A Theory of Design-by-Contract for Distributed Multiparty Interactions[⋆]

Laura Bocchi[1], Kohei Honda[2], Emilio Tuosto[1], and Nobuko Yoshida[3]

[1] University of Leicester [2] Queen Mary University of London [3] Imperial College London

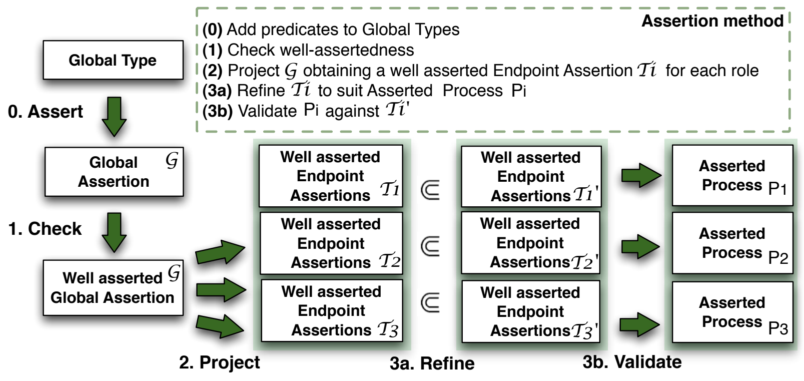
Abstract. Design by Contract (DbC) promotes reliable software development through elaboration of type signatures for sequential programs with logical predicates. This paper presents an assertion method, based on the π-calculus with full recursion, which generalises the notion of DbC to multiparty distributed interactions to enable effective specification and verification of distributed multiparty protocols. Centring on *global assertions* and their *projections* onto endpoint assertions, our method allows clear specifications for typed sessions, constraining the *content* of the exchanged messages, the *choice* of sub-conversations to follow, and *invariants* on recursions. The paper presents key theoretical foundations of this framework, including a sound and relatively complete compositional proof system for verifying processes against assertions.

1 Introduction

This paper introduces an assertion method for specifying and verifying distributed multiparty interactions, drawing ideas from a framework known as Design-by-Contract (DbC), which is widely used in practice for sequential computation [13, 18]. DbC [25] specifies a contract between a user and a sequential program by elaborating the type signature of the program with pre/post-conditions and invariants. Instead of saying "the method fooBar should be invoked with a string and an integer: then it will return (if ever) another string", DbC allows us to say "the method fooBar should be invoked with a string representing a date d between 2007 and 2008 and an integer n less than 1000 then it will (if ever) return a string representing the date n days after d". A type signature describes a basic shape of how a program can interact with other programs, stipulating its key *interface* to other components, which may be developed by other programmers. By associating signatures with logical predicates, DbC enables a highly effective framework for specifying, validating and managing systems' behaviour, usable throughout all phases of software development [21, 23, 28]. As a modelling and programming practice, DbC encourages engineers to make contracts among software modules precise [14, 25], and build a system on the basis of these contracts.

The traditional DbC-based approaches are however limited to type signature of sequential procedures. A typical distributed application uses interaction scenarios that are much more complex than, say, request-reply. To build a theory that extends the core

⋆ This work is partially supported by EPSRC EP/F003757, EP/F002114, EP/G015635 and EP/G015481, Leverhulme Trust Award "Tracing Networks".



Assertion method
(0) Add predicates to Global Types
(1) Check well-assertedness
(2) Project $\mathcal{G}$ obtaining a well asserted Endpoint Assertion $\mathcal{T}i$ for each role
(3a) Refine $\mathcal{T}i$ to suit Asserted Process Pi
(3b) Validate Pi against $\mathcal{T}i'$
Global Type
0. Assert
Global Assertion $\mathcal{G}$
1. Check
Well asserted Global Assertion $\mathcal{G}$
Well asserted Endpoint Assertions $\mathcal{T}1$
Well asserted Endpoint Assertions $\mathcal{T}2$
Well asserted Endpoint Assertions $\mathcal{T}3$
$\Subset$
Well asserted Endpoint Assertions $\mathcal{T}1'$
Well asserted Endpoint Assertions $\mathcal{T}2'$
Well asserted Endpoint Assertions $\mathcal{T}3'$
Asserted Process P1
Asserted Process P2
Asserted Process P3
2. Project
3a. Refine
3b. Validate

Fig. 1. The assertion method

idea of DbC to distributed applications, we consider a generalised notion of type signature for distributed interactions centring on abstraction units, called *sessions*. A session consists of a structured series of message exchanges among multiple participants. Each session follows a stipulated protocol, given as a type called *session type* [3, 19, 20], which prescribes a conversation scenario among its participants. Two or more sessions can interleave in a single endpoint. For example, a session for an electronic commerce will run interleaved with one for a financial transaction for payment. The communications in a distributed application are articulated as a collection of such sessions.

On this basis, we introduce a theory of assertions for distributed interactions centring on *global assertions*. A global assertion specifies a contract for participants in a multiparty session by elaborating a session type with logical predicates. A session type only specifies a skeletal protocol: it does not, for example, refer to constraints on message values except their types. Just as in the traditional DbC, the use of logical predicates allows us to specify more refined protocols, regarding, among others, *content* of messages, how *choice* of sub-conversations is made based on preceding interactions, and what *invariants* may be obeyed in recursive interactions. The key ideas are presented in Figure 1, which we illustrate below.

(0,1) A specification for a multiparty session is given as a *global assertion* $\mathcal{G}$, namely a protocol structure annotated with logical predicates. A minimal semantic criterion, *well-assertedness of* $\mathcal{G}$ (§ 3.1), characterises consistent specifications with respect to the temporal flow of events, to avoid unsatisfiable specifications.

(2) $\mathcal{G}$ is projected onto endpoints, yielding one *endpoint assertion* ($\mathcal{T}_i$) for each participant, specifying the behavioural responsibility of that endpoint (§ 4). The consistency of endpoint assertions are automatically guaranteed once the original global assertion is checked to be well-asserted.

(3) Asserted processes, modelled with the π-calculus[1] annotated with predicates (§ 5.1), are verified against endpoint assertions (§ 5.2) through a sound and relatively com-

[1] For the sake of a simpler presentation, the present paper does not treat name passing in full generality, except for private channel passing in session initiation. The theory however can incorpoate these elements, as explained in Section 7.

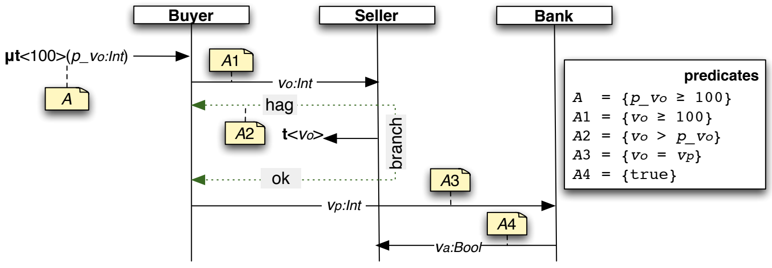


Fig. 2. Global assertion for the protocol.

plete compositional proof system (§ 6). Completeness, proved through generation of principal formulae, yields a relative decision procedure for satisfiability.

Our contributions include an algorithmic validation of consistency of global assertions (Prop. 3.2 and 4.3); semantic foundations of global assertions through labelled transitions (Prop. 6.4 and 6.3); a compositional proof system for validating processes against assertions (Theorem 6.5), leading to *predicate error freedom* (Theorem 6.6) which ensures that the process will meet its obligations assuming that the remaining parties do so. Theorem 6.7 is completeness. § 7 concludes with further results and related work.

2 DbC for Distributed Multiparty Interactions

The theory we present centres on the notion of *global assertion*. A global assertion uses logical formulae to prescribe, for each interaction specified in the underlying session type, what the sending party must guarantee, and dually what the receiving party can rely on. Concretely:

1. Each message exchange in a session is associated with a predicate which constrains the values carried in the message (e.g., "the quantity on the invoice from seller to buyer equals the quantity on the order");
2. Each branch in a session is associated with a predicate which constrains the selection of that branch (e.g., "seller chooses the 'sell' option for a product if the ordered quantity does not exceed the stock");
3. Each recursion in a session is associated with an invariant representing an obligation to be maintained by all parties at each repetition of the recursion (e.g., "while negotiating, seller and buyer maintain the price per unit about a fixed threshold").

