Refinements for Multiparty Message-Passing Protocols

Specification-Agnostic Theory and Implementation

Martin Vassor
University of Oxford, UK

Nobuko Yoshida
University of Oxford, UK

Abstract

Multiparty message-passing protocols are notoriously difficult to design, due to interaction mismatches that lead to errors such as deadlocks. Existing protocol specification formats have been developed to prevent such errors (e.g. multiparty session types (MPST)). In order to further constrain protocols, specifications can be extended with *refinements*, i.e. logical predicates to control the behaviour of the protocol based on previous values exchanged. Unfortunately, existing refinement theories and implementations are tightly coupled with specification formats.

This paper proposes a framework for multiparty message-passing protocols with refinements and its implementation in Rust. Our work *decouples* correctness of refinements from the underlying model of computation, which results in a *specification-agnostic* framework.

Our contributions are threefold. First, we introduce a trace system which characterises *valid refined traces*, i.e. a sequence of sending and receiving actions correct with respect to refinements. Second, we give a correct model of computation named *refined communicating system* (RCS), which is an extension of communicating automata systems with refinements. We prove that RCS only produce valid refined traces. We show how to generate RCS from mainstream protocol specification formats, such as *refined multiparty session types* (RMPST) or *refined choreography automata*. Third, we illustrate the flexibility of the framework by developing both a static analysis technique and an improved model of computation for dynamic refinement evaluation. Finally, we provide a Rust toolchain for decentralised RMPST, evaluate our implementation with a set of benchmarks from the literature, and observe that refinement overhead is negligible.

2012 ACM Subject Classification Software and its engineering → Specification languages; Theory of computation → Assertions; Theory of computation → Concurrency

Keywords and phrases Message-Passing Concurrency, Session Types, Specification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.41

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.23`

Funding Work supported by: EPSRC EP/T00006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462, EP/X015955/1n NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

Acknowledgements We thank B. Ekici, M. Giunti, P. Hou, A. Suresh, and F. Zhou.

1 Introduction

Message passing programming is a notoriously difficult task with new bugs arising with respect to sequential programming, for instance deadlocks. To address this increased complexity, various specifications have been introduced (e.g., message sequence charts [24], multiparty session types [38, 19, 18], choreography automata [1]). In general, specifications are used to constrain messages, in order to prevent errors such as deadlocks (via message ordering) or payload mismatch (by enforcing the sender and the receiver of a message to agree on the datatype exchanged).

In this paper, we tackle an important and advanced aspect of protocol specification, *logical constraints* (or *contracts*) on asynchronous message-passing communications. Contracts for heterogeneous systems are predominant for correctly designing, implementing, and composing software services, and have a long history in distributed software development as found in Design-by-Contracts [28], Service Level Agreements, and Component-Based Software Engineering. With contracts, software designers can define more precise (refined) and verifiable specifications for distributed software components. Contracts have been investigated from a variety of perspectives, using many different analysis techniques and formalisms. Our goal is to distill an essence of those models for protocol refinements by answering the following questions affirmatively:

(i) what does it mean for an execution of contracts for message-passing systems to be correct;
(ii) how do we integrate a theory to a variety of models;
(iii) how do we analyse their correctness?; and
(iv) how do we implement correct systems in a programming language?

To explain our framework, consider a *guessing game* (from [41]) with three participants where the first one (participant A) chooses a *secret* integer and sends it to the second participant (B). Then, the third participant (C) tries to *guess* this number. Depending on the guess, B replies with hints (*more* and *less*) until C succeeds in guessing the *correct* value.

The developer writing the specification for such protocol would like to ensure, *in the specification*, that hints from B are consistent with the previous values exchanged. For instance, if the *secret* is 5 and the guess is 10, the specification should constrain B to send *less*. Figure 1 shows a communication diagram of the protocol with constraints (which we call *refinements*) shown in light blue.

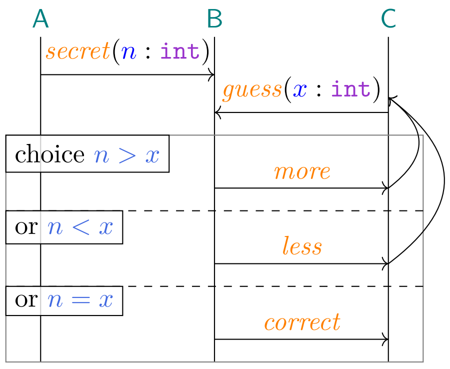


Figure 1 Communication diagram for the guessing game protocol with refinements.

In this paper, we develop a formal framework for refinements, agnostic to any particular specification formalism. Its core part is composed of a characterisation of refinement correctness: *Valid Refined Traces*, and a model of computation: *Refined Communicating Systems* (RCS), where communication is asynchronous and refinements are centrally and dynamically evaluated. For illustration, we use *Refined Multiparty Session Types* as the main specification format for multiparty protocols.

In addition, we demonstrate the versatility of our framework with multiple extensions. First, our framework can accommodate other protocol specification formats (e.g. choreography automata [1]). Second, it is used as a baseline for improved refinement evaluation: we present an optimised model of computation (decentralised refinement evaluation). Finally, it is also used as a baseline for implementing static analysis techniques: we present a simple strategy for statically removing redundant refinements.

Valid Refined Traces. The first building block is a common notion of *correct* executions with respect to added refinements. We introduce *valid refined traces* which are consistent traces with respect to refinements. This approach allows us to establish a general notion of refinements, which is applicable to different logics for constraints, type theories, models of computations, and programming languages. We consider asynchronous communications (FIFOs), distinguishing *sending* and *receiving* actions in traces.

To illustrate our approach, consider the guessing game example shown above. Each execution of that protocol is recorded in a trace, i.e. a sequence of the individual events that take place during the execution (c.f. Section 2.2). For instance, a possible trace of the first four events of the protocol is the following:

$$\mathsf{A}!\mathsf{B}\langle \mathit{secret}, \langle n, 5\rangle\rangle : \top \cdot \mathsf{A}?\mathsf{B}\langle \mathit{secret}, \langle n, 5\rangle\rangle : \top \cdot \mathsf{C}!\mathsf{B}\langle \mathit{guess}, \langle x, 5\rangle\rangle : \top \cdot \mathsf{C}?\mathsf{B}\langle \mathit{guess}, \langle x, 5\rangle\rangle : \top$$

This trace contains four actions, and each action records an event, i.e. a message emission (denoted with a !) or reception (denoted with a ?). For instance, $\mathsf{A}!\mathsf{B}\langle \mathit{secret}, \langle n, 5\rangle\rangle : \top$ records A sending a message to B, the payload of this message is a variable n, which has value 5. In the first four actions, we do not need any constraint, therefore actions are guarded by $\top$ which denotes a tautology predicate. The next action following this trace would be for B to send either *more*, *correct*, or *less*. Choosing *more* or *less* would be inconsistent with our protocol, since C guessed the correct number. For instance, choosing *more* would add the action $\mathsf{B}!\mathsf{C}\langle \mathit{more}\rangle : n > x$ at the end of the queue: the refinement $n > x$ would be violated, since $x = n = 5$.

Valid Refined Traces characterise consistency based on the produced trace; and we aim to provide a model of computation constrained in a way that prevents such inconsistent choices.

Refined Communicating Systems. The second building block of our framework is a model of computation that only produces correct traces. *Communicating Systems* (CS) [5] are a model of concurrent computation, where *Communicating Finite State Machines* communicate asynchronously using unbounded FIFO queues. CS are often used to model and implement MPST [12, 13, 7]. We adapt CS to accommodate refinements, which we call *Refined Communicating Systems* (RCS). The semantics is modified in order to check refinements at every step. For this, we introduce a shared map in order to keep track of variables and their values that are exchanged in messages (e.g. the values of x and n in the guessing game example). This record of values is used to evaluate refinements, preventing undesired transitions. In this paper, we show that RCS only produce valid refined traces and we explain how to generate an RCS from a RMPST.

Refined MPST. Working with CS is cumbersome, and, in practice, we would prefer to adapt existing specification formats. We present in depth how to integrate refinements in *Multiparty Session Types* (MPST) [38, 19, 18], which are a family of type systems that aims to prevent communication bugs.

The following refined global type ($G_{\pm}$) is a specification of the guessing game protocol (Figure 1), with refinements: a participant A begins by sending a *secret* to B; the value of the *secret* is stored in the variable n. Then, C tries to guess the value (stored in variable x), and B replies with *more*, *less* (in which case the protocol loops and C can make another guess: $\mu\mathbf{T}.G$ denotes the recursion) or *correct*, at which point the protocol terminates (`end` denotes the termination). The refinements specify conditions upon which the *more*, *less*, and *correct* branches are possible. For instance, the protocol can take the *correct* branch only if the values in the *secret* and the *guess* messages are the same, i.e. if $x = n$.

$$G_{\pm} = \\ \mathsf{A} \to \mathsf{B} \left\{ \mathit{secret}(n : \mathtt{int} \models \top).\mu\mathbf{T}.\mathsf{C} \to \mathsf{B} \left\{ \mathit{guess}(x : \mathtt{int} \models \top).\mathsf{B} \to \mathsf{C} \left\{ \begin{array}{l} \mathit{more}(\models x < n).\mathbf{T}, \\ \mathit{less}(\models x > n).\mathbf{T}, \\ \mathit{correct}(\models x = n).\mathtt{end} \end{array} \right\} \right\} \right\}$$

Compared to standard MPST, *Refined MPST* (RMPST) contain variable names (n and x) and refinements (denoted with $\models r$ in the payloads, meaning that to send the message, r must hold). We present those extensions as well as the relation between RMPST and RCS.

Applications and Extensions. To show the versatility of our framework, we extend it:

Decentralised Refinement Evaluation: The canonical semantics for RCS we present uses a single shared map of variables to provide a simple way to reason about refinements. Having this global map would not be suited for a distributed implementation. We extend our framework with an alternative semantics where each participant of a protocol has a local map of variables. We show that if variables are not duplicated, then this alternative model also produces valid refined traces.

Static Elision of Redundant Refinements: At places where refinements are redundant (e.g. where it is entailed by previous refinements), we could benefit from removing those refinements. In order to show the versatility of our framework, we show how to develop a simple static analysis technique to remove such redundant refinements.

Refined Choreography Automata: While we mostly use RMPST as an example of protocol specification language, we sketch another specification by (informally) presenting how to integrate refinements in choreography automata (in [35]).

Rust Implementation. The last objective of our work is to implement RMPST into Rust. We choose Rust for several reasons: its affine type system makes it easy to avoid unwanted reuse of values, which helps to prevent a participant from duplicating actions; and thanks to its growing popularity, there are already a few existing toolchains for session types in Rust [27, 6, 26, 25]. Among them, we choose Rumpsteak [7] since it already uses CS to implement MPST participants inside its toolchain. We extend Rumpsteak with refinements using the decentralised refinement evaluation approach. We finally measure the refinement overhead in Rumpsteak.

Contributions and Outline. Our main contribution is to unify the different points presented above in a *single* framework as presented in Figure 2. We introduce a uniform framework which is agnostic to any particular specification formalism, model, semantics and language, defining the correctness of refinements as validity of traces. We then prove the safety of the framework (Theorem 18). We demonstrate the *versatility of our framework* by accommodating *multiple protocol specifications* such as (refined) multiparty session types [38, 19, 18, 42] and (refined) choreography automata [1, 16], *multiple semantics* such as (refined) communicating automata [5] with centralised and decentralised semantics, and *multiple analysis techniques* such as dynamic and static analyses. We provide an implementation of an instance of the framework in Rust. Our framework is the first, to the best of our knowledge, to achieve such versatility.

