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 purpose of this paper is to introduce a theory of specification for communicating processes under the condition that the observation is done asynchronously, motivated by a semantic problem in specifications for distributed systems.

The semantic problem arose in a concrete engineering setting, through our collaboration with the design and development of a large IT infrastructure for ocean sciences [17], which is a typical large-scale distributed system. In that infrastructure, applications are predominantly built as asynchronous interactions among distributed components. Since some of these components may be contributed by the third party so that they may be buggy or untrusted, we cannot completely rely on static verification. To detect undesirable behaviours during runtime is thus needed. We start from consider having system-level observers observe the endpoint behaviours, and wish to provide a basis for *dynamically* safe-behaviours enforcement. However, putting system-level observer at every endpoint is expensive and they might be polluted by the malicious endpoint. To concur this problem, an ideal setting comes to have *remotely* located observer (e.g., "outline monitor" [9]), who would be asynchronously inspecting behaviours of a component against a specification. For this endeavour, we need to formulate an expressive *specification language* usable for asynchronously monitoring components. We then came across a basic issue in the *semantics* of a specification language in the presence of asynchronous communication. The issue makes naturally written specifications *semantically nonsensical*, thus posing a fundamental challenge to our endeavour to provide a consistent specification-verification framework.

M. Koutny and I. Ulidowski (Eds.): CONCUR 2012, LNCS 7454, pp. 209–224, 2012.

The combination of asynchrony and *state* is omnipresent in specifications for distributed systems capturing real-life scenarios, where e.g. the (expected) states of participants in the applications, such as the credit of a client for online shopping, or the purchase number for a transaction, play a critical role. 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 remotely outside the observee, the order of actions that she observes may not necessarily be the same as the one happening at the observee. We 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.

Contributions. In the remainder, §2 illustrates the background, including the semantic issue in asynchronous specifications, through concrete examples. Starting from these motivating examples, the 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 [11]) including a decidable one admitting a rich set of specifications (§4).

Finally in §5, we examine the practical implications of the theory, discuss related work and conclude with further topics. For the space sake, the proofs of the technical results as well as further examples are left to the full version [5].

2 Motivating Examples

2.1 Using State(s) in Protocol Specifications

Before formally introducing the syntax and semantics of specifications, we discuss key ideas through simple examples. Our specification language is based on multiparty session types [3, 13] annotated by logical formulae, extending [4] with local state(s).

We first motivate the use of state in specifications, considering the scenario below:

(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.

In [4, 8, 13], this scenario can only be realised as a single protocol between Buyer and Seller; while, by using state(s), it can be realised using *two* protocols, one for each step. Separating protocols has a merit in flexibility: when Buyer and Seller finish step 1, both can terminate, and an invoice may be issued any time later. Below we present a *stateful* specification that realising using two separate protocols.

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

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

Above G_{pcs} and G_{ivc} denote *stateful protocols*, or SPs from now on for short, respectively corresponding to steps 1 and 2. Each specifies the flow of interactions which the participants, S (for seller) and B (for buyer), should realise at each session. $\langle ...; ...\rangle\langle ...; ...\rangle$ are the obligations for sender (the former) and receiver (the latter), where the block before ";" is the predicate and the one after is the state(s) updating rule. $\langle \mathsf{truth}; \varepsilon\rangle$ means no obligation. The syntax is formally introduced in §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* and sends its identity, then (in practice, after authentication etc.) S sends back a *fresh* purchase number *PNo*, 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*, the conversation terminates.

Note our specifications use local state to record an abstraction of preceding interactions across sessions, used for constraining future behaviours. Our ultimate aim is to specify visible behaviours: thus the stipulated state does not have to come from an actual state of a process: we may call it a "ghost state" following JML [1].

2.2 Synchrony and Asynchrony in 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 purchase number incrementing the previously issued one: so S issues e.g. 1, 2, 3, . . . in a sequence of sessions. Figure 2 (a) shows the corresponding protocol G_{sync} which the participants, S and B, should realise at each session. **c** is a local state of S, denoting the next purchase number.