As an illustration, Figure 2 describes a simple multiparty session among the participants `Buyer`, `Seller`, and `Bank` exchanging messages whose content is represented by the *interaction variables* v_o, v_p (of type Int) and v_a (of type Bool). `Buyer` asynchronously sends an offer v_o, then `Seller` selects either to recursively start negotiating (hag) or to accept the offer (ok). In the latter case, `Buyer` instructs `Bank` to make a payment v_p. Finally, `Bank` sends `Seller` an acknowledgement v_a. The recursion parameter p_v_o is initially set to 100 and, upon recursive invocation, it takes the value that v_o had in the previous recursive invocation. This allows us to compare the current content of v_o with the one of the previous recursion instance (cf. $A2$ below). In Figure 2, the recursion invariant A states that p_v_o is always greater or equal than 100; `Buyer` guarantees $A1$ which, dually, `Seller` relies upon; by $A2$, `Buyer` has to increase the price during negotiations until an agreement is reached; the value of the (last) offer and the payment must be equal by $A3$, while $A4$ does not constrain v_a.

3 Global Assertions

We use the syntax of logical formulae, often called *predicates*, as follows.

$$A,B \ ::= \ e_1 = e_2 \ \mid \ e_1 > e_2 \ \mid \ \phi(e_1,\ldots,e_n) \ \mid \ A \wedge B \ \mid \ \neg A \ \mid \ \exists v(A)$$

where e_i ranges over expressions (which do not include channels) and ϕ over predefined atomic predicates with fixed arities and types [24, §2.8]. We denotes the set of *free variables* of A with $var(A)$, similarly for $var(e)$. We fix a model of the predicates, called *underlying logic*, for which we assume the validity of closed formulae to be decidable.

Global assertions (ranged over by $\mathcal{G}, \mathcal{G}', \ldots$) elaborate global session types in [20] with logical formulae. The syntax is given below:

$$\begin{array}{rll} \mathcal{G} \ ::= & \mathtt{p} \to \mathtt{p}' : k\,(\tilde{v} : \tilde{S})\{A\}.\mathcal{G} & \mid \ \mathbf{t}\langle\tilde{e}\rangle \\ \mid & \mathtt{p} \to \mathtt{p}' : k\,\{\{A_j\}l_j : \mathcal{G}_j\}_{j\in J} & \mid \ \mathcal{G}, \mathcal{G}' \\ \mid & \mu\mathbf{t}\langle\tilde{e}\rangle(\tilde{v} : \tilde{S})\{A\}.\mathcal{G} & \mid \ \mathsf{end} \end{array}$$

- $\mathtt{p}, \mathtt{p}', ..$ are *participants*,
- $k, k', ..$ are *channels*,
- $u, v, ..$ are *interaction variables*,
- $S, S', ..$ are *sorts*.

Interaction $\mathtt{p} \to \mathtt{p}' : k\,(\tilde{v} : \tilde{S})\{A\}.\mathcal{G}$ describes a communication between a sender $\mathtt{p}$ and a receiver $\mathtt{p}'$ via the k^{th} session channel (k is a natural number), followed by $\mathcal{G}$. The variables in the vector $\tilde{v}$ are called *interaction variables* and bind their occurrences in A and $\mathcal{G}$; interaction variables are sorted by *sorts* S (Bool, Int, ...) that denote types for first-order message values. The predicate A constrains the content of $\tilde{v}$: the sender $\mathtt{p}$ *guarantees* A and the receiver $\mathtt{p}'$ *relies* on A (like in the rely-guarantee paradigm [22]).

Branching $\mathtt{p} \to \mathtt{p}' : k\,\{\{A_j\}l_j : \mathcal{G}_j\}_{j\in J}$ allows the selector $\mathtt{p}$ to send to participant $\mathtt{p}'$, through k, a label l_i from $\{l_j\}_{j\in J}$ (J is a finite set of indexes) if $\mathtt{p}$ guarantees A_i (upon which $\mathtt{p}'$ can rely). Once l_i is selected, $\mathcal{G}_i$ is to be executed by all parties.

Recursive assertion $\mu\mathbf{t}\langle\tilde{e}\rangle(\tilde{v} : \tilde{S})\{A\}.\mathcal{G}$ (cf. [11], $\mathbf{t}$ is an *assertion variable*) specifies how a recursive session, which may be repeated arbitrarily many times, should be carried out through interactions among participants. The formal parameters $\tilde{v}$ are a vector of pairwise distinct variables sorted by a vector of sorts $\tilde{S}$ of the same length (each v_i in $\tilde{v}$ has sort S_i of $\tilde{S}$); $\tilde{v}$ binds their free occurrences in A. The *initialisation vector* $\tilde{e}$ denotes the initial values for the recursion, each e_i instantiating v_i in $\tilde{v}$. The *recursion invariant* A specifies the condition that needs be obeyed at each recursion instantiation; *recursion instantiation*, of the form $\mathbf{t}\langle\tilde{e}\rangle$, is to be guarded by prefixes, i.e. the underlying recursive types should be contractive. A recursive assertion can be unfolded to an infinite tree, as in the *equi-recursive* view on recursive types [30].

Composition $\mathcal{G}, \mathcal{G}'$ represents the parallel interactions specified by $\mathcal{G}$ and $\mathcal{G}'$, while end represents the termination. Sorts and trailing occurrences of end are often omitted.

We write $\mathtt{p} \in \mathcal{G}$ when $\mathtt{p}$ occurs in $\mathcal{G}$. For the sake of simplicity we avoid linearity-check [3] by assuming that each channel in $\mathcal{G}$ is used (maybe repeatedly) only between two parties: one party for input/branching and by the other for output/selection.

Example 3.1 (Global Assertions). The protocol described in § 2 is modelled by

$$\begin{aligned}\mathcal{G}_{neg} &= \mu\mathbf{t}\langle 100\rangle(p\_v_o : \mathsf{Int})\{A\}.\ \mathtt{Buyer} \to \mathtt{Seller}\colon k_1\,(v_o : \mathsf{Int})\{A1\}.\\ &\qquad\qquad\qquad \mathtt{Seller} \to \mathtt{Buyer}\colon k_2\{\{A2\}\mathbf{hag}\colon \mathbf{t}\langle v_o\rangle, \{\mathsf{true}\}\mathbf{ok}\colon \mathcal{G}_{ok}\}\\ \mathcal{G}_{ok} &= \mathtt{Buyer} \to \mathtt{Bank}\colon k_3\,(v_p : \mathsf{Int})\{A3\}.\ \mathtt{Bank} \to \mathtt{Seller}\colon k_4\,(v_a : \mathsf{Bool})\{A4\}.\ \mathsf{end}\end{aligned}$$

where k_1, k_2, k_3, and k_4 are channels and the recursion parameter p_v_o (initially set to 100) denotes the offer of Buyer in the previous recursion instance.

3.1 Well Asserted Global Assertions

When setting up global assertions as a contract among multiple participants, we should prevent inconsistent specifications, such as those in which it is logically impossible for a participant to meet the specified obligations. Below we give two constraints on predicates of global assertions that guarantee consistency.

Let $I(\mathcal{G})$ be the set of variables occurring in $\mathcal{G}$; a participant p *knows* $v \in I(\mathcal{G})$ if v occurs in an interaction of $\mathcal{G}$ involving p (this relation can be computed effectively, see [31]). $I(\mathcal{G}) \upharpoonright \mathtt{p}$ denotes the set of variables of $\mathcal{G}$ that $\mathtt{p} \in \mathcal{G}$ knows.

History-sensitivity A predicate guaranteed by a participant p can only contain those interaction variables that p knows.

Temporal-satisfiability For each possible set of values satisfying A and, for each predicate A' appearing after A, it is possible to find values satisfying A'.

Consider the following examples:

$$\mathtt{p_A} \to \mathtt{p_B}\colon k_1(v : \mathsf{Int})\{\mathsf{true}\}.\ \mathtt{p_B} \to \mathtt{p_C}\colon k_2\,(v' : \mathsf{Int})\{\mathsf{true}\}.\ \mathtt{p_C} \to \mathtt{p_A}\colon k_3\,(z : \mathsf{Int})\{z > v\}.\ \mathsf{end}$$
$$\mathtt{p_A} \to \mathtt{p_B}\colon k_1(v : \mathsf{Int})\{v < 10\}\ \mathtt{p_B} \to \mathtt{p_A}\colon k_2\,(z : \mathsf{Int})\{v > z \wedge z > 6\}.\ \mathsf{end}.$$

The first global assertion violates history-sensitivity since $\mathtt{p_C}$ has to send z such that $z > v$ without knowing v. The second global assertion violates temporal-satisfiability because if $\mathtt{p_A}$ sends $v = 6$, which satisfies $v < 10$, then $\mathtt{p_B}$ will not be able to find a value that satisfies $6 > z \wedge z > 6$.