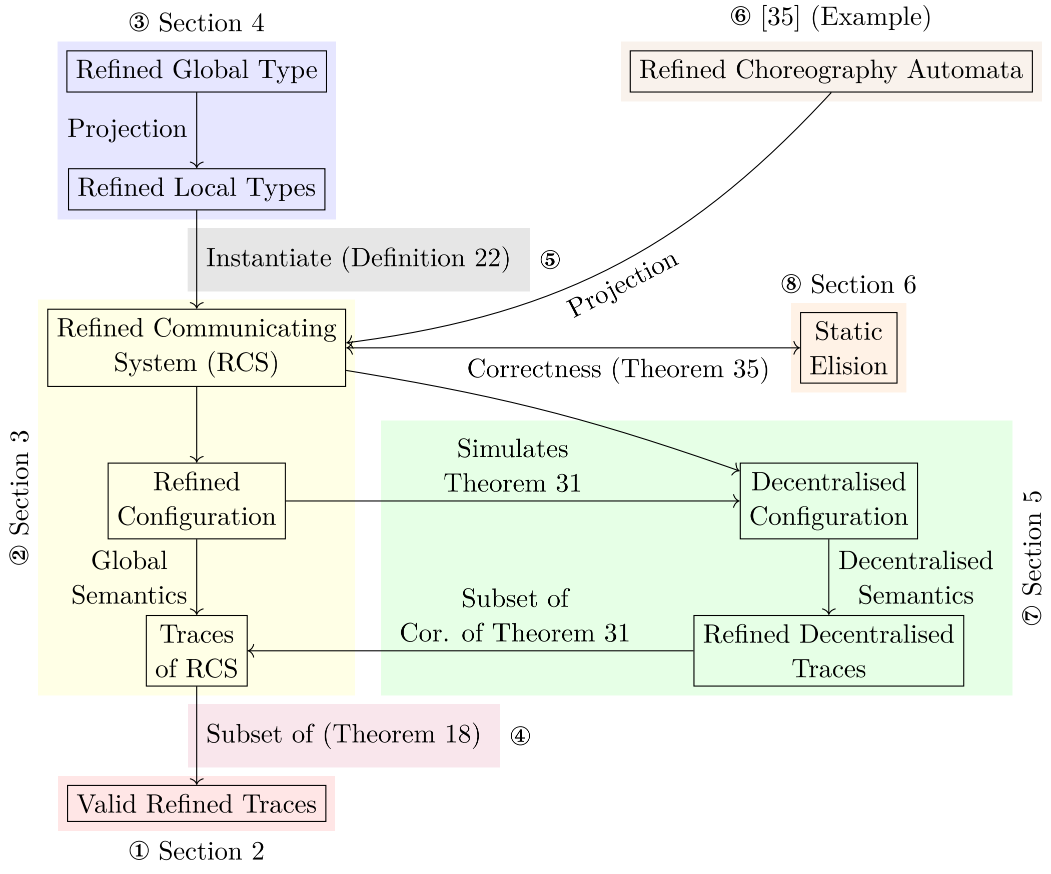


Figure 2 Overview of the framework for RMPST developed in this work. The coloured backgrounds show the main steps of this paper.

The framework is composed of the following parts (circled numbers refer to Figure 2):

① **Valid Refined Traces:** We introduce *valid refined traces* which characterise valid executions with respect to refinements.

② **Refined Communicating Systems (RCS):** We extend Communicating Systems to accommodate refinements. From a configuration of RCS, we induce a set of possible traces. One of our main results is Theorem 18 (④), which states that all traces produced by RCS are valid refined traces, which in turn proves the correctness of the RCS.

③ **Refined Multiparty Session Types (RMPST):** In Section 4, we adapt MPST (which consists of *global types* (which describe a multiparty protocol), *local types* (which describe the behaviour of a single participant), and a *projection* from global to local types which extracts the behaviour of a single participant) to accommodate for refinements. We show how to generate a RCS from a set of local types with refinements (⑤). In addition, in [35], we sketch how to accommodate refinements in choreography automata, to illustrate the versatility of the framework (⑥).

⑦ and ⑧ **Optimisations:** In Section 5 (⑦), we propose a *decentralised* model as an alternative for RCS. We show trace inclusion w.r.t. RCS, which ensures refinements are correctly checked. In Section 7, we implement this improved model in Rust. In addition, in Section 6, we demonstrate how to develop analysis techniques using the framework. We show how redundant refinements can, under some conditions, be statically removed (⑧).

2 Refined Traces and their Validity

This section introduces *refined traces* which are sequences of messages *actions*. We then define their *validity*, introducing two definitions on traces, *well-queued* and *well-predicated* traces. We precede this (in Section 2.1) with preliminary definitions used throughout this paper.

2.1 Preliminaries: Predicates Language and Semantics

This first subsection introduces the basic definitions we use in this paper.

Let $\mathbb{V}$ be a set of variables, ranged over by x, y, ...; and a finite set $\mathbb{C}$ of values (in this work, we take 32-bit integers: $\mathbb{Z}/2^{32}\mathbb{Z}$).

We use associative maps from variable names to values, noted M. $\mathtt{dom}(M)$ denotes the domain of a map, that is the set of variables that appear in the map. Maps are equipped with lookup ($M(x)$), update ($M[x \mapsto c]$) and removal ($M \setminus x$) operations. $M_1 \uplus M_2$ denotes the union of M_1 and M_2 if their domains are disjoint (see [35] for the definition of all those operators). Finally, $M_\varnothing$ denotes an empty map.

In order to keep our work general, we do not strictly specify the language of predicates, nor their semantics rules. Instead, we suppose we are given a language to express refinements, whose terms are produced by a rule $\mathcal{R}$. In this paper, we intentionally leave the logic underspecified so that it can be fine tuned by the end user. In practice, in our implementation (Section 7), custom predicates can easily be added. In the following, we use a simple grammar with arithmetic and relational operators as predicates. Let $\mathbb{R}$ be the set of refinement expressions. We assume refinements can have free variables, and that there exist a function $\mathtt{fv} : \mathbb{R} \to \mathcal{P}(\mathbb{V})$ that gives the free variables of each refinement expression. We note $\mathbb{R}_\mathbb{W}$ be the set of refinements of $\mathbb{R}$ whose set of free variables is $\mathbb{W} \subseteq \mathbb{V}$. We assume a variable substitution function, $\mathcal{R}\{^{v_i}/_{x_i}\}$ that substitutes every free occurrence of each variable x_i for the value v_i. For any refinement expression r, $r\{^{\cdots}/_{\mathtt{fv}(r)}\}$ is a closed refinement. Since our predicates are abstract, we do not explicitly specify their semantics, nor their well-formedness. Instead, we assume each closed refinement formula evaluates to $\top$ or $\bot$. We assume there exists a function $\mathtt{eval}(r)$ that evaluates the refinement r, provided that r is closed[1]. Finally, we assume the existence of a closed formula $\top$ that is a tautology, i.e. $\mathtt{eval}(\top) = \top$.

Given a map M and a refinement r, we note $M \models r$ if and only if the refinement r is closed under the map M: $\mathtt{fv}(r) \subseteq \mathtt{dom}(M)$, and evaluates to $\top$ after substitution: $\mathtt{eval}(r\{^{M(\mathtt{fv}(r))}/_{\mathtt{fv}(r)}\}) = \top$.

In a protocol with multiple participants, let $\mathbb{P}$ be a set of participants ranged over by $\mathsf{A}, \mathsf{B}, \ldots$ and $\mathsf{p}, \mathsf{q}, \ldots$ being meta-variables over participant names. In this work, messages contain a label, a variable, and a value. Let $\mathbb{L}$ be a set of labels; ℓ and its decorated variants range over labels in $\mathbb{L}$. We define $\mathbb{M} = \mathbb{L} \times (\mathbb{V} \times \mathbb{C})$ for the set of messages (as a reminder: $\mathbb{L}$ is the set of labels, $\mathbb{V}$ the set of variables, and $\mathbb{C}$ the set of values).

2.2 Traces

Let us denote $\vec{e} = e_1 {::} \ldots {::} e_n$ ($n \geq 0$) as a *FIFO*, i.e., a finite sequence of elements e_i (messages exchanged in this paper). We use ε for an empty FIFO ($n = 0$). We define: $\mathtt{enq}(\vec{e}, e) \stackrel{\text{def}}{=} e{::}\vec{e}$; $\mathtt{deq}(\vec{e}{::}e) \stackrel{\text{def}}{=} \vec{e}$ ($\mathtt{deq}(\varepsilon)$ is undefined); and $\mathtt{next}(\vec{e}{::}e) \stackrel{\text{def}}{=} e$ ($\mathtt{next}(\varepsilon)$ is undefined). Notice

[1] We do not discuss the decidability of the actual chosen logic of refinements here. For undecidable logics, providing such function is, of course, not possible; however this is not in the scope of this work.

that $\mathtt{deq}(\vec{e})$ is defined if and only if $\mathtt{next}(\vec{e})$ is defined. In this paper, we consider one FIFO channel per pair of participant. We call *queues* a map of all pairs of distinct participants to their communication FIFO of a system. We note $\mathtt{enq}_{(\mathsf{p},\mathsf{q})}(w, e)$, $\mathtt{deq}_{(\mathsf{p},\mathsf{q})}(w)$, $\mathtt{next}_{(\mathsf{p},\mathsf{q})}(w)$, where the indices indicates which FIFO of the set is affected (see [35] for the formal definition). We write $w_\varnothing$ for the empty queue, which is the queue where $w_{(\mathsf{p},\mathsf{q})} = \varepsilon$ for all p and q.

Actions are tuples consisting of a sending participant p, a direction of communication $\dagger \in \{!, ?\}$ (! stands for sending, and ? stands for receiving), a receiving participant q, a message m and a predicate r associated to the action (as a reminder: $\mathbb{R}$ is the set of refinements). We require participants to be distinct (i.e. $\mathsf{p} \neq \mathsf{q}$).

▶ **Definition 1** (Action and Trace). *An action is an element of $\mathbb{A}$ defined as follows: $\mathbb{A} = \mathbb{P} \times \{!, ?\} \times \mathbb{P} \times \mathbb{M} \times \mathbb{R}$. We write $\alpha = \mathsf{p}\dagger\mathsf{q}\langle m\rangle : r$ ($\mathsf{p} \neq \mathsf{q}$) when $\langle \mathsf{p}, \dagger, \mathsf{q}, m, r\rangle \in \mathbb{A}$.*

Traces (denoted by τ and its decorated variants) are finite sequences of actions, defined inductively from the rule $\mathcal{T} \quad ::= \quad \alpha \cdot \mathcal{T} \quad | \quad \epsilon$, where α is an action. We write $\mathbb{A}^\star$ for the set of traces.

▶ **Example 2** (Trace). We presented a trace in Section 1.

We denote $\tau_1 \cdot \tau_2$ for the concatenation of two traces. We assume an intuitive notion of the size of trace, as well as lemmas that allow us to infer that, if the size is 0, then the trace is ϵ.

2.3 Properties of Refined Traces

In this subsection, we characterise the *correctness* of traces w.r.t. refinements.

