i}}_{\mathsf{ass}}$, let $\mathbf{t} = w_1$ and $\mathbf{c} = w_2$. w_1 and w_2 are the current sets of states $\mathbf{t}$ and $\mathbf{c}$. Assume

$$\texttt{pred}(\xi^{\circ}_{\mathsf{ass}})(v, w_2) = \mathsf{true}$$

and

$$\texttt{pred}(\xi^{\mathtt{i}}_{\mathsf{ass}})(\varepsilon, \texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v, w_2)) = \texttt{pred}(\xi^{\mathtt{i}}_{\mathsf{ass}})(\varepsilon, w_2 \setminus \{v\}) = \mathsf{true}.$$

Then we have

$$\texttt{pred}(\xi^{\mathtt{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2) = \mathsf{true}$$

and

$$\texttt{pred}(\xi^{\circ}_{\mathsf{ass}})(v, \texttt{upd}(\xi^{\mathtt{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2)) = \texttt{pred}(\xi^{\circ}_{\mathsf{ass}})(v, w_2 \cup \{w_1 + 1\}) = \mathsf{true}.$$

Moreover,

$$\texttt{upd}(\xi^{\mathtt{i}}_{\mathsf{ass}})(\varepsilon, \texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v, w_2)) = \texttt{upd}(\xi^{\mathtt{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2 \setminus \{v\}) = w_1 + 1, w_2 \setminus \{v\} \cup \{w_1 + 1\},$$

which is equal to

$$\texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v, \texttt{upd}(\xi^{\mathtt{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2)) = \texttt{upd}(\xi^{\circ}_{\mathsf{ass}})(v, w_1 + 1 \;\; w_2 \cup \{w_1 + 1\})) = w_1 + 1, w_2 \cup \{w_1 + 1\} \setminus \{v\}.$$

However, $\xi^{\mathtt{i}}_{\mathsf{ass}}, \xi^{\circ}_{\mathsf{ass}}$ are not forward commutative. Assume

$$\texttt{pred}(\xi^{\mathtt{i}}_{\mathsf{ass}})(\varepsilon, w_1 \;\; w_2) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{ass}})(v,\mathtt{upd}(\xi^{\mathtt{i}}_{\mathsf{ass}})(\varepsilon,w_1\ \ w_2)) = \mathtt{pred}(\xi^{\circ}_{\mathsf{ass}})(v,w_1+1\ \ w_2\cup\{w_1+1\}) = \mathsf{true}.$$

However $\mathtt{pred}(\xi^{\circ}_{\mathsf{ass}})(v,w_2)$ needs not be equal to truth. So $\xi^{\circ}_{\mathsf{ass}},\xi^{\mathtt{i}}_{\mathsf{ass}}$ are forward commutative, but not commutative.

Then by Definition 58, G_{assign} is commutative, thus G_{assign} is asynchronous by Propositions 61 and 52. □

Proof (G_{async} is asynchronous). Assume $\xi^{\mathtt{i}}_{\mathsf{async}}$ and $\xi^{\circ}_{\mathsf{async}}$ are the notations for obligations in G_{async}.

$$\begin{aligned}\xi^{\mathtt{i}}_{\mathsf{async}} &\stackrel{\text{def}}{=} B?\mathrm{req}(\varepsilon)\langle\mathsf{true};\varepsilon\rangle\\ \xi^{\circ}_{\mathsf{async}} &\stackrel{\text{def}}{=} B!\mathrm{ans}(x:\ \mathsf{int})\langle x\notin\mathbf{c}\ ;\ \mathbf{c}:=\mathbf{c}\cup\{x\}\rangle\end{aligned}$$

$$\begin{aligned}\mathtt{pred}(\xi^{\mathtt{i}}_{\mathsf{async}}) &\stackrel{\text{def}}{=} \lambda\varepsilon,\mathbf{c}.(\mathsf{true})\\ \mathtt{pred}(\xi^{\circ}_{\mathsf{async}}) &\stackrel{\text{def}}{=} \lambda x,\mathbf{c}.(x\notin\mathbf{c})\\ \mathtt{upd}(\xi^{\mathtt{i}}_{\mathsf{async}}) &\stackrel{\text{def}}{=} \lambda\varepsilon,\mathbf{c}.\langle\varepsilon\rangle\\ \mathtt{upd}(\xi^{\circ}_{\mathsf{async}}) &\stackrel{\text{def}}{=} \lambda x,\mathbf{c}.\langle\mathbf{c}\cup\{x\}\rangle\end{aligned}$$

As for $\xi^{\circ}_{\mathsf{async}},\xi^{\circ}_{\mathsf{async}}$, let $\mathbf{c}=w$, where w is a set. Assume

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{async}})(v,w) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{async}})(v',\mathtt{upd}(\xi^{\circ}_{\mathsf{async}})(v,w)) = \mathtt{pred}(\xi^{\circ}_{\mathsf{async}})(v',w\cup\{v\}) = \mathsf{true}.$$

Then, since

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{async}})(v',w\cup\{v\}) = \mathsf{true}$$

implies $v'\neq v$ and $v'\notin w$, we have

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{async}})(v',w) = \mathsf{true}$$

and

$$\mathtt{pred}(\xi^{\circ}_{\mathsf{async}})(v,\mathtt{upd}(\xi^{\circ}_{\mathsf{async}})(v',w)) = \mathtt{pred}(\xi^{\circ}_{\mathsf{async}})(v,w\cup\{v'\}) = \mathsf{true}.$$

Moreover,

$$\mathtt{upd}(\xi^{\circ}_{\mathsf{async}})(v',\mathtt{upd}(\xi^{\circ}_{\mathsf{async}})(v,w)) = \mathtt{upd}(\xi^{\circ}_{\mathsf{async}})(v',w\cup\{v\}) = w\cup\{v\}\cup\{v'\},$$

which is equal to

$$\mathtt{upd}(\xi^{\circ}_{\mathsf{async}})(v,\mathtt{upd}(\xi^{\circ}_{\mathsf{async}})(v',w)) = \mathtt{upd}(\xi^{\circ}_{\mathsf{async}})(v,w\cup\{v'\})) = w\cup\{v'\}\cup\{v\}.$$

So that conditions 1. and 2. in Definition 56 both hold.
We can easily prove that $\xi^{\mathtt{i}}_{\mathsf{async}},\xi^{\mathtt{i}}_{\mathsf{async}}$, and $\xi^{\mathtt{i}}_{\mathsf{async}},\xi^{\circ}_{\mathsf{async}}$, and $\xi^{\circ}_{\mathsf{async}},\xi^{\mathtt{i}}_{\mathsf{async}}$ are semi-commutativities. By Definition 58, G_{assign} is commutative, thus G_{assign} is asynchronous by Propositions 61 and 52. □