Assertions satisfying history-sensitivity and temporal-satisfiability are called *well-asserted assertions*. For the formal definitions, including inductive rules to check well-assertedness, see [31].

Proposition 3.2 (Well-assertedness). *Checking well-assertedness of a given global assertion is decidable if the underlying logic is decidable.*

4 Endpoint Assertions and Projection

Endpoint assertions, ranged over by $\mathcal{T}, \mathcal{T}', ..$, specify the behavioural contract of a session from the perspective of a single participant. The grammar is given as follows.

$$\begin{array}{rcl}\mathcal{T} & ::= & k!(\tilde{v}\colon \tilde{S})\{A\};\mathcal{T} \ \mid\ \mu\mathbf{t}\langle\tilde{e}\rangle(\tilde{v}\colon\tilde{S})\{A\}.\mathcal{T} \ \mid\ k\&\{\{A_i\}l_i\colon \mathcal{T}_i\}_{i\in I}\\ & \mid & k?(\tilde{v}\colon \tilde{S})\{A\};\mathcal{T} \ \mid\ \mathbf{t}\langle\tilde{e}\rangle \qquad\qquad\qquad \mid\ k\oplus\{\{A_j\}l_j\colon \mathcal{T}_j\}_{j\in I} \ \mid\ \mathsf{end}\end{array}$$

In $k!(\tilde{v}\colon \tilde{S})\{A\};\mathcal{T}$, the sender guarantees that the values sent via k (denoted by $\tilde{S}$-sorted variables $\tilde{v}$) satisfy A, then behaves as $\mathcal{T}$; dually for the receiver $k?(\tilde{v}\colon \tilde{S})\{A\};\mathcal{T}$.

In $k \oplus \{\{A_j\}l_j : \mathcal{T}_j\}_{j\in I}$ the selector guarantees A_j when choosing l_j on k; dually $k\&\{\{A_j\}l_j : \mathcal{T}_j\}_{i\in I}$ states that A_j can be assumed when branching at k on a label l_j. Assertion $\mu\mathbf{t}\langle\tilde{e}\rangle(\tilde{v} : \tilde{S})\{A\}.\mathcal{T}$ constrains parameters $\tilde{v}$ of type $\tilde{S}$ which initially take values $\tilde{e}$; the invariant of the recursion is A.

The projection of predicate A on participant $\mathtt{p}$, written $A \upharpoonright \mathtt{p}$, existentially quantifies all the variables of A that $\mathtt{p}$ does not know, and is defined as $\exists V_{ext}(A)$ where $V_{ext} = var(A) \backslash I(\mathcal{G}) \upharpoonright \mathtt{p}$. Also, $\tilde{e} \upharpoonright \mathtt{p}$ are the expressions in $\tilde{e}$ including only such that $var(e_i) \subseteq I(\mathcal{G}) \upharpoonright \mathtt{p}$. The *projection* function in Definition 4.1 maps global assertions, predicates and participants to endpoint assertions.

Definition 4.1 (Projection). Given $\mathcal{G}$ and A, the *projection of $\mathcal{G}$ for a participant* $\mathtt{p}$ *wrt* A is denoted by $(\mathcal{G}) \downarrow^{A}_{\mathtt{p}}$ and, assuming $\mathtt{p}_1 \neq \mathtt{p}_2$, recursively defined as follows.

$$(1)\quad (\mathtt{p}_1 \to \mathtt{p}_2 : k\,(\tilde{v} : \tilde{S})\{A\}.\mathcal{G}') \downarrow^{A_P}_{\mathtt{p}} = \begin{cases} k!(\tilde{v} : \tilde{S})\{A\}.(\mathcal{G}') \downarrow^{A \wedge A_P}_{\mathtt{p}} & \text{if } \mathtt{p} = \mathtt{p}_1 \\ k?(\tilde{v} : \tilde{S})\{(A \wedge A_P) \upharpoonright \mathtt{p}\}.(\mathcal{G}') \downarrow^{A \wedge A_P}_{\mathtt{p}} & \text{if } \mathtt{p} = \mathtt{p}_2 \\ (\mathcal{G}') \downarrow^{A \wedge A_P}_{\mathtt{p}} & \text{otw} \end{cases}$$

$$(2)\quad (\mathtt{p}_1 \to \mathtt{p}_2 : k\,\{\{A_i\}l_i : \mathcal{G}_i\}_{i\in I}) \downarrow^{A_P}_{\mathtt{p}} = \begin{cases} k \oplus \{\{A_i\}l_i : (\mathcal{G}_i) \downarrow^{A_i \wedge A_P}_{\mathtt{p}}\}_{i\in I} & \text{if } \mathtt{p} = \mathtt{p}_1 \\ k\&\{\{(A_i \wedge A_P) \upharpoonright \mathtt{p}\}l_i : (\mathcal{G}_i) \downarrow^{A_i \wedge A_P}_{\mathtt{p}}\}_{i\in I} & \text{if } \mathtt{p} = \mathtt{p}_2 \\ (\mathcal{G}_1) \downarrow^{A_P \wedge \bigvee_{j\in I} A_j}_{\mathtt{p}} \ (= (\mathcal{G}_i) \downarrow^{A_P \wedge \bigvee_{j\in I} A_j}_{\mathtt{p}}) & \text{otw} \end{cases}$$

$$(3)\quad (\mathcal{G}_1, \mathcal{G}_2) \downarrow^{A_P}_{\mathtt{p}} = \begin{cases} (\mathcal{G}_i) \downarrow^{A_P}_{\mathtt{p}} & \text{if } \mathtt{p} \in \mathcal{G}_i \text{ and } \mathtt{p} \notin \mathcal{G}_j, i \neq j \in \{1,2\} \\ \mathsf{end} & \text{if } \mathtt{p} \notin \mathcal{G}_1 \text{ and } \mathtt{p} \notin \mathcal{G}_2 \end{cases}$$

$$(4)\quad (\mu\mathbf{t}\langle\tilde{e}\rangle(\tilde{v} : \tilde{S})\{A\}.\mathcal{G}) \downarrow^{A_P}_{\mathtt{p}} = \mu\mathbf{t}\langle\tilde{e} \upharpoonright \mathtt{p}\rangle(\tilde{v} \upharpoonright \mathtt{p} : S)\{A \upharpoonright \mathtt{p}\}.(\mathcal{G}) \downarrow^{A_P}_{\mathtt{p}}$$

$$(5)\quad (\mathbf{t}\langle\tilde{e}\rangle) \downarrow^{A_P}_{\mathtt{p}} = \mathbf{t}\langle\tilde{e} \upharpoonright \mathtt{p}\rangle \qquad\qquad (6)\ (\mathsf{end}) \downarrow^{A_P}_{\mathtt{p}} = \mathsf{end}$$

If no side condition applies, $(\mathcal{G}) \downarrow^{A}_{\mathtt{p}}$ is undefined. *The* projection of $\mathcal{G}$ on $\mathtt{p}$, denoted $\mathcal{G} \upharpoonright \mathtt{p}$, is given as $(\mathcal{G}) \downarrow^{\mathsf{true}}_{\mathtt{p}}$.

In (1), value passing interactions are projected. For a send, the projection of a predicate A consists of A itself. Notice that if $\mathcal{G}$ is well-asserted then $\mathtt{p}_1$ knows all variables in A (i.e., $A \upharpoonright \mathtt{p}_1 = A$). For a receive, it is not sufficient to verify the non-violation of the current predicate only. Consider the following well-asserted global assertion:

$$\mathtt{Seller} \to \mathtt{Buyer} : k_1\,(cost : \mathsf{Int})\{cost > 10\}.\mathtt{Buyer} \to \mathtt{Bank} : k_2\,(pay : \mathsf{Int})\{pay \geqslant cost\}.\mathsf{end}$$

The predicate $pay \geqslant cost$ is meaningless to $\mathtt{Bank}$ since $\mathtt{Bank}$ does not know $cost$; rather the projection on $\mathtt{Bank}$ should be $k_2?(pay : \mathsf{Int})\{\exists cost(cost > 10 \wedge pay \geqslant cost)\}$, which incorporates the constraint between $\mathtt{Buyer}$ and $\mathtt{Seller}$. Thus (1) projects all the past predicates while hiding incorporating the constraints on interactions $\mathtt{p}_2$ does not participate through existential quantification. This makes (1) the strongest precondition i.e. it is satisfied iff $\mathtt{p}_2$ receives a legal message, avoiding the burden of defensive programming (e.g. the programmer of $\mathtt{Bank}$ can concentrate on the case $pay \leqslant 10$).