There are two conditions valid traces should verify. First, the sending/reception of messages should be consistent (as with normal MPST). Second, for every action of the trace, predicates that guard the action should hold. We call traces that satisfy message consistency *well-queued traces*, and the traces that satisfy the predicates *well-predicated traces*. In the end, we consider traces that satisfy both conditions: we call those traces *valid refined traces*.

To start with well-queued traces, we first evaluate the impact of a trace on a queue, by looking at the effect of each action on that queue (Definition 3).

▶ **Definition 3** (Trace Ending Up with Queues, well-queued traces). *A trace τ ends up with the queue w_f w.r.t. a queue w_i if:*

1. *If $\tau = \epsilon$, $w_i = w_f$; and*
2. *If $\tau = \mathsf{p}!\mathsf{q}\langle m\rangle : r \cdot \tau'$, then τ' ends up with w_f w.r.t. $\mathtt{enq}_{(\mathsf{p},\mathsf{q})}(w_i, m)$; and*
3. *If $\tau = \mathsf{p}?\mathsf{q}\langle m\rangle : r \cdot \tau'$, then τ' ends up with w_f w.r.t. $\mathtt{deq}_{(\mathsf{p},\mathsf{q})}(w_i)$ and $\mathtt{next}_{(\mathsf{p},\mathsf{q})}(w_i) = m$.*

A trace τ is well-queued *with regards to the queue w if τ ends up with the empty queue $w_\varnothing$ with respect to an initial queue w.*

A trace τ is valid if τ is well-queued with respect to the empty queue $w_\varnothing$.

▶ Remark 4. In Definition 3, we say w_i is the *initial* queue.

Regarding well-predicated traces, the idea is to record the latest value of each variable in a map; and to use that map to evaluate refinements (Definition 5).

▶ **Definition 5** (Well-Predicated Traces). *A trace τ is* well-predicated *under a map M, if either*

(i) *$\tau = \epsilon$; or*

(ii) *$\tau = \mathsf{p}\dagger\mathsf{q}\langle l, (x, c)\rangle : r \cdot \tau'$ and $M[x{\mapsto}c] \models r$ and τ' is well-predicated under $M[x{\mapsto}c]$.*

▶ **Example 6** (Well-Predicated Traces). In Section 1, we presented the trace τ:

$\mathsf{A!B}\langle \mathit{secret}, \langle n, 5\rangle\rangle : \top \cdot \mathsf{A?B}\langle \mathit{secret}, \langle n, 5\rangle\rangle : \top \cdot \mathsf{C!B}\langle \mathit{guess}, \langle x, 5\rangle\rangle : \top \cdot \mathsf{C?B}\langle \mathit{guess}, \langle x, 5\rangle\rangle : \top$

To illustrate Definition 5, we propose two actions after τ:

(i) $\tau_1 = \mathsf{B!C}\langle \mathit{more}, \langle \_, \_\rangle\rangle : x > n$; and

(ii) $\tau_2 = \mathsf{B!C}\langle \mathit{correct}, \langle \_, \_\rangle\rangle : x = n$.

We can investigate whether $\tau \cdot \tau_1$ (resp. $\tau \cdot \tau_2$) is a well-predicated trace under $M_\varnothing$. According to Definition 5, we have to investigate whether τ_1 (resp. τ_2) is well predicated under $M = \{\langle n, 5\rangle, \langle x, 5\rangle\}$.

For τ_1, according to Item ii in Definition 5, then $x > n$ must hold under M, which is not the case, therefore $\tau \cdot \tau_1$ is not well-predicated.

Regarding τ_2, according to Item ii in Definition 5, then $x = n$ must hold under M, which is the case.

Finally, we consider traces that are both valid with respect to predicates and to messages. We call those *Valid Refined Traces.* Our overall goal is to show that our framework only produces such valid refined traces.

▶ **Definition 7** (Valid Refined Traces). *A refined trace τ is* valid *if*

(i) *τ is well-queued with respect to the empty queue $w_\varnothing$; and*

(ii) *τ is well-predicated under the empty map $M_\varnothing$.*

3 Refined Communicating Automata

In this section, we model message-passing concurrent systems with refinements. We ensure that this model only generates valid refined traces (c.f. Definition 7). Our model of computation is an extension of *communicating systems* (CS) [5, 8], which are sets of Finite State Machines communicating using queues. We introduce *refined communicating systems* (RCS), a variant of CS which accounts for refinements and we show that all traces produced by RCS are valid refined traces (Theorem 18).

Refined Communicating Finite State Machines. *Communicating systems* [5] are a concurrent model of computation composed of a set of *communicating finite state machines* (CFSM) that interact with exchanges of messages. CFSM are standard finite state machines, where labels represent actions (i.e. sending or receiving messages). Individual FSM are then given a concurrent semantics, which performs messages exchanges. The state of the system is called a *configuration*, which records the state of the individual CFSMs as well as the content of the message queues. In this section, we adapt communicating systems for refinements.

First, we add refinements to the transitions of CFSM, which we call *refined* CFSM. This appears in the additional $\mathbb{R}$ in Definition 8 (we recall $\mathbb{R}$ is the set of refinements).

▶ **Definition 8** (Refined Communicating Finite State Machine (RCFSM)). *An RCFSM is a finite transition system given by $M = \langle Q, C, q_0, \mathbb{M}, \delta\rangle$, where Q is a set of states; $C = \{\mathsf{pq} \in \mathbb{P}^2 \mid \mathsf{p} \neq \mathsf{q}\}$ is a set of channels*[2]*; $q_0 \in Q$ is an initial state; $\mathbb{M}$ is a finite alphabet of messages; and $\delta \subseteq Q \times (C \times \{!, ?\} \times \mathbb{A} \times \mathbb{R}) \times Q$ is a finite set of transitions.*

We write $s \xrightarrow{\mathsf{i}\dagger\mathsf{j}\langle m\rangle : r} s'$ for $\langle s, \langle \mathsf{ij}, \dagger, m, r\rangle, s'\rangle \in \delta$. *Refined communicating systems* (RCS) are analogous to their non-refined counterparts and simply consist of a tuple of RCFSM, with one RCFSM per participant. For *refined configurations*, as with (non-refined) configurations,

[2] The original definition uses *channels*, which we do not use. We keep them for the sake of consistency.

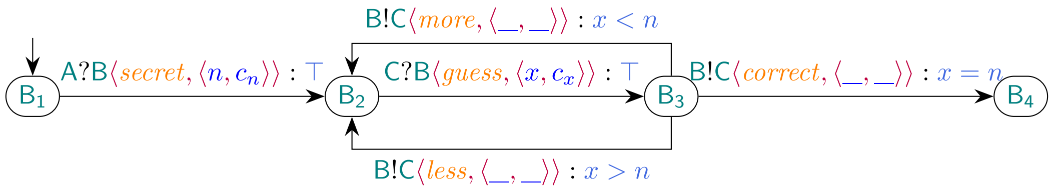

B!C⟨more, ⟨_, _⟩⟩ : x < n
A?B⟨secret, ⟨n, c_n⟩⟩ : ⊤
C?B⟨guess, ⟨x, c_x⟩⟩ : ⊤
B!C⟨correct, ⟨_, _⟩⟩ : x = n
B!C⟨less, ⟨_, _⟩⟩ : x > n
B1
B2
B3
B4

Figure 3 RCFSM of B in the $G_{\pm}$ protocol.

we store the states of the individual CFSM and the content of queues. In addition, contrary to non-refined configurations, refined configurations also contain a map in order to keep track of the values of the variables in order to be able to evaluate refinements.

▶ **Definition 9** (Refined Communicating System (RCS)). *A refined communicating system is a tuple $R = \langle M_\mathsf{p} \rangle_{\mathsf{p} \in \mathbb{P}}$ of RCFSMs such that $M_\mathsf{p} = \langle Q_\mathsf{p}, C, q_{0\mathsf{p}}, \mathbb{M}, \delta_\mathsf{p} \rangle$.*

An RCS uses one RCFSM per participant $\mathsf{i} \in \mathbb{P}$. A *configuration* represents the state of such RCS, where each participant i is in a local state s_i.

▶ **Definition 10** (Refined Configuration). *A* refined configuration *of an RCS R is a tuple S as follows: $S \overset{\text{def}}{=} \langle \langle s_1, \ldots, s_\mathsf{n} \rangle, w, M \rangle_R$ where each $s_\mathsf{i} \in Q_\mathsf{i}$, w is a queue of messages, and M is a map from variables names to values. Let $\mathbb{S}$ be the set of refined configurations.*

▶ Remark 11. Refined configurations are indexed by their RCS. This allows the configuration to store the automaton of the participant. The semantics developed below uses those (local) transitions to infer the global semantics. When the context is clear, we omit this index.

From that, we characterise *initial* and *final* configurations. We call a configuration *initial* when it is a possible configuration where no actions have been taken yet. This means that there is no pending messages (which would imply a previous *send* action), nor known variables (which would imply a previous action initialised the variable). We say a configuration is *final* when there are no pending messages (otherwise, we would expect a *receive* action to take place). Notice that it does not mean the system cannot take action at all.

▶ **Definition 12** (Initial and Final Refined Configuration). *A refined configuration $\langle \langle s_1, \ldots, s_\mathsf{n} \rangle, w, M \rangle \in \mathbb{S}$ is* initial *if and only if*
(i) *$w = w_\varnothing$;*
(ii) *$M = M_\varnothing$; and*
(iii) *each s_i is initial in the RCFSM.*

A refined configuration $S = \langle \langle s_1, \ldots, s_\mathsf{n} \rangle, w, M \rangle \in \mathbb{S}$ is final *if and only if $w = w_\varnothing$.*

▶ **Example 13** (RCS). The RCFSM of participant B in the guessing game is shown in Figure 3. See [35] for the RCFSM of A and C. Together, they form a RCS, which initial configuration is $\langle \langle \mathsf{A}_1, \mathsf{B}_1, \mathsf{C}_1 \rangle, w_\varnothing, M_\varnothing \rangle$.

Refined Semantics. We now define the semantics of RCS in Definition 14 with two reduction rules GRREC and GRSND (the initial GR stands for *global refined*, to distinguish the rules from variants in the following parts of this work), which are respectively used for receiving and sending messages. To avoid confusion with RCFSM reductions, we use a double arrow ($\Rightarrow$) to represent reductions at the refined communicating system level.

Rule GRSND specifies that, if a participant i reduces from state s_i to state s'_i while sending a message m and if the refinement predicate r attached to the action holds, then the transition is taken at the global level. In the resulting refined configuration, the message is enqueued in the relevant queue and the map of known variables M is updated to take into account the new value of the carried variable c.

Rule GRREC is similar, with the additional requirement that the message received must be the next in the participant's queue (the third premise).

Notice that the verification of refinements is *dynamic*, as it is performed by the corresponding premise in each of the rules, i.e. at execution time.