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

(a) synchronous spec

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

(b) asynchronous spec

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

In the first line of G_{sync}, B requests S 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. $\langle \mathsf{truth}; \varepsilon\rangle$ 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, for a remote observer, even if S *actually* sends the series of purchase 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 practical assumption that the order of messages belonging to *distinct* sessions may not be preserved. In particular, 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 incongruent.

As a remedy, we present G_{async} in Example 2(b), which is intended for asynchronous observation. We now use the *set* of purchase numbers: $\mathbf{c}$, whose type is a set of integers, corresponds to **PLog** in Example 2.1. 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

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

Example 3 (A refinement of G_{async})

$$\begin{aligned} G_{\mathsf{ass}} = {} & B \to S : \mathrm{req}(\varepsilon)\langle \mathsf{truth}\ ;\ \varepsilon\rangle\langle \mathsf{truth}\ ;\ \mathbf{t} := \mathbf{t}+1, \mathbf{c} := \mathbf{c} \uplus \{\mathbf{t}\}\rangle. \\ & S \to B : \mathrm{ans}(x : \mathsf{int})\langle x \in \mathbf{c};\ \mathbf{c} := \mathbf{c} \setminus \{x\}\rangle\langle \mathsf{truth}\ ;\ \varepsilon\rangle. \\ & \mathsf{end} \end{aligned}$$

G_{ass} in Example 3 is a refinement of G_{async} in Example 2: while still being suitable for asynchronous observations, it 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 basic idea is that, if S receives n requests, then (assuming the server issues the purchase numbers starting from 1) as a whole the numbers which can be issued are among $\{1,2,..,n\}$. And if S issues a number from this set, the remaining numbers are what it can issue.

To understand G_{ass} 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_{ass} 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).

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 from G_{ass}

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 sure 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: the request-answer order in each session is preserved because without the request, its answer cannot occur. Unlike G_{async}, the observer can expect, based on G_{ass}, that the first answer is surely 1; and the second is surely 2. This example shows how we can represent causality while (intuitively) keeping the asynchronous nature of specifications.

3 Asynchronous Specifications

3.1 Syntax of Protocols and Specifications

Grammar of global and local stateful protocols. Figure 2 summarises the grammar of global 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 extends [4] with local states and operations on them: by adding simple state update, we obtain a rich class of stateful specifications.

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

$$\begin{array}{lllll} G ::= \mathsf{p} \to \mathsf{q} : \{l_i(x_i : S_i)\langle A_i;E_i\rangle\langle A'_i;E'_i\rangle.G_i\}_{i\in I} & \text{G-cm} & T ::= \mathsf{p}!\{l_i(x_i : S_i)\langle A_i;E_i\rangle.T_i\}_{i\in I} & \text{L-sel} \\ \quad \mid G_1 \mid G_2,\ role(G_1) \cap role(G_2) = \emptyset & \text{G-par} & \quad \mid \mathsf{p}?\{l_i(x_i : S_i)\langle A_i;E_i\rangle.T_i\}_{i\in I} & \text{L-bra} \\ \quad \mid \mathsf{end} & \text{G-end} & \quad \mid \mathsf{end} & \text{L-end} \end{array}$$

Fig. 2. The grammar of stateful protocols

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. We call $\langle A;E\rangle$ *obligation*. We use updates instead of post-conditions for usability in runtime verification. $(S,\ldots)$ are sorts (types of expressions), and $(e,\ldots)$ are expressions, where $op(e_1,...,e_n)$ is the operation op on parameters $e_1,...,e_n$. We use 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, $\mathbf{f} := e$ is assigned by e. The grammar of G and T is simplified for distilled presentation. In particular we omit recursion, which however can be added preserving all results, see §5.