In (2), the "otw" case says the projection should be the same for all branches. In (3), each participant is in at most a single global assertion to ensure each local assertion is single threaded. In (4), the projection to $\mathtt{p}$ is the recursive assertion itself with its predicate projected on $\mathtt{p}$ by existential quantification, similarly in (5).

Example 4.2 (Projection). The projection of $\mathcal{G}_{neg}$ (Example 3.1) on `Seller` is

$$\begin{array}{lll}
\mathcal{T}_{sel} & = & \mu\mathbf{t}\langle 100\rangle(p\_v_o : \mathsf{Int})\{p\_v_o \geqslant 100\}; k_1?(v_o : \mathsf{Int})\{B\}; \mathcal{T}_2 \\
\mathcal{T}_2 & = & k_2 \oplus \{\{v_o > p\_v_o\}\mathbf{hag} : \mathbf{t}\langle v_o\rangle, \{\mathsf{true}\}\mathbf{ok} : \mathcal{T}_{ok}\} \\
\mathcal{T}_{ok} & = & \mathcal{G}_{ok} \upharpoonright \mathtt{Seller} = k_4?(v_a : \mathsf{Bool})\{B'\}
\end{array}$$

where $B = p\_v_o \geqslant 100 \wedge v_o \geqslant 100$ and $B' = \exists p\_v_o.B \wedge v_o = v_p$.

Below well-assertedness can be defined on endpoint assertions as for global assertions, characterising the same two principles discussed in §3.1.

Proposition 4.3 (Projections). *Let $\mathcal{G}$ be a well-asserted global assertion. Then for each* $\mathtt{p} \in \mathcal{G}$*, if* $\mathcal{G} \upharpoonright \mathtt{p}$ *is defined then* $\mathcal{G} \upharpoonright \mathtt{p}$ *is also well-asserted.*

5 Compositional Validation of Processes

5.1 The π-Calculus with Assertions

We use the π-calculus with multiparty sessions [20, §2], augmented with predicates for checking (both outgoing and incoming) communications.

The grammar of *asserted processes* or simply *processes* $(P, Q, \ldots)$ is given below.

$$\begin{array}{llrlrrl}
P ::= \overline{a}[2..\mathtt{n}](\tilde{s}).P & & \text{request} & \mid s \lhd \{A\}l; P & \text{select} & P_{rt} & ::= P \mid (\nu\tilde{s})P_{rt} \\
\quad\mid a[\mathtt{p}](\tilde{s}).P & & \text{accept} & \mid s \rhd \{\{A_i\}l_i : P_i\}_{i\in I} & \text{branch} & & \mid s : \tilde{h} \\
\quad\mid (\nu a)P & & \text{hide} & \mid P \mid Q & \text{parallel} & & \mid \mathsf{errH} \mid \mathsf{errT} \\
\quad\mid s!\langle\tilde{e}\rangle(\tilde{v})\{A\}; P & & \text{send} & \mid \mu X\langle\tilde{e}\tilde{t}\rangle(\tilde{v}\tilde{s}).P & \text{rec def} & e & ::= n \mid e \wedge e' \ldots \\
\quad\mid s?(\tilde{v})\{A\}; P & & \text{receive} & \mid X\langle\tilde{e}\tilde{s}\rangle & \text{rec call} & n & ::= a \mid \mathsf{true} \mid \mathsf{false} \\
\quad\mid \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q & & \text{conditional} & \mid \mathbf{0} & \text{idle} & h & ::= l \mid \tilde{n}
\end{array}$$

On the left, we define *programs*. $\overline{a}[2..\mathtt{n}](\tilde{s}).P$ multicasts a session initiationrequest to each $a[\mathtt{p}](\tilde{s}).P$ (with $2 \leqslant \mathtt{p} \leqslant n$) by multiparty synchronisation through a *shared name* a. Send, receive, and selection, all through a *session channel* s, are associated with a predicate. Branch associates a predicate to each label. Others are standard.

Runtime processes P_{rt}, given in the third column in the grammar, extend programs with runtime constructs. Process $s : h_1..h_n$ represents messages in transit through a session channel s, assuming asynchronous in-order delivery as in TCP, with each h_i denoting either a branching label or a vector of sessions/values. The empty queue is written $s : \varnothing$. Processes errH and errT denote two kinds of run-time assertion violation: errH (for "error here") indicates a predicate violation by the process itself; and errT ("error there") a violation by the environment.

Example 5.1 (Seller's Process). We set `Buyer`, `Seller`, `Bank` to be participants `1`, `2`, `3` and define a process implementing the global assertion $\mathcal{G}_{neg}$ in Examples 3.1 and 4.2 as $P_{neg} = \overline{a}[2,3](\tilde{s}).P_1 \mid a[2](\tilde{s}).P_2 \mid a[3](\tilde{s}).P_3$. Let us focus on the `Seller`

$$\begin{array}{lll}
P_2 & = & \mu X\langle 100, \tilde{s}\rangle(p\_v_o, \tilde{s}).s_1?(v_o)\{B\}; Q_2 \\
Q_2 & = & \mathsf{if}\ e\ \mathsf{then}\ (s_2 \lhd \mathbf{hag}; X\langle v_o, \tilde{s}\rangle)\ \mathsf{else}\ (s_2 \lhd \mathbf{ok}; P_{ok}) \quad \text{where} \quad P_{ok} = s_4?(v_a)\{B'\}; \mathbf{0}
\end{array}$$

where B and B' are as in Example 4.2, $\tilde{s} = s_1, .., s_4$, and Q_2 uses a policy e to select a branch (e.g., $e = \{v_o > 200 \wedge v_o > p\_v_o\}$).

$$\begin{array}{lr}
\overline{a}[2..\mathtt{n}]\,(\tilde{s}).P_1 \mid a[2]\,(\tilde{s}).P_2 \mid \ldots \mid a[\mathtt{n}]\,(\tilde{s}).P_n \to (\nu\tilde{s})(P_1 \mid P_2 \mid \ldots \mid P_n \mid s_1 : \varnothing \mid \ldots \mid s_n : \varnothing) & [\textsc{r-link}] \\
s!\langle\tilde{e}\rangle(\tilde{v})\{A\};P \mid s : \tilde{h} \to P[\tilde{\mathtt{n}}/\tilde{v}] \mid s : \tilde{h}\cdot\tilde{\mathtt{n}} \qquad (\tilde{e}\downarrow\tilde{\mathtt{n}} \wedge A[\tilde{\mathtt{n}}/\tilde{v}]\downarrow \mathsf{true}) & [\textsc{r-send}] \\
s?(\tilde{v})\{A\};P \mid s : \tilde{\mathtt{n}}\cdot\tilde{h} \to P[\tilde{\mathtt{n}}/\tilde{v}] \mid s : \tilde{h} \qquad (A[\tilde{\mathtt{n}}/\tilde{v}]\downarrow \mathsf{true}) & [\textsc{r-recv}] \\
s \rhd \{\{A_i\}l_i : P_i\}_{i\in I} \mid s : l_j\cdot\tilde{h} \to P_j \mid s : \tilde{h} \qquad (j \in I \text{ and } A_j \downarrow \mathsf{true}) & [\textsc{r-branch}] \\
s \lhd \{A\}l : P \mid s : \tilde{h} \to P \mid s : \tilde{h}\cdot l \qquad (A \downarrow \mathsf{true}) & [\textsc{r-select}] \\
\mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to P\ \ (e\downarrow \mathsf{true}) \quad \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \to Q\ \ (e\downarrow \mathsf{false}) & [\textsc{r-if}] \\
\hline
s!\langle\tilde{e}\rangle(\tilde{v})\{A\};P \to \mathsf{errH} \quad (\tilde{e}\downarrow\tilde{\mathtt{n}} \wedge A[\tilde{\mathtt{n}}/\tilde{v}]\downarrow \mathsf{false}) & [\textsc{r-sendErr}] \\
s?(\tilde{v})\{A\};P \mid s : \tilde{\mathtt{n}}\cdot\tilde{h} \to \mathsf{errT} \mid s : \tilde{h} \quad (A[\tilde{\mathtt{n}}/\tilde{v}]\downarrow \mathsf{false}) & [\textsc{r-recvErr}] \\
s \rhd \{\{A_i\}l_i : P_i\}_{i\in I} \mid s : l_j\cdot\tilde{h} \to \mathsf{errT} \mid s : \tilde{h} \quad (j \in I \text{ and } A_j \downarrow \mathsf{false}) & [\textsc{r-branchErr}] \\
s \lhd \{A\}l : P \to \mathsf{errH} \quad (A \downarrow \mathsf{false}) & [\textsc{r-selectErr}]
\end{array}$$