▶ **Definition 14** (Refined Global Semantics). *Given a RCS $R = \langle M_\mathsf{p} \rangle_{\mathsf{p} \in \mathbb{P}}$, we define:*

$$\textit{GRREC}\ \frac{t = s_\mathsf{i} \xrightarrow{\mathsf{j}?\mathsf{i}\langle \ell, \langle x, c\rangle\rangle : r} s'_\mathsf{i} \in \delta_\mathsf{i} \qquad M[x \mapsto c] \models r \qquad \mathtt{next}_{(\mathsf{j},\mathsf{i})}(w) = \langle \ell, \langle x, c \rangle\rangle}{\langle\langle \ldots, s_\mathsf{i}, \ldots\rangle, w, M\rangle_R \overset{t}{\Rightarrow} \langle\langle \ldots, s_\mathsf{i}{}', \ldots\rangle, \mathtt{deq}_{(\mathsf{j},\mathsf{i})}(w), M[x \mapsto c]\rangle_R}$$

$$\textit{GRSND}\ \frac{t = s_\mathsf{i} \xrightarrow{\mathsf{i}!\mathsf{j}\langle \ell, \langle x, c\rangle\rangle : r} s'_\mathsf{i} \in \delta_\mathsf{i} \qquad M[x \mapsto c] \models r}{\langle\langle \ldots, s_\mathsf{i}, \ldots\rangle, w, M\rangle_R \overset{t}{\Rightarrow} \langle\langle \ldots, s'_\mathsf{i}, \ldots\rangle, \mathtt{enq}_{(\mathsf{i},\mathsf{j})}(w, \langle \ell, \langle x, c \rangle\rangle), M[x \mapsto c]\rangle_R}$$

▶ Remark 15. Global transitions are labelled with the underlying local transition. When the local transition is not relevant, we do not show it.

▶ **Example 16** (Transitions of a RCS). Considering the RCS of $G_\pm$ (Figure 3) in its initial configuration $C_i = \langle\langle \mathsf{A}_1, \mathsf{B}_1, \mathsf{C}_1\rangle, w_\varnothing, M_\varnothing\rangle$, we have that the automaton of A can fire a transition $\mathsf{A}_1 \xrightarrow{\mathsf{A}!\mathsf{B}\langle secret, \langle n, 5\rangle\rangle : \top} \mathsf{A}_2$, and $M_\varnothing[n \mapsto 5] \models \top$, by definition of $\top$. Therefore, C_i can take a GRSND transition and reduce to $\langle\langle \mathsf{A}_2, \mathsf{B}_1, \mathsf{C}_1\rangle, w, \{\langle n, 5\rangle\}\rangle$, where w contains a single message $\langle secret, \langle n, 5\rangle\rangle$ in $w_{(\mathsf{A},\mathsf{B})}$.

If the RCS is in the configuration $C = \langle\langle \mathsf{A}_2, \mathsf{B}_3, \mathsf{C}_2\rangle, w_\varnothing, M\rangle$ with $M = \{\langle x, 5\rangle, \langle n, 5\rangle\}$, the RCFSM of participant B offers three possible transitions:

(i) $\mathsf{B}_3 \xrightarrow{\mathsf{B}!\mathsf{C}\langle more, \langle \_, \_\rangle\rangle : x<n} \mathsf{B}_2$;

(ii) $\mathsf{B}_3 \xrightarrow{\mathsf{B}!\mathsf{C}\langle less, \langle \_, \_\rangle\rangle : x>n} \mathsf{B}_2$; and

(iii) $\mathsf{B}_3 \xrightarrow{\mathsf{B}!\mathsf{C}\langle correct, \langle \_, \_\rangle\rangle : x=n} \mathsf{B}_4$.

The predicates carried in first two do not hold under M: $M \not\models x < n$ (resp. for $x > n$). Therefore, only $\mathsf{B}_3 \xrightarrow{\mathsf{B}!\mathsf{C}\langle correct, \langle \_, \_\rangle\rangle : x=n} \mathsf{B}_4$ is feasible as a GRSND transition in the RCS. As we will see below (Theorem 18), this semantics prevents invalid traces.

Trace of Refined Communicating Systems. In order to show that the semantics of RCS captures the intuition of refinements, we study the traces formed by sequences of reductions (see [35] for the formal definition of traces of RCS).

▶ **Example 17** (Trace of an RCS). The trace $\tau \cdot \tau_2$ (Example 6) is a trace of the RCS of $G_\pm$.

We conclude this section with our main result, which is that all traces produced by $\mathcal{S}(G)$ are valid refined traces. A trace is *initial* (resp. *final*) if it is obtained from a run whose first (resp. last) state is initial (resp. final).

▶ **Theorem 18** (Traces of Refined Communicating Systems are Valid Refined Traces). *For all RCS R, for all initial and final traces τ of R, τ is a valid refined trace.*

The proof is in [35].

$$\begin{array}{lllll}
\mathcal{G} & ::= & \mathsf{p} \to \mathsf{q}\{l_i(x_i : \mathcal{S} \models \mathcal{R}).\mathcal{G}\}_{i \in I} \quad | \quad \mu\mathbf{t}.\mathcal{G} & & \textit{communication, recursive type} \\
 & | & \mathbf{t} \quad | \quad \mathtt{end} & & \textit{type variable, termination} \\
\mathcal{L} & ::= & \mathsf{p}\oplus\{l_i(x_i : \mathcal{S} \models \mathcal{R}).\mathcal{L}\}_{i \in I} \quad | \quad \mathbf{t} \quad | \quad \mathtt{end} & & \textit{internal choice, type variable, termination} \\
 & | & \mathsf{p}\&\{l_i(x_i : \mathcal{S} \models \mathcal{R}).\mathcal{L}\}_{i \in I} \quad | \quad \mu\mathbf{t}.\mathcal{L} & & \textit{external choice, recursive type} \\
\mathcal{S} & ::= & \mathtt{int} \quad | \quad \ldots & & \textit{sort (payload types)}
\end{array}$$

Figure 4 Syntax of Global ($\mathcal{G}$) and Local ($\mathcal{L}$) Types and Sorts ($\mathcal{S}$).

4 Refined Multiparty Session Types (RMPST)

In the two previous sections, we introduced refinement validity and a variant of CS which is correct with respect to our validity criterion. However, working with RCS is cumbersome, in particular if we intend to prove additional properties (e.g. deadlock freedom). Fortunately, various models for message-passing concurrent computation have been developed in the literature, many of which can be encoded into CS. Multiparty session types (MPST) [38, 19, 18] is an example of such model. We focus on MPST as they have proved successful for many applications and the theory enjoy many useful properties (e.g. session fidelity, deadlock freedom, liveness etc). However, MPST is not the only possible choice, and we sketch different input models in [35]. In this section, we introduce *refined multiparty session types* (RMPST), which are an extension of MPST annotated with refinement predicates and we show how one can extend existing models to easily obtain refinements.

In Section 4.1, we first present the syntax of *global* and *local* refined multiparty session types, adapted for refinements. In Section 4.2, we present how to obtain RCS from local RMPST, extending a standard approach to implement MPST in CS [12] with refinements.

4.1 Syntax of RMPST

We define the syntax of RMPST. First we assume that messages carry different sorts of payload. As a reminder, for simplicity, in our examples, we only consider `int` payloads. Also, we recall the conventions from Section 2.1: $\mathbb{P}$ is the set of participants and $\mathbb{L}$ is the set of labels. For recursion, we introduce type variables that range over $\{\mathbf{T}, \mathbf{U}, \ldots\}$; $\mathbf{t}$ is a meta-variable taken over the set of type variables. We assume all type variables appearing in a type are distinct and we do not (syntactically) distinguish global and local type variables. Finally, x_i are meta-variables over payload variables taken from the set $\mathbb{V}$.

We first define *global refined multiparty session types*, which are inductive data types generated by the production $\mathcal{G}$ in Figure 4. The type $\mathsf{A} \to \mathsf{B}\{l_i(x_i : \mathsf{S}_i \models r_i).G_i\}_{i \in I}$ describes a protocol where A chooses a label l_i amongst possible I and sends a message to B. The message contains a payload of type $\mathtt{S_i}$, which is bound to x_i when sent. *Refinement predicates* we introduce guard the communication they are attached to, meaning the system can select a choice with predicate r_i only if r_i holds. In that case, the message is sent and the protocol continues with its continuation of type G_i. $\mu\mathbf{T}.G$ binds $\mathbf{T}$ in G, and a bound type variable $\mathbf{T}$ in a type denotes a protocol recursion. Let $\mathtt{frv}(G)$ denotes the free recursion variables occuring in G. Finally $\mathtt{end}$ describes a terminated protocol. Let $\mathtt{parts}(G)$ be the set of participants that appear in G (c.f. [35] for the definition of $\mathtt{parts}(G)$). We write $\mathsf{p} \in G$ for $\mathsf{p} \in \mathtt{parts}(G)$.

▶ **Example 19** (Refined Global Multiparty Session Type)**.** The type $G_\pm$ presented in Section 1 is a refined global type; we have $\mathtt{parts}(G_\pm) = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$.

To characterise the behaviour of individual participants, we define *refined local multiparty session types*, which are inductive datatypes generated by $\mathcal{L}$ in Figure 4. Recursion, type variables and termination are similar in local and global types. Only the communication specifications differs: in a local type $\mathsf{p}\oplus\{l_i(x_i : \mathsf{S}_i \models r_i).L_i\}_{i\in I}$ describes an *internal choice*, i.e. the participant chooses a label l_i and sends it to p. Conversely, $\mathsf{p}\&\{l_i(x_i : \mathsf{S}_i \models r_i).L_i\}_{i\in I}$ describes an *external choice*: p makes a choice amongst the possible l_i and the local participant *receives* this choice.

Global and local MPST are related: we can *project* a global type onto the local types of its participants. Below, we define a *projection* (partial) operator $G\restriction_{\mathsf{p}}$, which returns the local type of p with respect to the global type G.

We define a projection with a *merge* (partial) operator, which merges multiple local types of a participant into a single local type. This is used to merge the (possibly different) types of the continuations present in the communication branches. The study of different variants of merge operators is an active field (e.g. [32, Section 3]). For the sake of simplicity, in this paper we use a naïve merge operator, which simply ensures that all types are the same.

▶ **Definition 20** (Projection). *Given* p, q *and* r *three distinct participants:*

$$\mathsf{p} \to \mathsf{q}\{l_i(x_i : \mathsf{S}_i \models R_i).G_i\}_{i\in I}\restriction_{\mathsf{p}} = \mathsf{q}\oplus\{l_i(x_i : \mathsf{S}_i \models R_i).G_i\restriction_{\mathsf{p}}\}_{i\in I}$$
$$\mathsf{q} \to \mathsf{p}\{l_i(x_i : \mathsf{S}_i \models R_i).G_i\}_{i\in I}\restriction_{\mathsf{p}} = \mathsf{q}\&\{l_i(x_i : \mathsf{S}_i \models \top).G_i\restriction_{\mathsf{p}}\}_{i\in I}$$
$$\mathsf{q} \to \mathsf{r}\{l_i(x_i : \mathsf{S}_i \models R_i).G_i\}_{i\in I}\restriction_{\mathsf{p}} = \sqcap_{i\in I}(G_i\restriction_{\mathsf{p}})$$

$$\mu\mathbf{t}.G'\restriction_{\mathsf{p}} = \begin{cases} \mu\mathbf{t}.(G'\restriction_{\mathsf{p}}) & \textit{if } \mathsf{p} \in G' \textit{ or } \mathsf{frv}(G') \neq \varnothing \\ \mathit{end} & \textit{otherwise} \end{cases} \qquad \mathbf{t}\restriction_{\mathsf{p}} = \mathbf{t}\,\mathit{end}\restriction_{\mathsf{p}} = \mathit{end}$$

where a merge *operator is defined as:* $\sqcap_{i\in I}L_i \overset{\text{def}}{=} L$ *if* $\forall i \in I \cdot L = L_i$, *undefined otherwise.*

Notice that our local RMPST accept refinements on both receiving and sending, and the semantics developed in Section 3 accept any position for verification. When projecting a global type $G = \mathsf{A} \to \mathsf{B}\{\ell(x : \mathtt{int} \models r).\mathtt{end}\}$ onto local types, we therefore have a choice to project the refinement:

- on the send side: $G\restriction_{\mathsf{A}} = \mathsf{B}\oplus\{\ell(x : \mathtt{int} \models r).\mathtt{end}\}$ and $G\restriction_{\mathsf{B}} = \mathsf{A}\&\{\ell(x : \mathtt{int} \models \top).\mathtt{end}\}$
- on the receive side: $G\restriction_{\mathsf{A}} = \mathsf{B}\oplus\{\ell(x : \mathtt{int} \models \top).\mathtt{end}\}$ and $G\restriction_{\mathsf{B}} = \mathsf{A}\&\{\ell(x : \mathtt{int} \models r).\mathtt{end}\}$
- or a combination of both[3].