In G, $\mathsf{p} \to \mathsf{q}$ describes the communication from sender p to receiver q, while $\mathsf{p}!$ and $\mathsf{p}?$ are endpoint actions for output (to p) and input (from p). In $l_i(x_i : S_i)$, l_i is the label for a branch: when l_i is chosen, the interaction variable is x_i, and S_i 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 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 to satisfy A' and as a result E' is done. In G-par, its side condition (where $role(G)$ denotes the set of roles in G) demands no role is shared by G_1 and G_2, 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 truth 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.

Well-formedness and projection. Assume $\mathsf{p} \to \mathsf{q} : \{l_i(x_i : S_i)\langle A_i;E_i\rangle\langle A'_i;E'_i\rangle.G_i\}_{i\in I}$ is inside a context, with possibly preceding interactions. The following well-formedness conditions, based on [4], stipulate consistency of global protocols:

(1) (a) $\forall i \in I, \mathit{field}(A'_i) = \emptyset$ (where $\mathit{field}(A)$ denotes the sets of field names occurring in A); 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 p, a sender, has sent or received before, as well as x_i. Similarly for A'_i and E'_i 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, i.e. at each step).

(1-a) says that a predicate of a receiver is stateless (generally, 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). (1-b) says that, in every interaction, the predicate at sender always imply the predicate at the receiver: together with (1-a), this means that if a sender sends a message that satisfies the sender's predicate, then automatically the receiver's predicate is satisfied (the latter however is useful for the receiver to know what it can expect). (2) and (3) are from [4]. All examples treated in this paper are easily well-formed. *Henceforth we assume all global SPs we treat are well-formed.*

A global protocol is useful to capture the overall interaction scenario, while a local protocol specifies what the endpoint is expected to do. They are linked by *endpoint projection*. Leaving its formal definition to [5], we illustrate the idea by an example.

Example 4 (endpoint projection). The local SPs projected from G_{ass} are:

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

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:

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

Above, $\mathtt{I}$ (resp. $\mathtt{O}$) is a mode denoting input (resp. output) capability. Γ, *shared environment*, describes the permitted behaviour at each shared channel; and the type of each field. When a process has $a : \mathtt{I}(G[\mathrm{p}])$, it can *accept* invitations via a shared channel a to play the role p following what (the p-projection of) G specifies; while $a : \mathtt{O}(G[\mathrm{p}])$ is its dual. In Δ, *session environment*, $s[\mathrm{p}] : T$ describes the session behaviour (T) in a session s as 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. In formulae, a field $\mathbf{f}$ itself represents its current value.

Example 5. Based on G_{ass} in Example 3 and its local SPs in Example 4, 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_{ass}, assuming there are two ongoing sessions s_1 and s_2.

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

$$T_B = S!\mathrm{req}(\varepsilon).S?\mathrm{ans}(x : \mathsf{int}).\mathsf{end}, \quad \Theta_{B_1} = \langle \Gamma'_{B_1}, \mathtt{b}_1 : \mathtt{O}(G_{\mathsf{ass}}[B]);\ \Delta'_{B_1},\ s_1[B] : T_B;\ D_{B_1}\rangle$$
$$\Theta_{B_2} = \langle \Gamma'_{B_2}, \mathtt{b}_2 : \mathtt{O}(G_{\mathsf{ass}}[B]);\ \Delta'_{B_2}, s_2[B] : T_B;\ D_{B_2}\rangle$$

The data storage in Θ_{ass} is $D'_{Ser}, \mathbf{t}, \mathbf{c}$. In this protocol, no state in D'_{Ser} 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 [Req-ini], [Req] and [Acc] defined in Figure 3), it shows that buyers B_1 and B_2 are the invitors requesting S to join session s_1 and s_2.

3.2 Semantics of Specifications

We present the semantics of specifications as a labelled transition system (LTS). The transition is of the form $\Theta \xrightarrow{\ell} \Theta'$, which intuitively means Θ as a specification *allows* a process to do an action ℓ, and the resulting process should conform to Θ'. 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 fresh. $a(s[\mathtt{p}]:G)$ 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). The LTS is defined in Figure 3 below: the induced transition is deterministic: if $\Theta \xrightarrow{\ell} \Theta'$ and $\Theta \xrightarrow{\ell} \Theta''$, then $\Theta' = \Theta''$.