Fig. 3. Reduction: non-error cases (top) - error cases (bottom)

The reduction rules with predicate checking are given in Figure 3, which generate $\to$ by closing the induced relation under $\mid$ and ν and taking terms modulo the standard structural equality[2] [20]. The satisfaction of the predicate is checked at each communication action: *send*, *receive*, *selection* and *branching*, where we write $A \downarrow \mathsf{true}$ (resp. $\tilde{e} \downarrow \tilde{\mathtt{n}}$) for a closed formula A (resp. expression $\tilde{e}$) when it evaluates to true (resp. $\tilde{\mathtt{n}}$). When initiating a session, [R-LINK] establishes a session through multiparty synchronisation, generating queues and hiding all session channels. The remaining rules are standard, modelling communications in a session via queues [3, 20].

5.2 Validation Rules

For validation, we use judgements of the form $\mathcal{C};\Gamma \vdash P \rhd \Delta$, which reads: *"under $\mathcal{C}$ and Γ, process P is validated against Δ"*. Here, $\mathcal{C}$ is an *assertion environment*, which incrementally records the conjunction of predicates; hereafter, $\Gamma \vdash P \rhd \Delta$ abbreviates $\mathsf{true};\Gamma \vdash P \rhd \Delta$. Γ is a *global assertion assignment* that is a finite function mapping shared names to well-asserted global assertions and process variables to the specification of their parameters (we write $\Gamma \vdash a : \mathcal{G}$ when Γ assigns $\mathcal{G}$ to a and $\Gamma \vdash X : (\tilde{v} : \tilde{S})\mathcal{T}_1 @ \mathtt{p}_1 \ldots \mathcal{T}_n @ \mathtt{p}_\mathtt{n}$ when Γ maps X to the vector of endpoint assertions $\mathcal{T}_1 @ \mathtt{p}_1 \ldots \mathcal{T}_n @ \mathtt{p}_\mathtt{n}$ using the variables $\tilde{v}$ sorted by $\tilde{S}$). Δ is an *endpoint assertion assignment* which maps the channels for each session, say $\tilde{s}$, to a well-asserted endpoint assertion located at a participant, say $\mathcal{T} @ \mathtt{p}$.

The validation rules are given in Figure 4. In each rule, we assume all occurring (global/endpoint) assertions to be well-asserted. The rules validate the process against assertions, simultaneously annotating processes with the interaction predicates from endpoint assertions. We illustrate the key rules.

Rule [SND] validates that participant p sends values $\tilde{e}$ on session channel k, *provided* that $\tilde{e}$ satisfy the predicate under the current assertion environment; and that the con-

[2] The structural equality includes $\mu X\langle\tilde{e}\rangle(\tilde{v}\tilde{s}_1 \ldots \tilde{s}_n).P \equiv P[\mu X(\tilde{v}\tilde{s}_1 \ldots \tilde{s}_n).P/X][\tilde{e}/\tilde{v}]$ where $X\langle\tilde{e}'\tilde{s}'\rangle[\mu X(\tilde{v}\tilde{s}_1 \ldots \tilde{s}_n).P/X]$ is defined as $\mu X\langle\tilde{e}'\tilde{s}'\rangle(\tilde{v}\tilde{s}_1 \ldots \tilde{s}_n).P$.

$$\frac{\mathcal{C} \supset A[\tilde{e}/\tilde{v}] \quad \mathcal{C};\Gamma \vdash P[\tilde{e}/\tilde{v}] \rhd \Delta, \tilde{s}:\mathcal{T}[\tilde{e}/\tilde{v}] @ \mathtt{p} \quad \Gamma \vdash \tilde{e} : \tilde{S}}{\mathcal{C};\Gamma \vdash s_k!\langle\tilde{e}\rangle(\tilde{v}:\tilde{S})\{A\};P \rhd \Delta, \tilde{s}:k!(\tilde{v})\{A\};\mathcal{T} @ \mathtt{p}}\text{[Snd]}$$

$$\frac{\mathcal{C} \wedge A;\Gamma,\tilde{v}:\tilde{S} \vdash P \rhd \Delta,\ \tilde{s}:\mathcal{T} @ \mathtt{p}}{\mathcal{C};\Gamma \vdash s_k?(\tilde{v}:\tilde{S})\{A\};P \rhd \Delta,\ \tilde{s}:k?(\tilde{v}:\tilde{S})\{A\};\mathcal{T} @ \mathtt{p}}\text{[Rcv]}$$

$$\frac{\mathcal{C} \supset A_j \quad \mathcal{C};\Gamma \vdash P \rhd \Delta, \tilde{s}:\mathcal{T}_j @ \mathtt{p} \quad j \in I}{\mathcal{C};\Gamma \vdash s_k \lhd \{A_j\}l_j : P \rhd \Delta,\ \tilde{s}:k \oplus \{\{A_i\}l_i : \mathcal{T}_i\}_{i \in I} @ \mathtt{p}}\text{[Sel]}$$

$$\frac{\mathcal{C} \wedge A_i;\Gamma \vdash P_i \rhd \Delta, \tilde{s}:\mathcal{T}_i @ \mathtt{p} \quad \forall i \in I}{\mathcal{C};\Gamma \vdash\ s_k \rhd \{\{A_i\}l_i : P_i\}_{i \in I} \rhd \Delta, \tilde{s}:k\&\{\{A_i\}l_i : \mathcal{T}_i\}_{i \in I} @ \mathtt{p}}\text{[Bra]}$$

$$\frac{\mathcal{C};\Gamma \vdash P \rhd \Delta, \tilde{s}:(\Gamma(a) \restriction \mathtt{p}) @ \mathtt{p} \quad \mathtt{p} \gneq 1}{\mathcal{C};\Gamma \vdash a[\mathtt{p}](\tilde{s}).P \rhd \Delta}\text{[Macc]} \qquad \frac{\mathcal{C};\Gamma \vdash P \rhd \Delta, \tilde{s}:(\Gamma(a) \restriction 1) @ \mathtt{1}}{\mathcal{C};\Gamma \vdash \overline{a}[2..\mathtt{n}](\tilde{s}).P \rhd \Delta}\text{[Mcast]}$$

$$\frac{\mathcal{C} \wedge e;\Gamma \vdash P \rhd \Delta \quad \mathcal{C} \wedge \neg e;\Gamma \vdash Q \rhd \Delta}{\mathcal{C};\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \rhd \Delta}\text{[If]} \quad \frac{\mathcal{C};\Gamma \vdash P \rhd \Delta \quad \mathcal{C};\Gamma \vdash Q \rhd \Delta'}{\mathcal{C};\Gamma \vdash P \mid Q \rhd \Delta, \Delta'}\text{[Conc]} \quad \frac{\Delta\ \mathsf{end\ only}}{\mathcal{C};\Gamma \vdash \mathbf{0} \rhd \Delta}\text{[Idle]}$$

$$\frac{\mathcal{C};\Gamma, a:\mathcal{G} \vdash P \rhd \Delta \quad a \notin \mathrm{fn}(\mathcal{C},\Gamma,\Delta)}{\mathcal{C};\Gamma \vdash (\nu a:\mathcal{G})P \rhd \Delta}\text{[Hide]} \qquad \frac{\mathcal{C}';\Gamma \vdash P \rhd \Delta' \quad \mathcal{C} \supset \mathcal{C}' \quad \Delta' \Supset \Delta}{\mathcal{C};\Gamma \vdash P \rhd \Delta}\text{[Conseq]}$$