Our projection takes the first option, i.e. refinements are checked when the message is emitted, but with any of these choices, our developments would not substantially change.

▶ **Example 21** (Projection). We project $G_\pm$ (Section 1) onto participants A and B[4]:

$$G_\pm\restriction_{\mathsf{A}} = \mathsf{B}\oplus\{\mathit{secret}(n : \mathtt{int} \models \top).\mathtt{end}\}$$
$$G_\pm\restriction_{\mathsf{B}} =$$
$$\mathsf{A}\&\left\{\mathit{secret}(n : \mathtt{int} \models \top).\mu\mathbf{T}.\mathsf{C}\&\left\{\mathit{guess}(x : \mathtt{int} \models \top).\mathsf{C}\oplus\left\{\begin{array}{l}\mathit{more}(\models x < n).\mathbf{T},\\ \mathit{less}(\models x > n).\mathbf{T},\\ \mathit{correct}(\models x = n).\mathtt{end}\end{array}\right\}\right\}\right\}$$

[3] For instance, if we want to implement a centralised server that communicates with (isolated) clients, we may want all refinements to be asserted by the server, independently of the direction.

[4] The projection onto C is similar to the recursive part of the projection onto B, with ! and ? swapped.

4.2 From Refined MPST to Refined Communicating System

In this subsection, we show how to generate an RCS from local RMPSTs. As shown in Definition 20, local types are projected from global multiparty session types. Therefore, this step allows us to complete the conversion from a global RMPST into an RCS. We adapt the translation from local type to CFSMs presented in [13] to accommodate refinements in types.

The intuition behind the translation is to decompose a local type into the individual steps it specifies. For this, we first need to retrieve all those steps. We define the set of types that occur nested in another type: a type T' *occurs* in a type T (noted $T' \in T$) if it appears in the continuations of T after one or multiple steps (see [35]).

Given this, we can proceed to the translation itself, in Definition 22. This definition says that the states of the RCFSM of a local type T_0 are composed of the (sub)types that appear in T_0, stripped of the leading $\mu\mathbf{t}.$ (the function $\mathtt{strip}$ removes the leading recursions variables; this formalises [13, Item (2) in Definition 3.4]) and of recursive variables $\mathbf{t}$. We define the set of transitions of this RCFSM by taking the action each type (i.e. each state) can take, and adding a transition with this action from the initial state to the continuation (stripped of leading $\mu\mathbf{t}.$). In the case that the continuation is a recursion variable $\mathbf{t}$, we have to search in the original type the continuation. Compared to [13, Item (2) in Definition 3.4], we simply add the support for the refinements predicates, which appear both in the types (i.e. in the states) and in the actions (i.e. in the transitions).

▶ **Definition 22** (RCFSM of Refined Local Types (extends Definition 3.5 in [13])). *Given a global type G, the RCFSM of participant p (with local type $T_0 = G\!\upharpoonright_{\mathsf{p}}$) is the automaton $\mathcal{A}(T_0) = \langle Q, C, \mathtt{strip}(T_0), \mathbb{M}, \delta\rangle$ where:*

- $Q = \{T' \mid T' \in T_0 \wedge T' \neq \mathbf{t} \wedge T' \neq \mu\mathbf{t}.T_\mu\}$;
- $C = \{\mathsf{pq} \mid \mathsf{p}, \mathsf{q} \in G, \mathsf{p} \neq \mathsf{q}\}$; *and*
- δ *is the smallest set of transitions such that: for all $T \in T_0$ in Q, for all $c \in \mathbb{C}$:*
  - *if T is $\mathsf{q}\oplus\{\ell_i(x_i : \mathsf{S}_i \models r_i).T_i\}_{i\in I}$, for all T_i:*
    - *if $T_i \neq \mathbf{t}$, then $\langle T, \mathsf{p}!\mathsf{q}\langle \ell_i, \langle x, c\rangle\rangle : r, \mathtt{strip}(T_i)\rangle \in \delta$*
    - *if $T_i = \mathbf{t}$ with $\mu\mathbf{t}.T' \in T_0$, then $\langle T, \mathsf{p}!\mathsf{q}\langle \ell_i, \langle x, c\rangle\rangle : r, \mathtt{strip}(T')\rangle \in \delta$*
  - *if T is $\mathsf{q}\&\{\ell_i(x_i : \mathsf{S}_i \models r_i).T_i\}_{i\in I}$, for all T_i:*
    - *if $T_i \neq \mathbf{t}$, then $\langle T, \mathsf{q}?\mathsf{p}\langle \ell_i, \langle x, c\rangle\rangle : r, \mathtt{strip}(T_i)\rangle \in \delta$*
    - *if $T_i = \mathbf{t}$ with $\mu\mathbf{t}.T' \in T_0$, then $\langle T, \mathsf{q}?\mathsf{p}\langle \ell_i, \langle x, c\rangle\rangle : r, \mathtt{strip}(T')\rangle \in \delta$*

where $\mathtt{strip}(T) \overset{\text{def}}{=} \mathtt{strip}(T')$ if $T = \mu\mathbf{t}.T'$; and $\mathtt{strip}(T) \overset{\text{def}}{=} T$ otherwise.

Finally, we define the *RCS of a type*.

▶ **Definition 23** (Refined Communicating System of a Type). *The RCS of a type G, noted $\mathcal{S}(G)$, is a tuple composed of the RCFSM of all participants $\mathcal{S}(G) \overset{\text{def}}{=} \langle \mathcal{A}(G\!\upharpoonright_{\mathsf{p}})\rangle_{\mathsf{p}\in\mathtt{parts}(G)}$. We note $\mathcal{C}(G)$ the initial configuration of $\mathcal{S}(G)$.*

▶ **Example 24** (Refined Communicating System of $G_\pm$). The communicating system of $G_\pm$ is $\mathcal{S}(G_\pm) = \langle \mathcal{A}(G_\pm\!\upharpoonright_{\mathsf{A}}), \mathcal{A}(G_\pm\!\upharpoonright_{\mathsf{B}}), \mathcal{A}(G_\pm\!\upharpoonright_{\mathsf{C}})\rangle$.

The initial configuration $\mathcal{C}(G_\pm)$ of this RCS $\mathcal{S}(G_\pm)$ is $\langle\langle \mathsf{A}_1, \mathsf{B}_1, \mathsf{C}_1\rangle, w_\varnothing, M_\varnothing\rangle$.

Theorem 18 applies to RCS obtained from RMPST: RCS generated from Definition 23 only produce valid refined traces, with the refined global semantics presented in Definition 14. Notice also that, if refinements always hold, RMPST and their semantics coincide with the semantics presented in [12].

5 Decentralised Refined Multiparty Session Types

In the previous section, we presented RCS and we showed that every trace of an RCS is a valid refined trace. However, RCS are theoretical constructions and are not intended to be implemented directly, as they use a global shared map of variables. In practice, a user may want to develop more precise analysis techniques on specific classes of RCS to remove this global map, which allows a decentralised verification of refinements, while keeping the validity of refined traces.

The goal of this section is twofold: on the one hand, the decentralised semantics we develop serves as a theoretical background for our implementation (Section 7). On the other hand, it illustrates the modularity of our framework. We show that the decentralised approach produces valid refined traces by showing refined configurations we developed in Section 3 simulate decentralised systems. This approach is not specific to our variant: we expect other optimisations presented in the literature could be integrated similarly.

This section is divided in the following steps: first, we define what we mean by *decentralised verification of the refinements*, by adapting the semantics of RCS (Definitions 25 and 28). We split the global map of variables' values into local maps (one per participant). Then, we show that despite being modified, the new variant still produces valid refined traces (Definition 7). We justify this claim by proving that under some conditions, the original RCS *simulates* (c.f. [30, Exercise 1.4.17, p. 26]) the decentralised variant (Theorem 31). Since trace equivalence is coarser than simulation, this is sufficient to prove that decentralised configurations that meet the said conditions produce valid refined traces.

The conditions we mentioned above are:

(i) variables should not be duplicated; and

(ii) when evaluating a predicate, the free variables of the predicate must be in the local map.

Without the first condition, we can possibly have two distinct values assigned to the same variable without being able to distinguish which is the most recent. The second condition is required to verify the refinements locally (e.g. predicates that constraint an action of A should be checked by A itself). To illustrate the importance of the first condition, consider the type $\mathsf{A} \to \mathsf{B}\,\{\ell_1(x : \mathtt{int}).\mathsf{C} \to \mathsf{D}\,\{\ell_2(x : \mathtt{int}).\mathtt{end}\}\}$. In the centralised approach, x is aliased, while in the decentralised approach, the x exchanged between A and B is stored in a local map, and the x exchanged between C and D is stored in another local map; both are not aliased. To prevent different semantics, we need to prevent such difference, which is the goal of the first condition.

Decentralised Configurations and Decentralised Semantics. First, we define *decentralised* configurations in Definition 25. Compared to Definition 10, instead of a global map in the tuple, we associate a local map to each automata state. Those maps store the variables each participant has access to.

▶ **Definition 25** (Decentralised Configuration). *A* Decentralised Configuration *of an RCS* $\mathcal{S}(G) = \langle\langle Q_{\mathsf{i}}, C_{\mathsf{i}}, q_{0,\mathsf{i}}, \mathbb{A}, \delta_{\mathsf{i}}\rangle\rangle_{\mathsf{i}\in\mathtt{parts}(G)}$ *is a tuple* $\langle\langle\langle s_1, M_1\rangle, \ldots, \langle s_{\mathsf{n}}, M_{\mathsf{n}}\rangle\rangle, w\rangle_{\mathcal{S}(G)}$ *where each* $s_{\mathsf{i}} \in Q_{\mathsf{i}}$*, each* M_{i} *is a local map of variables to values, and* w *is a queue of messages.*

Let $\mathbb{S}_D$ *be the set of decentralised configurations. We note* $\mathcal{D}(G)$ *the initial decentralised configuration of* $\mathcal{S}(G)$*.*

Remark 11 also applies for decentralised configurations.

▶ **Example 26** (Initial decentralised configuration of $G_{\pm}$)**.** In Example 13, we presented the refined communicating system of $G_{\pm}$ and its associated refined configuration. The *initial decentralised configuration* of this system is $\langle\langle \mathsf{A}_1, M_{\varnothing}\rangle, \langle \mathsf{B}_1, M_{\varnothing}\rangle, \langle \mathsf{C}_1, M_{\varnothing}\rangle, w_{\varnothing}\rangle$. In particular, notice that it uses the same set of refined CFSM than the refined configuration.