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

[Req]
$$\frac{a:\mathtt{O}(G[\mathtt{p}_j]) \in \Gamma,\ \mathit{role}(G) = \{\mathtt{p}_i\}_{i\in I}}{\langle \Gamma; \Delta, s[\mathtt{p}_j] : G \upharpoonright \mathtt{p}_j; D\rangle \xrightarrow{\overline{a}\langle s[\mathtt{p}_j]:G\rangle} \langle \Gamma; \Delta; D\rangle}$$

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

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

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

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

Fig. 3. Labelled transition system for specifications

The first two rules are for invitations. [Req-ini] is used when s is fresh, i.e. when the first request happens to the sender to ask someone for playing role $\mathtt{p}_j$ in a fresh s. The round parenthesis in $\overline{a}(s[\mathtt{p}_j]:G)$ indicates s in this label 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, when s is *not* fresh in the session environment, and if Θ has an output channel a with G, (1) the target behaviour is permitted to send a request $\overline{a}\langle s[\mathtt{p}_j]:G\rangle$ to ask someone to play role $\mathtt{p}_j$ in session s; and (2) after requesting, we take off the capability at $\mathtt{p}_j$. Rule [Acc] says that, if s is a new session, and all states declared in $G \upharpoonright \mathtt{q}$, $\mathit{field}(G \upharpoonright \mathtt{q})$, are in D, when 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 in 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 j-th branch of T under Γ (note that, when v is a name, Γ needs to have the knowledge of its type, but it is not needed if v is a non-channel value, like 3 or ”*hello*” whose type is automatically known without Γ); and A_j after substitution holds under Γ. The condition $s \notin \mathsf{dom}(\Delta)$ says that, when an agent communicates in a session, it is playing only a single role (this restriction can be taken off but simplifies the technical development). In the conclusion, T'_j substitutes v for x_j and prepares for the next action, 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 $\mathbf{f} := \mathbf{f} \uplus \{x_j\}$, 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.

3.3 Processes and Satisfaction

Definition 6 (trace). A *trace* $(\mathtt{s}, \mathtt{s}', \ldots)$ is a sequence of actions where we assume a request/accept action introducing the session channel, say s, binds the later occurrences of s. Based on this binding, we only consider traces which satisfy the standard binding conventions, i.e. two binding occurrences never coincide and if free s occurs then it cannot do so before a binding occurrence (by an accept or request).

Below $\mathsf{sbj}(\ell)$ denotes the *subject* of ℓ, given as, for a request/accept, the initial shared channel (e.g. $\mathsf{sbj}(\overline{a}\langle s[\mathtt{p}_j] : G\rangle) = a$); and, for a session action, the session channel with the interacting role (e.g. $\mathsf{sbj}(s[\mathtt{p}, \mathtt{q}]!l_j(v)) = s[\mathtt{q}]$, $\mathsf{sbj}(s[\mathtt{p}, \mathtt{q}]?l_j(v)) = s[\mathtt{p}]$).

Definition 7 (legal unit permutation). Let $\ell_1 \cdot \ell_2$ be a trace. Then a permutation from $\ell_1 \cdot \ell_2$ to $\ell_2 \cdot \ell_1$ is *legal* if one of the following conditions holds:

1. ℓ_1 and ℓ_2 are both inputs and either both are session actions and $\mathsf{sbj}(\ell_1) \neq \mathsf{sbj}(\ell_2)$ to the same receiver, or one of them is an accept action and ℓ_1 does not bind ℓ_2.
2. ℓ_1 and ℓ_2 are both outputs and either both are session actions and $\mathsf{sbj}(\ell_1) \neq \mathsf{sbj}(\ell_2)$ to the same sender, or one of them is a request action and ℓ_1 does not bind ℓ_2.
3. ℓ_1 is an output and ℓ_2 is an input and ℓ_1 does not bind ℓ_2.

Such a permutation is called a *legal unit permutation*. 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*.