$$\frac{\mathcal{T}_1[\tilde{e}/\tilde{v}],\ldots,\mathcal{T}_n[\tilde{e}/\tilde{v}]\ \text{ well-asserted and well-typed under } \Gamma,\tilde{v}:\tilde{S}}{\mathcal{C}\ ;\ \Gamma, X:(\tilde{v}:\tilde{S})\mathcal{T}_1 @ \mathtt{p}_1..\mathcal{T}_n @ \mathtt{p}_\mathtt{n} \vdash X\langle\tilde{e}\tilde{s}_1..\tilde{s}_n\rangle \rhd \tilde{s}_1 : \mathcal{T}_1[\tilde{e}/\tilde{v}] @ \mathtt{p}_1,..,\tilde{s}_n : \mathcal{T}_n[\tilde{e}/\tilde{v}] @ \mathtt{p}_n}\text{[Var]}$$

$$\frac{\mathcal{C}\ ;\ \Gamma, X:(\tilde{v}:\tilde{S})\mathcal{T}_1 @ \mathtt{p}_1..\mathcal{T}_n @ \mathtt{p}_n \vdash P \rhd \tilde{s}_1 : \mathcal{T}_1 @ \mathtt{p}_1..\tilde{s}_n : \mathcal{T}_n @ \mathtt{p}_n}{\mathcal{C}\ ;\ \Gamma \vdash \mu X\langle\tilde{e}\tilde{s}_1..\tilde{s}_n\rangle(\tilde{v}\tilde{s}_1..\tilde{s}_n).P \rhd\ \tilde{s}_1 : \mathcal{T}_1[\tilde{e}/\tilde{v}] @ \mathtt{p}_1..\tilde{s}_n : \mathcal{T}_n[\tilde{e}/\tilde{v}] @ \mathtt{p}_n}\text{[Rec]}$$

Fig. 4. Validation rules for program phrases

tinuation is valid, once $\tilde{v}$ gets replaced by $\tilde{e}$. Dually, rule [Rcv] validates a value input against the continuation of the endpoint assertion under the extended assertion environment $\mathcal{C} \wedge A$ (i.e., the process can rely on A for the received values after the input). Rules [Sel] and [bra] are similar. Rules [Macc] and [Mcast] for session acceptance and request validate the continuation against the projection of the global assertion onto that participant ($\mathtt{n}$ is the number of participants in $\mathcal{G}$ and $\mathtt{p}$ is one of them).

Rule [If] validates a conditional against Δ if each branch is validated against the same Δ, under the extended environment $\mathcal{C} \wedge e$ or $\mathcal{C} \wedge \neg e$, as in the corresponding rule in Hoare logic. As in the underlying typing [20], rule [Conc] takes a disjoint union of two channel environments, and rule [Idle] takes Δ which only contains end as endpoint assertions. Rule [hide] is standard, assuming a is not specified in $\mathcal{C}$.

Rule [Conseq] uses the *refinement* relation $\Supset$ on endpoint assertions. If $\mathcal{T} \Supset \mathcal{T}'$, $\mathcal{T}$ specifies a *more refined behaviour* than $\mathcal{T}'$, in that $\mathcal{T}$ strengthens the predicates for send/selection, so it emits/selects less; and weakens those for receive/branching, so it can receive/accept more. Example 5.2 illustrates this intuition.

Example 5.2 (Refinement). Below, endpoint assertion $\mathcal{T}_s$ refines $\mathcal{T}_w$ (i.e., $\mathcal{T}_s \Supset \mathcal{T}_w$):

$$\begin{aligned} \mathcal{T}_s &= k_1!(v:\mathsf{Int})\{v > 10\};\ k_2?(z:\mathsf{Int})\{z > 0\};\ \ k_3\&\{\{\mathsf{true}\}\mathbf{l1}: \mathcal{T}_1, \{v > 100\}\mathbf{l2}: \mathcal{T}_2\} \\ \mathcal{T}_w &= k_1!(v:\mathsf{Int})\{v > 0\};\ \ k_2?(z:\mathsf{Int})\{z > 10\};\ k_3\&\{\{v > 100\}\mathbf{l1}: \mathcal{T}_1\} \end{aligned}$$

$\mathcal{T}_s$ has a stronger obligation on the sent value v, and a weaker reliance on the received value z; while $\mathcal{T}_s$ has a weaker guarantee at **l1** and offers one additional branch.

The formal definition is in [31], where we also show that the refinement relation is decidable if we restrict the use of recursive assertions so that only those in identical shapes are compared, which would suffice in many practical settings.

Rule [Var] validates an instantiation of X with expressions against the result of performing the corresponding substitutions over endpoint assertions associated to X (in the environment). In [Rec], a recursion is validated if the recursion body P is validated against the given endpoint assertions for its zero or more sessions, under the same endpoint assumptions assigned to the process variable X. The validity of this rule hinges on the partial correctness nature of the semantics of the judgement.

Example 5.3 (Validating Seller Process). We validate the `Seller` part of P_{neg} in Example 5.1 using $\mathcal{T}_{sel}$ from Example 3.1. We focus on one branch of Q_2 in P_{neg} and associate each $s_1,\ldots,s_4$ of P_{neg} to a channel $k_1,\ldots,k_4$ of $\mathcal{T}_{neg}$, respectively. Recall that $B = p\_v_o \geqslant 100 \wedge v_o \geqslant 100, A1 = v_o > p\_v_o$, and $A2 = \exists v_p.p\_v_o \geqslant 100 \wedge v_o \geqslant 100 \wedge v_o = v_p$. Below $Q_{ok} = s_4?(v_a)\{B'\};\mathbf{0}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{-}{(B \wedge \neg e \wedge B'),\Gamma \vdash \mathbf{0} \rhd t : \mathsf{end} @ 2}\;[\textsc{Idle}]}{(B \wedge \neg e),\Gamma \vdash s_4?(v_a)\{B'\};\mathbf{0} \rhd \tilde{s} : k_4?(v_a : \mathsf{Int})\{B'\};\mathsf{end} @ 2}\;[\textsc{Rcv}]}{(B \wedge \neg e) \supset A1 \quad (B \wedge \neg e),\Gamma \vdash Q_{ok} \rhd \tilde{s} : \mathcal{T}_{ok} @ 2}\;\text{(substituting)}}{B \wedge \neg e,\Gamma \vdash s_2 \lhd \mathbf{ok};Q_{ok} \rhd \tilde{s} : k_2 \oplus \{\{\mathsf{true}\}\mathbf{ok} : \mathcal{T}_{ok}, \{A1\}\mathbf{hag} : \mathbf{t}\langle v_o\rangle\} @ 2 \qquad \ldots}\;[\textsc{Sel}]}{B,\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ (s_2 \lhd \mathbf{hag};X\langle v_o,\tilde{s}\rangle)\ \mathsf{else}\ (s_2 \lhd \mathbf{ok};s_4?(v_a)\{B'\};\mathbf{0}) \rhd \tilde{s} : \mathcal{T}_2 @ 2}\;[\textsc{If}]$$

$$\dfrac{B,\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ (s_2 \lhd \mathbf{hag};X\langle v_o,\tilde{s}\rangle)\ \mathsf{else}\ (s_2 \lhd \mathbf{ok};s_4?(v_a)\{B'\};\mathbf{0}) \rhd \tilde{s} : \mathcal{T}_2 @ 2}{\mathsf{true},\Gamma \vdash s_1?(v_o)\{B\};Q_2 \rhd \tilde{s} : k_1?(v_o : \mathsf{Int})\{B\};\mathcal{T}_2 @ 2}\;[\textsc{Rcv}]$$

The ... on the premise of [IF] indicates the missing validation of the first branch. The interested reader may refer to [31] for a complete validation example with recursion.

6 Error-Freedom and Completeness

6.1 Semantics of Assertions

The semantics of asserted processes is formalised as a labelled transition relation that uses the following labels

$$\alpha ::= \overline{a}[2..n](\tilde{s}) \mid a[i](\tilde{s}) \mid s!\tilde{n} \mid s?\tilde{n} \mid \mid s \lhd l \mid s \rhd l \mid \tau$$

for session requesting/accepting, value sending/receiving, selection, branching, and the silent action, respectively. We write $P \xrightarrow{\alpha} Q$ when P has a one-step transition α to Q. The transition rules are the standard synchronous ones[3] except that: (*i*) predicates are checked at each communication action and, if the predicate is violated, in the case of input/branching action the process moves to errT, in the case of an output/selection the process moves to errH with τ-action, (*ii*) they include the reduction semantics given in § 5.1 (i.e., $P \to Q$ induces $P \xrightarrow{\tau} Q$).