The global reduction rules are adapted accordingly: in the rules DREC and DSND ("D" stands for "decentralised"), when a message is sent or received, the corresponding local map is updated, instead of a global map as in GRREC and GRSND.

▶ Remark 27. Contrary to Definition 14, when a variable is sent, it is removed from the local map of variables. Intuitively, when a participant sends a variable, it erases its knowledge of it, to prevent aliasing issues. A direct consequence of this is that, in the centralised implementation, the global map of variables is a *superset* of the local maps in the corresponding decentralised implementation. Indeed, while a variable is in transit, it appears neither in the sender's map, nor in the receiver's one. This observation will be proved together with the simulation proof (Theorem 31).

▶ **Definition 28** (Decentralised Global Semantics)**.** *Given an RCS* $R = \langle M_{\mathsf{p}}\rangle_{\mathsf{p}\in\mathbb{P}}$

$$\textit{DREC}\ \frac{t = s_{\mathsf{i}} \xrightarrow{\mathsf{j}?\mathsf{i}\langle \ell, \langle x, c\rangle\rangle : r} s'_{\mathsf{i}} \in \delta_{\mathsf{i}} \qquad \mathtt{next}_{(\mathsf{j},\mathsf{i})}(w) = \langle \ell, \langle x, c\rangle\rangle \qquad M_{\mathsf{i}}[x \mapsto c] \models r}{\langle\langle \ldots, \langle s_{\mathsf{i}}, M_{\mathsf{i}}\rangle, \ldots\rangle, w\rangle_R \overset{t}{\Rightarrow} \langle\langle \ldots, \langle s_{\mathsf{i}}, M_{\mathsf{i}}[x \mapsto c]\rangle, \ldots\rangle, \mathtt{deq}_{(\mathsf{j},\mathsf{i})}(w)\rangle_R}$$

$$\textit{DSND}\ \frac{t = s_{\mathsf{i}} \xrightarrow{\mathsf{i}!\mathsf{j}\langle \ell, \langle x, c\rangle\rangle : r} s'_{\mathsf{i}} \in \delta_{\mathsf{i}} \qquad M_{\mathsf{i}}[x \mapsto c] \models r}{\langle\langle \ldots, \langle s_{\mathsf{i}}, M_{\mathsf{i}}\rangle, \ldots\rangle, w\rangle_R \overset{t}{\Rightarrow} \langle\langle \ldots, \langle s_{\mathsf{i}}, M_{\mathsf{i}} \backslash x\rangle, \ldots\rangle, \mathtt{enq}_{(\mathsf{i},\mathsf{j})}(w, \langle \ell, \langle x, c\rangle\rangle)\rangle_R}$$

Conditions for Decentralised Verification and Correctness Proofs. We now focus on proving that this decentralised semantics produces valid refined traces. As we mentioned above, this holds under two conditions, which we define first:

▶ **Definition 29** (Conditions for Decentralised Verification Simulation)**.** *Given a decentralised configuration* $\langle\langle\langle s_{\mathsf{i}}, M_{\mathsf{i}}\rangle, \ldots\rangle, w\rangle$, *the* conditions for simulation *are:*

1. *No duplication:*
   a. *if* $\exists M_{\mathsf{i}} \cdot x \in \mathtt{dom}(M_{\mathsf{i}})$, *then* $\forall \mathsf{i}, \mathsf{j} \cdot x \notin w_{(\mathsf{i},\mathsf{j})}$ *and* $\forall \mathsf{j} \neq \mathsf{i} \cdot x \notin \mathtt{dom}(M_{\mathsf{j}})$.
   b. *if* $\exists \langle \mathsf{i}, \mathsf{j}\rangle \cdot x \in w_{(\mathsf{i},\mathsf{j})}$, *then* $\forall \mathsf{i} \cdot x \notin \mathtt{dom}(M_{\mathsf{i}})$ *and* $\forall \langle \mathsf{i}', \mathsf{j}'\rangle \neq \langle \mathsf{i}, \mathsf{j}\rangle \cdot x \notin w_{(\mathsf{i}',\mathsf{j}')}$.
2. *Free variables are in the map:* $\forall \mathsf{i} \cdot \forall s_{\mathsf{i}}' \cdot s_{\mathsf{i}} \xrightarrow{\mathsf{i}\dagger\mathsf{j}\langle \ell, \langle x, c\rangle\rangle : r} s'_{\mathsf{i}} \cdot \mathsf{fv}(r) \subseteq \mathtt{dom}(M_{\mathsf{i}}[x \mapsto c])$

▶ **Definition 30** (Decentralisable Type)**.** *A type* G *is* decentralisable *if the two conditions hold for all reachable decentralised configurations from* $\mathcal{D}(G)$.

Notice that the second condition is redundant, as the condition $M_{\mathsf{i}}[x \mapsto c] \models r$ (in the premises of the reduction rules) already requires that $\mathsf{fv}(r)$ is a subset of the variables in $M_{\mathsf{i}}[x \mapsto c]$. Even without making this condition explicit, the system would stall if a predicate cannot be verified. For the sake of clarity, we keep it explicit in the two required conditions.

We now observe a correspondence between the (centralised) refined configuration and the decentralised configuration of a global type G. To characterise the correspondence between centralised and decentralised configuration, we establish a *simulation* relation between the two (see [35] and [30]). Intuitively, a simulation captures the fact that one system (the centralised configuration in our case) can mimic all transitions of another system (the decentralised one here).

We can now prove the main result of this section, which is that the decentralised semantics does not induce new (unwanted) behaviours, i.e. all decentralised transitions can be mimicked by centralised transitions, i.e. the centralised approach simulates the decentralised one.

▶ **Theorem 31** (Centralised simulates Decentralised). *For all decentralisable RMPST G (Definition 30), $\mathcal{C}(G)$ simulates $\mathcal{D}(G)$.*

Proof. The proof is available in [35]. ◀

This result shows that any type that verifies the conditions stated in Definition 29 can be verified in a decentralised way. The difficulty is that the conditions are about the execution: we do not know whether a predicate will have a missing variable during the execution. With a knowledge flow algorithm, we can infer (from the communication specifications in the global type) which participant has access to which variables at any point in the execution of the protocol, i.e. we can *localise* each variable throughout the execution of the protocol. This algorithm (which we present in [35]) does not present major challenges.

Notice that the reverse simulation does not hold: $\mathcal{D}(G)$ does not simulate $\mathcal{C}(G)$. Indeed, $\mathcal{C}(G)$ can verify a predicate whose variables are spread over different participants, i.e. where variables would be spread across multiple M_i in the decentralised variant.

6 Static Elision of Redundant Refinements

In this section, we present a second optimisation, which is complimentary from the first one. The main idea is to statically analyse a given protocol to find and remove redundant refinements. Our approach is to consider a *target* transition, which we aim to remove the refinement, if possible. Our optimisation can then be applied successively to different target transitions one after each other. For instance, consider the following protocol G_s. We target the second refinement, $x < 10$, which necessarily holds if the first one does (since x does not change). Therefore it is redundant and can be removed.

$$G_s = \mathsf{A} \to \mathsf{B}\,\{\ell_1(x : \mathtt{int} \models x < 0).\mathsf{A} \to \mathsf{B}\,\{\ell_2(y : \mathtt{int} \models x < 10).\mathtt{end}\}\}$$

However, removing refinements is not always trivial, since the communication semantics is asynchronous. Consider for instance the following type:

$$\mathsf{A} \to \mathsf{C}\,\{\ell_1(x : \mathtt{int} \models x > 20).\mathsf{A} \to \mathsf{B}\,\{\ell_2(x : \mathtt{int} \models x < 0).\mathsf{C} \to \mathsf{B}\,\{\ell_3(y : \mathtt{int} \models x < 10).\mathtt{end}\}\}\}$$

A naïve approach would be to remove the refinement of the last communication ($x < 10$), since the previous communication has a stronger guarantee ($x < 0$). However, due to the asynchrony of communications, the second and third communications could be swapped at runtime, but the refinement ($x < 10$) does not hold before the action $\mathsf{A} \to \mathsf{B}\,\{\ell_2(x : \mathtt{int} \models x < 0)\ldots\}$ occurs. Therefore, in this case, removing the last refinement is incorrect. The optimisation we present takes into account those cases, by keeping track of causal relations between actions.

This section is independent of the previous one, although this second optimisation can help to make some protocols localisable: for instance, G_s above is not localisable. Since the second step $\mathsf{A} \to \mathsf{B}\,\{\ell_2(y : \mathtt{int} \models x < 10).\mathtt{end}\}$ requires A to access x, which is at B. However, once removed, the protocol becomes localisable, and can therefore be decentralised, helping the first optimisation introduced in Section 5.

As with the previous section, the optimisation we present could easily be further improved. Here, we focus on a simple case, as our goal is not to discuss the optimisation itself, but rather to show the versatility of the framework.

We present this section in two steps: first, in Section 6.1, we focus on RCS, which form the core of our framework; then, in Section 6.2, we apply the above result to RMPST.

$$\begin{tikzcd}
\arrow[d] & & \\
\mathsf{A}_1 \arrow[r, "\mathsf{A!B}\langle \ell_1, \langle x, \_\rangle\rangle : x < 0"] & \mathsf{A}_2 \arrow[r, "\mathsf{A!B}\langle \ell_2, \langle y, \_\rangle\rangle : x < 10"] & \mathsf{A}_3
\end{tikzcd}$$

(a) RCFSM of A in the protocol G_s ($\mathcal{A}(G_s\restriction_{\mathsf{A}})$).

$$\begin{tikzcd}
\arrow[d] & & \\
\mathsf{B}_1 \arrow[r, "\mathsf{A?B}\langle \ell_1, \langle x, \_\rangle\rangle : \top"] & \mathsf{B}_2 \arrow[r, "\mathsf{A?B}\langle \ell_2, \langle y, \_\rangle\rangle : \top"] & \mathsf{B}_3
\end{tikzcd}$$

(b) RCFSM of B in the protocol G_s ($\mathcal{A}(G_s\restriction_{\mathsf{B}})$).

Figure 5 RCFSM of the RCS of G_s, the running example of Section 6.

6.1 Static Elision of Refinements in RCS

In a first step, we develop and prove the correctness of our analysis in RCS. The question is therefore whether, given a RCS R with one CFSM containing a transition with refinement r, this RCS R is equivalent (bisimilar) to an RCS where r is replaced with $\top$.

For the sake of simplicity, in this subsection, we'll explain the static elision of refinements in RCS using examples. Formal definitions, lemmas and their proofs are available in [35]. We use the RCS of G_s shown in Figure 5.

If we aim to i.e. transitions which payload modify variables that do not appear free in the refinement of the considered transition.

▶ **Example 32** (Independent transitions)**.** In $\mathcal{S}(G_s)$, $\mathsf{A}_2 \xrightarrow{\mathsf{A!B}\langle \ell_2,\langle y,\_\rangle\rangle : x<10} \mathsf{A}_3$ depends on the variable $x \in \mathsf{fv}(x < 10)$. This transition is self-independent. Since the payload of $\mathsf{A}_1 \xrightarrow{\mathsf{A!B}\langle \ell_1,\langle x,\_\rangle\rangle : x<0} \mathsf{A}_2$ is x, the former transition depends on the later.