Example 8 (legal permutation). 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.

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 9 (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 10 (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.

Definition 11 ($|\Theta|$: valid traces of Θ). We define $|\Theta|$, the set of *valid traces* of Θ, as finite sequences from the LTS of Θ defined in Figure 3.

Intuitively, a valid trace is a trace that Θ approves. The following 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.

Definition 12 (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) $\mathsf{Obs}_s(P) \subset |\Theta|$.

2.a (input consistency) Whenever $\mathtt{s} \in \mathsf{Obs}_s(P)$ and $\mathtt{s} \cdot \ell \in |\Theta|$ where ℓ is an input, $\mathtt{s} \cdot \ell' \in \mathsf{Obs}_s(P)$ and ℓ' is an input with the same subject 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, as well as:

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

Note that a synchronous process (2.a) can accept a valid input only when it is ready to receive it; while an asynchronous process (2.b) can, and should, accept any valid input.

Example 13 (valid/invalid traces of G_{ass}). We consider Θ_{ass} from Example 5 which uses the local SP from G_{ass} in Example 3 for the server side. Then, for example, the trace $s_2[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[S,B]!\mathrm{ans}(1) \cdot s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_1[S,B]!\mathrm{ans}(2)$ is valid for Θ_{ass}, but $s_2[B,S]?\mathrm{req}(\varepsilon) \cdot s_2[S,B]!\mathrm{ans}(2) \cdot s_1[B,S]?\mathrm{req}(\varepsilon) \cdot s_1[B,S]!\mathrm{ans}(1)$ is not its trace (violation is at the second step), i.e. it is not permitted by Θ_{ass}.

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 a process behaviour. 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.

Definition 14 (asynchronously verifiable specification). We say Θ is *asynchronously verifiable* or simply *asynchronous* when $\mathtt{s} \in |\Theta|$ and $\mathtt{s} \curvearrowright \mathtt{s}'$ imply $\mathtt{s}' \in |\Theta|$.

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

Example 15. Let T_{sync} be the local SP at server, projected from G_{sync}. Then $\Theta_{\mathsf{sync}} = \langle \Gamma'_{Ser}, \mathtt{ser} : \mathtt{I}(G_{\mathsf{sync}}[S])\ ;\ \Delta'_{Ser}, s[S] : T_{\mathsf{sync}}\ ;\ D'_{Ser}, \mathbf{c}\rangle$, where I contains the sessions using G_{sync}, is *not* asynchronous by the traces given in §2.

On the other hand, checking asynchrony by Definition 14 means we should verify the property for all traces, which are usually infinitely many. Later we shall find methods by which we can validate the asynchrony of, for example, Θ_{ass} and all the corresponding specifications that use $G_{\mathsf{pcs}}/G_{\mathsf{ivc}}$ and G_{async}.

The following characterisation says that, if a specification Θ is asynchronous, the anomaly we discussed in §2.2, for G_{sync} in Figure 2(a), can never take place: 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 16. *Θ is asynchronous iff, for each P, $P \models_{\mathtt{sync}} \Theta$ implies $P \models_{\mathtt{async}} \Theta$.*

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 [14], we have shown how $\approx_{\mathtt{async}}$ (but not its synchronous counterpart) can be used for non-trivial optimising transformation.

Proposition 17. *If $P \approx_{\mathtt{async}} Q$ and $P \models_{\mathtt{async}} \Theta$ then $Q \models_{\mathtt{async}} \Theta$.*

4.2 Asynchrony in Specifications through Commutativity

A basic issue in Definition 14 and its characterisation in Proposition 16 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 [11], we may call a criterion 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 criterion.