The semantics of endpoint assertions is defined as another labelled transition relation, of form $\langle\Gamma,\Delta\rangle \xrightarrow{\alpha} \langle\Gamma',\Delta'\rangle$, which reads: *the specification* $\langle\Gamma,\Delta\rangle$ allows *the action* α*, with* $\langle\Gamma',\Delta'\rangle$ *as the specification for its continuation.* In this transition relation, only legitimate (assertion-satisfying) actions are considered.

We define the semantic counterpart of $\Gamma \vdash P \rhd \Delta$ by using a simulation between the transitions of processes and those of assertions. The simulation (Definition 6.1),

[3] The synchronous transition suites our present purpose since it describes how a process places/retrieves messages at/from queues, when message content may as well be checked.

requires an input/branching action to be simulated only for "legal" values/labels, i.e. for actions in which predicates are not violated. Intuitively, we demand conformance to a proper behaviour only if the environment behaves properly. Below we use the *predicate erasure* to show that the validation can prevent bad behaviour even without runtime predicate checking, writing $\mathsf{erase}(P)$ for the result of erasing all predicates from P. Similarly $\mathsf{erase}(\Gamma)$ and $\mathsf{erase}(\Delta)$ erase predicates from the underlying session types, giving the *typing* environments. P is *closed* if it is without free variables.

Definition 6.1 (Conditional Simulation). Let $\mathcal{R}$ be a binary relation whose elements relate a closed process P without errH or errT and a pair of assignments $\langle\Gamma,\Delta\rangle$ such that $\mathsf{erase}(\Gamma) \vdash \mathsf{erase}(P) \rhd \mathsf{erase}(\Delta)$ in the typing rules in [20, §4]. Then $\mathcal{R}$ is a *conditional simulation* if, for each $(P,\langle\Gamma,\Delta\rangle) \in \mathcal{R}$:

1. for each input/branching/session input $P \xrightarrow{\alpha} P'$, $\langle\Gamma,\Delta\rangle$ has a respective move at $\mathsf{sbj}(\alpha)$ (the subject of α) and, if $\langle\Gamma,\Delta\rangle \xrightarrow{\alpha} \langle\Gamma',\Delta'\rangle$ then $(P',\langle\Gamma',\Delta'\rangle) \in \mathcal{R}$.
2. for each output/selection/τ/session output move $P \xrightarrow{\alpha} P'$, $\langle\Gamma,\Delta\rangle \xrightarrow{\alpha} \langle\Gamma',\Delta'\rangle$ such that $(P',\langle\Gamma',\Delta'\rangle) \in \mathcal{R}$.

If $\mathcal{R}$ is a conditional simulation we write $P \precsim \langle\Gamma,\Delta\rangle$ for $(P,\langle\Gamma,\Delta\rangle) \in \mathcal{R}$.

The conditional simulation requires P to be well-typed against $\mathsf{erase}(\Gamma)$ and $\mathsf{erase}(\Delta)$. Without this condition, the inaction $\mathbf{0}$ would conditionally simulate any Δ. This stringent condition can be dropped, but our interest is to build an assertion semantics on the basis of the underlying type discipline.

Definition 6.2 (Satisfaction). Let P be a closed program and Δ an end-point assertion assignment. If $P \precsim \langle\Gamma,\Delta\rangle$ then we say that P *satisfies* Δ *under* Γ, and write $\Gamma \models P \rhd \Delta$. The satisfaction is extended to open processes, denoted $\mathcal{C};\Gamma \models P \rhd \Delta$, by considering all closing substitutions respecting Γ and $\mathcal{C}$ over Δ and P.

The judgement $\Gamma \models P \rhd \Delta$ in Definition 6.2 states that (1) P will send valid messages or selection labels; and (2) P will continue to behave well (i.e., without going into error) w.r.t. the continuation specification after each valid action in (1) as well as after receiving each valid message/label (i.e. which satisfies an associated predicate). The satisfaction is about partial correctness since if P (is well-typed and) has no visible actions, the satisfaction trivially holds.

6.2 Soundness, Error Freedom and Completeness

To prove soundness of the validation rules, we first extend the validation rules to processes with queues, based on the corresponding typing rules in [3, 20].

Proposition 6.3 (Subject Reduction). *Let $\Gamma \vdash P \rhd \Delta$ be a closed program and suppose we have $\langle\Gamma,\Delta\rangle \xrightarrow{\alpha_1..\alpha_n} \langle\Gamma',\Delta'\rangle$. Then $P \xrightarrow{\alpha_1..\alpha_n} P'$ implies $\Gamma' \vdash P' \rhd \Delta'$.*

The proof uses an analysis of the effects of τ-actions on endpoint assertions, observing the reduction at free session channels changes the shape of the session typing [3, 20].

Let $\Delta \Supset \Delta'$ be a point-wise extension of $\Supset$ (defined when $\mathrm{dom}(\Delta) = \mathrm{dom}(\Delta')$); Proposition 6.4 says that a process satisfying a stronger specification also satisfies a weaker one. Using these results we obtain Theorem 6.5.

Proposition 6.4 (Refinement). *If* $\Gamma \models P \rhd \Delta$ *and* $\Delta \Supset \Delta'$ *then* $\Gamma \models P \rhd \Delta'$.

Theorem 6.5 (Soundness of Validation Rules). *Let P be a program. Then* $\mathcal{C};\Gamma \vdash P \rhd \Delta$ *implies* $\mathcal{C};\Gamma \models P \rhd \Delta$.

A direct consequence of Theorem 6.5 is the error freedom of validated processes. Below we say $\langle \Gamma, \Delta \rangle$ *allows* a sequence of actions $\alpha_1..\alpha_n$ $(n \geqslant 0)$ if for some $\langle \Gamma', \Delta' \rangle$ we have $\langle \Gamma, \Delta \rangle \xrightarrow{\alpha_1..\alpha_n} \langle \Gamma', \Delta' \rangle$.

Theorem 6.6 (Predicate Error Freedom). *Suppose P is a closed program,* $\Gamma \vdash P \rhd \Delta$ *and* $P \xrightarrow{\alpha_1..\alpha_n} P'$ *such that* $\langle \Gamma, \Delta \rangle$ *allows* $\alpha_1..\alpha_n$. *Then P' contains neither* errH *nor* errT.

The proof system is complete relative to the decidability of the underlying logic for processes without hidden shared names. We avoid name restriction since it allows us to construct a process which is semantically equivalent to the inaction if and only if interactions starting from a hidden channel terminate. Since we can simulate arbitrary Turing machines by processes, this immediately violates completeness. In this case, non-termination produces a *dead code*, i.e. part of a process which does not give any visible action, which causes a failure in completeness.[4]

For each program without hiding, we can compositionally construct its "principal assertion assignment" from which we can always generate, up to $\Supset$, any sound assertion assignment for the process. Since the construction of principal specifications is compositional, it immediately gives an effective procedure to check $\models$ as far as $\Supset$ is decidable (which is relative to the underlying logic). We conclude:

Theorem 6.7 (Completeness of Validation Rules for Programs without Hiding). *For each closed program P without hiding, if* $\Gamma \models P \rhd \Delta$ *then* $\Gamma \vdash P \rhd \Delta$. *Further* $\Gamma \models P \rhd \Delta$ *is decidable relative to the decidability of* $\Supset$.

7 Extensions and Related Work

Extensions to shared and session channel passing. The theory we have introduced in the preceding sections directly extends to shared channel passing and session channel passing, or delegation, carrying over all formal properties. In both cases, we have only to add predicate annotations to channels in assertions as well as in asserted processes. The shape of the judgement and the proof rules do not change, similarly the semanics of the judgement uses a conditional simulation. We obtain the same soundness result as well as completeness of the proof rules for the class of processes whose newly created channels are immediately exported. Since the presentation of such extension would require a detailed presentation of the notion of refinement, for space constraints and simplicity of presentation we relegate it to [31].