▶ Remark 33. We note $\mathbb{T}_x$ the set of transitions $\sigma \xrightarrow{\_\dagger\_\langle\_,\langle x,\_\rangle\rangle : \_} \sigma'$. Given a transition t with refinement r, if $x \in \mathsf{fv}(r)$, then t depends on all transitions of $\mathbb{T}_x$.

Essentially, when attempting to remove a refinement from a target transition t, we can disregard all transitions t is independent of.

The second definition we will need is about transitions being *well-defined*. So far, nothing prevents us to use refinements with undefined free variables, we simply consider the refinement does not hold (c.f. Definition 14). In this section, we specifically focus on systems where free variables of refinements are in the map when the refinement is evaluated. When it is the case, we call transitions with such refinements *well-defined*.

▶ **Example 34** (Well-defined transition)**.** Considering the RCS in Figure 5. In the RCFSM of A, the (local) state A_2 is only accessible with a transition $\mathsf{A}_1 \xrightarrow{\mathsf{A!B}\langle \ell_1,\langle x,\_\rangle\rangle : x<0} \mathsf{A}_2$. Therefore, any global state $\langle\langle \mathsf{A}_2, \mathsf{B}_{\{1,2,3\}}\rangle, \_, M\rangle$ necessarily contains a preceding transition $\mathsf{A}_1 \xrightarrow{\mathsf{A!B}\langle \ell_1,\langle x,\_\rangle\rangle : x<0} \mathsf{A}_2$. Therefore, x is always in the map M of that state.

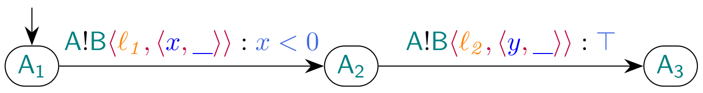
Therefore, the transition $\mathsf{A}_2 \xrightarrow{\mathsf{A!B}\langle \ell_2,\langle y,\_\rangle\rangle : x<10} \mathsf{A}_3$ is well-defined.

We can now conclude our analysis technique: consider a target transition t with refinement r that is self-independent (it does not modify the variables of its refinement) and well-define. If all transitions that modify the free variables of r can guarantee (via their refinement) that the modification they do is correct with respect to r, then we can safely remove r.

▶ **Theorem 35** (Correctness of refinement elision)**.** *Given an RCS R containing an RCFSM $M = \langle Q, C, q_0, \mathbb{A}, \delta\rangle$, and $t = s_{\mathsf{i}} \xrightarrow{\mathsf{p}\dagger\mathsf{q}\langle m\rangle : r} s_{\mathsf{i}}{}' \in \delta$, a well-defined self-independent transition. Let $t' = s_{\mathsf{i}} \xrightarrow{\mathsf{p}\dagger\mathsf{q}\langle m\rangle : \top} s_{\mathsf{i}}{}'$; $\delta' = \delta \setminus \{t\} \cup t'$; $M' = \langle Q, C, q_0, \mathbb{A}, \delta'\rangle$; and R' be R where M is replaced with M'. If, for each transition $t_w = \_ \xrightarrow{\_!\_\langle\_\rangle : r_w} \_$ in $\bigcup_{x \in \mathsf{fv}(r)} \mathbb{T}_x$, for all map M, $M \models r_w$ entails $M \models r$, then there exists a bisimulation relating the states of R' and R.*

Proof. Proving each direction of the bisimulation is direct (see the proof in [35]). ◀

▶ **Example 36** (Application of Theorem 35)**.** The following RCFSM, where $x < 10$ is removed, is a valid replacement for $\mathcal{A}(G_s\upharpoonright_{\mathsf{A}})$ in $\mathcal{S}(G_s)$.


A_1
$\mathsf{A}!\mathsf{B}\langle \ell_1, \langle x, \_\rangle\rangle : x < 0$
A_2
$\mathsf{A}!\mathsf{B}\langle \ell_2, \langle y, \_\rangle\rangle : \top$
A_3

6.2 Application to RMPST Protocols

The above subsection explains how to remove some redundant refinements in RCS. In this subsection, we intend to do the same, focusing on RMPST instead of RCS.

Our goal is the following: we are given an RMPST G, and we would like to remove one of its refinement (which we call the *target* refinement r). For the sake of simplicity, in this section, we assume all labels are uniquely used. For the general case, we can simply uniquely rename redundant labels. Overall, the roadmap for this subsection is to show that given the type G', which is G where r is *replaced* by $\top$, G and G' behave similarly, i.e. the RCS the generate are bisimilar. To achieve this, we show that Theorem 35 applies to $\mathcal{S}(G)$ and $\mathcal{S}(G')$. Therefore, the main point is finding conditions on RMPST that ensures hypothesis of Theorem 35 holds; we have to verify the following items:

1. all transitions our refinement depends on should entail the refinement itself;
2. the transition that carries the refinement must be well-defined. Since variables cannot be removed from the map, the first occurrence of the target transition must respect the domain condition. Therefore, for this step, we can ignore recursion.

The main difference with automata is that, in types, we have *communications*, which possibly contains choices with multiple branches; and we our goal is to remove the refinement of one of those branches. Therefore, we first introduce *steps* of a communication, i.e. given a choice, what are the possible choices it can take. We then extend this to types. We show that steps in a type correspond to transitions in the automata of that type.

▶ **Example 37** (Step)**.** The type $G_y = \mathsf{A} \to \mathsf{B}\,\{\ell_2(y : \mathtt{int} \models x < 10).\mathtt{end}\}$ has the step $\mathsf{A} \to \mathsf{B}\langle \ell_2, y\rangle \models x < 10$. Since G_y occurs in one of the branches of G_s (from the introduction of this section), this step *occurs* in G_s.

Given this notion of steps occurring in a type that is analogous to transitions in the RCFSM of that type, we can now focus on the conditions of Theorem 35. Therefore, we have to characterise what corresponds to *well-defined transitions* in a type. Since transitions (in RCS) and steps (in types) are analogous, we introduce *well-define steps* in a type. We recall that, in a RCS, a transition is well-defined if the free variables of the refinement it carries are always known when the transition is fired. Since variables are never removed from the map, we can focus on the first occurrence of the transition. So far, we do not have a notion of *run* for a type. Therefore, we first define an *happens-before* relation in RMPST, and we use this relation to define *well-defined steps* as steps that contain a refinement which free variable are all exchanged in a communication that *happens-before* the step we consider. With those two definitions, we can finally prove that a well-defined step in a type corresponds to a well-defined transition in the corresponding RCS.

▶ **Example 38** (Well-define step in a type)**.** Consider G_s and G_y as in Example 37. The step $\mathsf{A} \to \mathsf{B}\langle \ell_2, y\rangle \models x < 10$ is well-defined. Indeed, $\mathsf{fv}(x < 10) = \{x\}$, $G_s < G_y$, and G_s contains a branch that sends x and which continuation contains G_y.

We can finally proceed to the overall goal of this section: showing that the type with and without the target refinement behave similarly. Thanks to the above lemmata, we simply have to target a refinement with the appropriate conditions and apply Theorem 35.

▶ **Theorem 39** (Static elision of redundant refinements in types). *Given two a global types* G *and* $G_s = \mathsf{p} \to \mathsf{q}\{\ell_i(x_i : \mathsf{S}_i \models r_i).G_i\}_{i \in I} \in G$*, such that, for one* $t \in I$*,* $\mathsf{p} \to \mathsf{q}\langle \ell_t, x_t\rangle \models r_t$ *is a well-defined step with* $x_t \notin \mathsf{fv}(r_t)$*. Let* $\ell_{t'} = \ell_t$*,* $x_{t'} = x_t$*,* $\mathsf{S}_{t'} = \mathsf{S}_t$*,* $r'_t = \top$*,* $G_{t'} = G_t$*,* $G_{s'} = \mathsf{p} \to \mathsf{q}\{\ell_i(x_i : \mathsf{S}_i \models r_i).G_i\}_{i \in I \setminus \{t\} \cup \{t'\}}$*; and* G' *be* G *where* G_s *is replaced with* $G_{s'}$*. If, for all steps,* $\mathsf{r} \to \mathsf{s}\langle \_, x_w\rangle \models r_w$ *occurring in* G *(for each* $x \in \mathsf{fv}(r)$*),* $M \models r_w$ *entails* $M \models r$ *(for all* M*), there exists a bisimulation between the states of* $\mathcal{S}(G)$ *and those of* $\mathcal{S}(G')$*.*

Proof. We prove this by showing that Theorem 35 applies to $\mathcal{S}(G)$ and $\mathcal{S}(G')$. The proof is provided [35]. ◀

▶ **Example 40** (Application of Theorem 39). Given G_s as in Example 37 and G'_s as follows (notice the second refinement is replaced by $\top$), G_s and the following G'_s have the same behaviour:

$$G'_s = \mathsf{A} \to \mathsf{B}\,\{\ell_1(x : \mathtt{int} \models x < 0).\mathsf{A} \to \mathsf{B}\,\{\ell_2(y : \mathtt{int} \models \top).\mathtt{end}\}\}$$

7 Implementation

In the previous section, we introduced an instance of our framework: a system that accommodates refinements using a decentralised verification mechanism. In this section, we follow up on this example with an implementation, based on Rumpsteak, of this system.

Rumpsteak [7] is a framework to write Rust programs according to an MPST specification. The framework is divided into two parts:

(i) a runtime library that provides primitives to write asynchronous programs in Rust; and

(ii) a tool (`rumpsteak-generate`) to generate skeleton Rust files from specification files (i.e. from global types), in two steps.

Working with Rumpsteak takes two manual steps. The user specifies (step 1) the protocol in a global type(written as Scribble files [39], see Figure 6a). This global type is automatically projected using νScr [15] and the projected types are used to generate skeleton Rust files (see Figure 6b). The generated Rust code contains Rust types that encode local types (e.g. the type for A is shown in Line 1 in Figure 6b). The user then manually implements (step 2) the process of each participant, following their type (Line 7), using provided communication primitives (Line 13). Rumpsteak relies on Rust's typechecker to ensure the consistency of the implementation. For the sake of clarification where needed, we call *Vanilla Rumpsteak* the framework without refinements (i.e. as presented in [7]), and *Refined Rumpsteak* the framework modified to accommodate refinements.

In this section, we explain the main differences between Vanilla and Refined Rumpsteak: we introduce refinements in the types used in the runtime library, we modify the program generation step accordingly, and we introduce tools that ensure the localisation conditions are met (Definition 29 in Section 5). The overall workflow is presented in Figure 7. We conclude this section by measuring the overhead induced by the refinement w.r.t. Vanilla Rumpsteak and the time needed for asserting the localisation conditions.

```
 1 (*# RefinementTypes #*)
 2
 3 global protocol PlusMinus
 4 (role A, role B, role C)
 5 {
 6 Secret(n: int) from A to B;
 7 rec Loop {
 8   Guess(x: int) from C to B;
 9   choice at B {
10     More(x: int {x < n}) from B to C;
11     continue Loop;
12   } or {
13     Less(x: int {x > n}) from B to C;
14     continue Loop;
15   } or {
16     Correct(x: int {x = n}) from B to C;
17   }}}
```

(a) νScr description of the guessing game protocol.

```
 1 type PlusMinusA =
 2 Send<B, 'n',
 3 Secret,
 4 Tautology::<Name, Value, Secret>,
 5 Constant<Name, Value>, End>;
 6 // ...
 7 async fn a(role: &mut A)
 8 -> Result<(), Box<dyn Error>> {
 9   try_session(role,
10   HashMap::new(),
11   |s: PlusMinusA<'_, _>| async {
12     let s =
13     s.send(Secret(10)).await?;
14     return Ok(((), s))
15   })
16   .await
17 }
```

(b) Rust type and implementation of participant A of the guessing game protocol. The handwritten code (Line 7 to Line 17) is the same than with Vanilla Rumpsteak.