Definition 18 (confluence). Θ is *confluent* if, whenever $\Theta \xrightarrow{\mathtt{s}} \Theta'$, if $\Theta' \xrightarrow{\ell_1 \cdot \ell_2} \Theta''$ and $\ell_2 \cdot \ell_1 \curvearrowright \ell_1 \cdot \ell_2$, then $\Theta' \xrightarrow{\ell_2 \cdot \ell_1} \Theta''$ 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 19. *Θ is asynchronous iff $\mathtt{s} \cdot \ell_1 \cdot \ell_2 \in |\Theta|$ and $\ell_1 \cdot \ell_2 \curvearrowright \ell_2 \cdot \ell_1$ imply $\mathtt{s} \cdot \ell_2 \cdot \ell_1 \in |\Theta|$ for each $\mathtt{s}$, ℓ_1 and ℓ_2.*

Proposition 20. *If Θ is confluent then it is asynchronous.*

Note that the other way round is not true. Given Θ is asynchronous, for any $\mathtt{s}, \ell_1$, and ℓ_2, $\mathtt{s} \cdot \ell_1 \cdot \ell_2 \in |\Theta|$ implies $\mathtt{s} \cdot \ell_2 \cdot \ell_1 \in |\Theta|$. However, it is possible that $\Theta \xrightarrow{\mathtt{s}} \Theta' \xrightarrow{\ell_1 \cdot \ell_2} \Theta''$ while $\Theta \xrightarrow{\mathtt{s}} \Theta' \xrightarrow{\ell_2 \cdot \ell_1} \Theta'''$, where $\Theta'' \neq \Theta'''$.

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 21 (predicate/update functions). Let $\xi \overset{\text{def}}{=} \mathtt{r} \dagger l(x:S)\langle A;E\rangle$ 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:

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

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

Example 22. Below we project G_{sync} and G_{ass} (all from §2) to the server. For simplicity we assume its local state only consists of those fields specified in global SP.

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

$$G_{\mathsf{ass}} \upharpoonright S = B?\text{req}(\varepsilon)\langle \mathsf{truth};\mathbf{t}:=\mathbf{t}+1\ \ \mathbf{c}:=\mathbf{c}\uplus\{\mathbf{t}\}\rangle \ .\ B!\text{ans}(x:\ \mathsf{int})\langle x\in\mathbf{c}\ ;\ \mathbf{c}:=\mathbf{c}\setminus\{x\}\rangle$$

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

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

Once we can treat obligations as operations on state(s), we can define their commutativity. Since the commutativity we need is asymmetric (corresponding to asymmetric permutations induced by asynchrony, cf. Definition 7), 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 23 (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 the value of 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*.

Example 24. We show ξ_1 in Example 22 does not commute over itself (i.e. ξ_1, ξ_1 is not commutative). Let $\mathbf{f}=\mathbf{c}$. We know $\mathtt{pred}(\xi_1)(1,1)$, $\mathtt{pred}(\xi_1)(2,\mathtt{upd}(\xi_1)(1,1))$ and $\mathtt{pred}(\xi_1)(2,2)$ are all truth, 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 SP in Θ or in the projection of a global SP in 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 SP carried by an input shared channel type, only the specified role is potentially local).

Definition 25 (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'.

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. We can easily show:

Proposition 26. *If Θ is commutative then it is confluent (hence asynchronous).*

Note that the other way round is not true: Θ is confluent does not imply that it is commutative. Since, based on Definition 18, Θ is confluent, then whenever $\Theta \xrightarrow{\mathsf{s}} \Theta'$, $\Theta' \xrightarrow{\ell_1 \cdot \ell_2} \Theta''$ and $\ell_1 \cdot \ell_2 \curvearrowright \ell_2 \cdot \ell_1$ imply $\Theta' \xrightarrow{\ell_2 \cdot \ell_1} \Theta''$. Θ' is commutative, but Θ' cannot imply that Θ is commutative.