Hennessy-Milner logic for the π-calculus. Hennessy-Milner Logic (HML) is an expressive modal logic with an exact semantic characterisation [17]. The presented theory addresses some of the key challenges in practical logical specifications for the π-calculus, unexplored in the context of HML. First, by starting from global assertions, we

[4] Not all dead codes cause failure in completeness. For example a dead branch in a branching/-conditional does not cause this issue since the validation rules can handle it.

gain in significant concision of descriptions while enjoying generality within its scope (properties of individual protocols). Previous work [2, 11] show how specifications in HML, while encompassing essentially arbitrary behavioural properties of processes, tend to be lengthy from the practical viewpoint. In this context, the direct use of HML is tantamount to *reversing* the methodology depicted in Figure 1 of § 1: we start from endpoint specifications and later try to check their mutual consistency, which may not easily yield understandable global specifications.

As another practical aspect, since $\Supset$ is decidable for practically important classes assertions [31], the present theory also offers algorithmic validation methods for key engineering concerns [32] including consistency of specifications (cf. §3.1) and correctness of process behaviours with full recursion against non-trivial specifications (cf. Theorem 6.7), whose analogue may not be known for the general HML formulae on the π-calculus. The use of the underlying type structures plays a crucial role.

From the viewpoint of logical specifications for name passing, the present theory takes an *extensional* approach: we are concerned with what behaviours will unfold starting from given channels, than their (in)equality [11]. While our approach does reflect recommended practices in application-level distributed programming (where the direct use of network addresses is discouraged), it is an interesting topic to study how we can treat names as data as studied in [11].

Corresponding assertions and refinement/dependent types. The work [6] combines session-types with *correspondence assertions*. The type system can check that an assertion **end** L, where L is a list of values (not a logical formula), is matched by the corresponding **begin** effect.

The use of session types to describe behavioural properties of objects and components in CORBA is studied in [33]. In another vein, the refinement types for channels (e.g. [5]) specify value dependency with logical constraints. For example, one might write $?(x\colon \mathsf{int}, !\{y : \mathsf{int} \mid y > x\})$ using the notations from [15, 34]. It specifies a dependency at a *single point* (channel), unable to describe a constraint for a series of interactions among multiple channels. Our theory, based on multiparty sessions, can verify processes against a contract globally agreed by multiple distributed peers.

Contract-based approaches to functions and communications and functions. Verification using theories of contracts for programming functional languages, with applications to the validation of financial contracts, is studied in [29, 35]. Our theory uses the π-calculus with session types as the underlying formalism to describe contracts for distributed interactions. We observe that a contract-based approach for sequential computing is generally embeddable to the present framework (noting that function types are a special form of binary session types and that the pre/post conditions in sequential contracts are nothing but predicates for interactions resulting from the embedding); it is an interesting subject of study to integrate these and other sequential notions of contracts into the present framework, which would enable a uniform reasoning of sequential and concurrent processes.

In [8, 12] use c-semirings to model constraints that specify a Service Level Agreement. It would be interesting to consider global assertions where the logical language is replaced with c-semirings. This would allow global assertions to express soft constraints but it could affect the effectiveness of our approach. However c-semirings do not feature negation and the decidability of logics based on c-semrings has not been deeply investigated.

The global consistency checking is used in advanced security formalisms. In [16] a rely-guarantee technique is applied to a trust-management logic. The main technical difference is that users have to directly annotate each participant with assertions because of the the absence of global assertions. In [4] cryptography is used to ensure integrity of sessions but logical contracts are not considered.

Theories of contracts for web services based on advanced behavioural types are proposed, including those using CCS [7], π-calculus [10], and conversation calculus [9]. Some of the authors in this line of study focus on *compliance* of client and services, often defining compliance in terms of deadlock-freedom, e.g., in [1] a type system guaranteeing a progress property of clients is defined.

Our approach differs from the preceding works in its use of global assertions for elaborating the underlying type structure, combined with the associated compositional proof system. This permits us to express and enforce fine-grained contracts of choreographic scenarios. Global/endpoint assertions can express constraints over message values (including channels), branches and invariants, which cannot be represented by types alone, cf. [20]. The enriched expressiveness of specifications introduces technical challenges: in particular, consistency of specifications becomes non-trivial. The presented consistency condition for global assertions is mechanically checkable relatively to the decidability of the underling logic, and ensures that the end-point assertions are automatically consistent when projected. On this basis a sound and relatively complete proof system is built that guarantees semantic consistency.

As a different DbC-based approach to concurrency, an extension of DbC has been proposed in [27], using contracts for SCOOP [26] in order to reason about liveness properties of concurrent object-oriented systems. The main difference of our approach from [27] is that our framework specifies focuses on systems based on distributed message passing systems while [27] treats shared resources. The notion of pre-/post-conditions and invariants for global assertions centring on communications and the use of projections are not found in [27]. The treatment of liveness in our framework is an interesting topic for further study.

References

1. L. Acciai and M. Borale. A type system for client progress in a service-oriented calculus. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 625–641. Springer, 2008.
2. M. Berger, K. Honda, and N. Yoshida. Completeness and logical full abstraction for modal logics for the typed π-calculus. In *ICALP'*, volume 5126 of *LNCS*, pages 99–111, 2008.
3. L. Bettini et al. Global Progress in Dynamically Interfered Multiparty Sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
4. K. Bhargavan, R. Corin, P.-M. Deniélou, C. Fournet, and J. Leifer. Cryptographic protocol synthesis and verification for multiparty sessions. In *CSF*, pages 124–140, 2009.
5. K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *POPL*, pages 445–456, 2010.
6. E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence assertions for process synchronization in concurrent communications. *JFC*, 15(2):219–247, 2005.

7. M. Bravetti and G. Zavattaro. A foundational theory of contracts for multi-party service composition. *Fundamenta Informaticae*, XX:1–28, 2008.
8. M. Buscemi and U. Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *ESOP*, volume 4421 of *LNCS*, pages 18–32, 2007.
9. L. Caires and H. T. Vieira. Conversation types. In *ESOP*, volume 5502 of *LNCS*, pages 285–300, 2009.
10. G. Castagna and L. Padovani. Contracts for mobile processes. In *CONCUR*, volume 5710 of *LNCS*, pages 211–228. Springer, 2009.
11. M. Dam. Proof systems for pi-calculus logics. In *Logic for Concurrency and Synchronisation*, Trends in Logic, Studia Logica Library, pages 145–212. Kluwer, 2003.
12. R. De Nicola et al. A Basic Calculus for Modelling Service Level Agreements. In *Coordination*, volume 3454 of *LNCS*, pages 33 – 48. Springer, 2005.
13. R. W. Floyd. Assigning meaning to programs. In *Proc. Symp. in Applied Mathematics*, volume 19, 1967.
14. D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.
15. T. Freeman and F. Pfenning. Refinement types for ml. *SIGPLAN Not.*, 26(6):268–277, 1991.
16. J. D. Guttman et al. Trust management in strand spaces: A rely-guarantee method. In *ESOP*, volume 2986 of *LNCS*, pages 325–339. Springer, 2004.
17. M. Hennessy and R. Milner. Algebraic laws for non-determinism and concurrency. *JACM*, 32(1), 1985.
18. T. Hoare. An axiomatic basis of computer programming. *CACM*, 12, 1969.
19. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer, 1998.
20. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.
21. *The Java Modeling Language (JML) Home Page*.
22. C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
23. K. R. M. Leino. Verifying object-oriented software: Lessons and challenges. In *TACAS*, volume 4424 of *LNCS*, page 2, 2007.
24. E. Mendelson. *Introduction to Mathematical Logic*. Wadsworth Inc., 1987.
25. B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.
26. B. Meyer. *Object-Oriented Software Construction (Chapter 31)*. Prentice Hall, 1997.
27. P. Nienaltowski, B. Meyer, and J. S. Ostroff. Contracts for concurrency. *Form. Asp. Comput.*, 21(4):305–318, 2009.
28. OMG. Object Constraint Language Version 2.0, May 2006.
29. S. Peyton Jones et al. Composing contracts: an adventure in financial engineering. In *ICFP*, pages 281–292. ACM, 2000.
30. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
31. Full version of this paper. `http://www.cs.le.ac.uk/people/lb148/fullpaper.html`.
32. SAVARA JBoss Project webpage. `http://www.jboss.org/savara`.
33. A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. *Fundamenta Informaticæ*, 73(4):583–598, 2006.
34. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM, 1999.
35. D. Xu and S. Peyton Jones. Static contract checking for Haskell. In *POPL*, pages 41–52. ACM, 2009.