This method can be strengthened by adding an invariant (including correlation among states) in state and checking that invariant continues to hold at each step. We can now show all our example specifications except the one induced by G_{sync} is asynchronous. Below we let Θ_{async}'s shared environment contains $a : I(G_{\mathsf{async}}[S])$, and let Θ_{async}'s data storage contains $\mathbf{c} \mapsto \{\}$. By inspecting the (semi-)commutativity of induced predicates and operations, we easily obtain:

Proposition 27. *Θ_{async} and Θ_{ass} at server are both commutative, hence asynchronous.*

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

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, by [20]:

Proposition 28. *With the SP language given in §3 restricting operations on integers to be the addition and the subtraction, then the commutativity is decidable.*

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:

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

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_{ass} 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 (for a concrete analysis, see the Appendix in our full version [5]). 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 [2,6,10,12]. The permutations associated with asynchronous communication used in Definition 7 are noted in these works (and implicit in such work as [15]). Their more explicit presentation in the categorical setting is found in [19]. There is also a study in component validation based on asynchronous histories such as [18]. In spite of these precursors and close technical connection, the existing works (except [16] 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 23) 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 27 (the proofs are in our full version [5]), we crucially use this semi-commutativity when verifying G_{ass} 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 [17] 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 [17] and EPSRC grants EP/F002114/1 and EP/G015481/1.

References

1. The Java Modeling Language (JML) homepage, `http://www.jmlspecs.org/`
2. Amadio, R., Castellani, I., Sangiorgi, D.: On Bisimulations for the Asynchronous π-Calculus. In: Montanari, U., Sassone, V. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 147–162. Springer, Heidelberg (1996)
3. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global Progress in Dynamically Interleaved Multiparty Sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
4. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A Theory of Design-by-Contract for Distributed Multiparty Interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010)
5. Chen, T.-C., Honda, K.: Full Version of this paper, to appear as an EECS technical report, Queen Mary. University of London
6. de Boer, F.S., Kok, J.N., Palamidessi, C., Rutten, J.J.M.M.: The Failure of Failures in a Paradigm for Asynchronous Communication. In: Groote, J.F., Baeten, J.C.M. (eds.) CONCUR 1991. LNCS, vol. 527, pp. 111–126. Springer, Heidelberg (1991)
7. Dennis, G., Seater, R., Rayside, D., Jackson, D.: Automating commutativity analysis at the design level. In: ISSTA 2004, pp. 165–174. ACM, New York (2004)
8. Chen, T.-C., Bocchi, L., Deniélou, P.-M., Honda, K., Yoshida, N.: Asynchronous Distributed Monitoring for Multiparty Session Enforcement. In: Bruni, R., Sassone, V. (eds.) TGC 2011. LNCS, vol. 7173, pp. 25–45. Springer, Heidelberg (2012)
9. Falcone, Y.: You Should Better Enforce Than Verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 89–105. Springer, Heidelberg (2010)
10. He, J., Josephs, M., Hoare, T.: A theory of synchrony and asynchrony. In: Programming Concepts and Methods. IFIP, pp. 459–478 (1990)
11. Hoare, C., Jifeng, H.: Unifying theories of programming. Prentice Hall series in computer science. Prentice Hall (1998)
12. Honda, K., Tokoro, M.: An Object Calculus for Asynchronous Communication. In: America, P. (ed.) ECOOP 1991. LNCS, vol. 512, pp. 133–147. Springer, Heidelberg (1991)
13. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM (2008)
14. Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K.: Type-Safe Eventful Sessions in Java. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 329–353. Springer, Heidelberg (2010)
15. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–564 (1978)

16. A multiparty multi-session logic, http://www.cs.le.ac.uk/people/lb148/StatefulAssertions/main-long.pdf
17. Ocean Observatories Initiative (OOI), http://www.oceanleadership.org/programs-and-partnerships/ocean-observing/ooi/
18. Owe, O., Steffen, M., Torjusen, A.B.: Model Testing Asynchronously Communicating Objects using Modulo AC Rewriting. ENCS 264(3), 69–84 (2010)
19. Selinger, P.: First-Order Axioms for Asynchrony. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 376–390. Springer, Heidelberg (1997)
20. Zarba, C.G.: Combining Sets with Integers. In: Armando, A. (ed.) FroCoS 2002. LNCS (LNAI), vol. 2309, pp. 103–116. Springer, Heidelberg (2002)