Figure 6 Implementation of the guessing game using Rumpsteak.

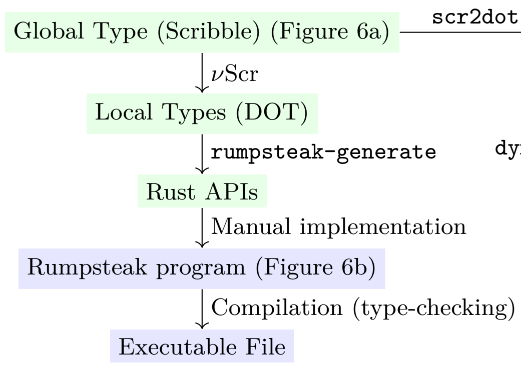


Figure 7 Workflow of Rumpsteak. Green nodes represent steps that already existed in Vanilla Rumpsteak and that have been adapted to accommodate for refinements, red nodes represent new steps, and blue nodes represent unmodified steps. The three new steps (`scr2dot`, `mpst_unroll`, and `dynamic_verify`) verify the conditions mentioned in Definition 29.

7.1 Refinement Implementation

Modifications to the Rumpsteak Library. In order to accommodate for refinements, we have to introduce new elements in to the Rumpsteak's encoding of local types. Consider the local type of participant A introduced in Example 21 $\mathsf{B}\oplus\{\mathit{secret}(n : \texttt{int} \models \top).\mathsf{end}\}$: Rumpsteak now has to take into account the name of the variable sent (n), and the refinement attached to the transition ($\top$). Consider the type declaration in Line 1 to Line 5, Figure 6b. Compared to Vanilla Rumpsteak, we introduce `'n'`, a const generic[5], that carries the name of the variable sent (Line 4). Regarding the refinement, we introduce `Tautology::<Name, Value, Secret>`,

[5] `https://github.com/rust-lang/rfcs/blob/master/text/2000-const-generics.md`

which represent the refinement $\top$. The generic parameters are used to specify the type of variable names (`chars` in our case) and values (`i32`) as well as the label of the message (`Secret`). We modified νScr and `rumpsteak-generate` to generate skeleton files (the content of the file up to Line 5). Rumpsteak provides a set of available refinements, and additional ones can be written ad-hoc (for specific needs). To add an ad-hoc refinement, the user simply implements the trait `Predicate` (which extends `Default`), which requires a method `check` that asserts whether the predicate holds. For instance, the `check` function of `Tautology` simply returns `true`.

Verification of the Conditions for Decentralised Refinement Assertion. As we explained in Section 5, to make sure that refinements can be verified in a decentralised way, we require to check that variables needed for the refinements are located correctly (Definition 29). To perform this verification, we implemented new tools for the Rumpsteak framework (in red in Figure 7).

Our tools:
(i) convert the global type into a graph (`scr2dot`);
(ii) unroll the loops once to precisely capture variables initialisations (`unroll_mpst`); and
(iii) localise variables on the unrolled graph (`dynamic_verify`).

The core part of this verification, `dynamic_verify`, finds variables locations with simple inference rules written in Datalog. We use the *crepe* library [40] which provides a Datalog DSL for Rust. We provide more details on the algorithm in [35].

Limitations. The current implementation makes extensive use of the Rust feature *const generics*[9] which introduces a limited form of dependent types in Rust. It allows to use constant values in types. As of today, only some *basic types* can be used as const generics, in particular `chars` and the various integer types. We use such const generics to encode informations about the variables into the types: for instance, the predicate $x < 5$ would have the type `LTnConst<L, 'x', 5>`, where the `'x'` and the `5` are const generics.

For readability, we choose to set variables to `chars`, meaning that in the current implementation, we can only accommodate a limited number of distinct variables. Should more be needed, one could easily modify our implementation to replace them with `u64`, which allows 2^{64} variables names. Similarly, we only consider `i32` as message payloads. Should different types of messages be needed, they could be encoded in an `enum`.

Finally, the static elision optimisation (Section 6) is not implemented.

7.2 Runtime and Localisation Benchmarks

We evaluate how Rumpsteak with refinements performs with respect to Rumpsteak without refinements. First, we measure the runtime of our analysis tool which verifies the two conditions in Definition 29 (`scr2dot`, `unroll_mpst` and `dynamic_verify`). Although not a runtime cost, and while we expect this analysis to be possibly expensive, we would like to ensure that it is still practical for test cases from the literature. Secondly, we evaluate the runtime overhead of adding refinements with respect to Rumpsteak without refinements.

Setup and Benchmark Programs. We evaluate the performance of Rumpsteak with refinement with benchmarks. Most of them are taken from the literature (Table 1). This set of program contains various micro-benchmarks with a variety of combination of properties (whether the protocol is binary or multiparty, contains recursivity or choice). Notice that protocols that contain recursivity with no choice (e.g. *simple auth* are infinite). Therefore,

Table 1 The set of micro benchmarks together with their characteristics. "MP" denotes a multiparty protocol, "Rec" the presence of recursion, and "Choice" the presence of choice.

Name	MP	Rec	Choice
① simple adder [21]	no	no	no
② travel_agency [23]	no	no	yes
③ ping pong [42]	no	yes	no
④ simple auth.	no	yes	yes
⑤ ring max	yes	no	no
⑥ three_buyers [19]	yes	no	yes
⑦ plus or minus	yes	yes	yes

such protocols are only measured in the variable localisation paragraph. Also, where it applies, protocols were modified in order to add relevant refinements; such modifications are listed below. By default, we add `Tautology` predicates (Section 7.1). The tests were performed on a machine running Ubuntu 22.04.1 LTS x86_64 (kernel 5.15.0-60) with an Intel i7-6700 processor (4 cores, 8 threads running at 4GHz maximum) and 16GB of memory[6]. We compare Rumpsteak with refinement vs. Vanilla Rumpsteak. For a comparison between Vanilla Rumpsteak and other libraries, see [7, Figure 6].

Added Refinements & Protocol Modifications. Some benchmarks from the literature were adapted in order to accommodate refinements. In addition, we introduce three benchmarks. Those benchmarks are close to examples from the literature, adapted to better highlight refinements.

simple adder: This example is adjusted from the *Adder* ([21]) protocol, but we remove the choice of operation in order to increase the benchmark diversity;

ping pong: In [42], some of the loops were statically unrolled, and the protocol contained a choice to exit. Ours is equivalent to an infinite $PingPong_1$ in [42].

simple authentication: This example is a binary example of an authentication protocol (e.g. OAuth [31]). The added refinements enforce that access is granted if and only if the given password is correct.

ring max: A multiparty protocol where participants receive a value from their predecessor (except for the initial participant), and forward an other value to their successor (the final participant forwards it to the initial one). Refinements ensure that the value forwarded is greater than or equal to the value received.

plus or minus: An implementation of our running example.

Static Analysis of Variable Locations. Table 2 shows the decentralised verification time cost for each refined global label. As shown in Figure 7, this static analysis is performed with three tools. The results shown account for the whole pipeline, and were measured over 50 samples, with 10 warmup runs (excluded from the measurements). Overall, the runtime for variable localisation is stable (around 5.6ms). We suspect that, for graphs with a low number of states, the runtime is dominated by the accesses to the file.

[6] The micro-benchmarks are not memory intensive. The memory size is not a limiting factor. However, the benchmarks seem to be dominated by the startup time, which includes memory access time.

Table 2 Benchmark of the localisation analysis (Red branch in Figure 7). $|S|$ denotes the number of states of the graph of the protocol; $|U|$ denotes the number of states after unrolling the recursion loops once; and $|V|$ denotes the number of variables in the protocol. $|S|$, $|U|$ and $|V|$ are computed manually to give an insight on how protocols compare. *et* is the execution time, measured by the benchmark (in ms).

	$\lvert S\rvert$	$\lvert U\rvert$	$\lvert V\rvert$	et $(\mu \pm \sigma)$
①	4	4	3	5.5 ± 0.2
②	7	7	6	5.5 ± 0.2
③	2	4	1	5.5 ± 0.2
④	6	11	3	5.6 ± 0.2
⑤	8	8	7	5.7 ± 0.2
⑥	10	10	7	5.6 ± 0.2
⑦	4	19	2	5.6 ± 0.2

Table 3 Evaluation of the runtime overhead due to the addition of refinements in Rumpsteak. p is the MWU p-value, m is the baseline median runtime and m_r is the median runtime with refinements when applicable ($p < 0.05$). All times are in ms.

	p	m	m_r
①	0.00	0.7	0.8
②	0.11	0.8	N/A
④	0.29	0.8	N/A
⑤	0.17	0.8	N/A
⑥	0.68	0.7	N/A
⑦	0.04	0.7	0.8

Runtime Overhead of Refinement Feature. Our second set of benchmarks aims to measure the overhead of runtime refinement verification with respect to the original Rumpsteak framework. We are expecting Rumpsteak with refinements to be slower than the original Rumpsteak, due to the additional cost of evaluating refinements. This benchmark has two objectives: first, to find out whether there is an actual, statistically significant, overhead; and second, if so, estimate this overhead. To measure this overhead, we only consider the protocols that terminate from the benchmark set.

To fulfil the first objective, we use a Mann-Whitney U test (MWU). We used MWU as it is a non-parametric test, and our runtime distributions do not follow a normal distribution, which prevents us to do simpler analysis. As MWU is sensitive to the number of samples, we run each benchmark 30 times, on both the original Rumpsteak and Rumpsteak with refinements. We perform the MWU test on the collected 30 samples, preceded by 10 iterations to warm the system up. Our hypothesis for the MWU test are the following:

H_0: The distributions of runtimes with and without refinements are identical.
H_1: The distributions of runtimes with and without refinements are distincts.

The p-values obtained from the MWU test are reported in the first column of Table 3. We also report the baseline (Rumpsteak without refinements) median run time (over the 30 runs) in the second column of the table. Most often, the overhead is not significant ($p \geq 0.05$) and H_0 can not be rejected. When the overhead is statistically significant, we also report the median runtime (over the 30 runs) of Rumpsteak with refinements in the third column. With our set of microbenchmarks, in most cases we cannot distinguish Rumpsteak with refinement from Rumpsteak without refinements. We suspect Rumpsteak runtime is dominated by communications and context switching. However, as our refinements can be arbitrarily complex, specific instances could show real slowdown due to refinement evaluation.

8 Related Work and Conclusion

Design-by-Contract for (Multiparty) Session Types. In binary session types, [37] introduces contracts for binary sessions, and provides an analysis tool which verifies whether a given program comply with its associated contract. The verification is done with symbolic execution. Compared to this paper, we address multiparty sessions. Besides, our framework is more generic (specific instances could be based on symbolic execution, but we can also accommodate other verification methods). Bocchi et al. [4] present a variant of MPST that allows predicates on exchanges, that must hold for a typed process to take transitions. The main difference with our work is that their approach focuses on *correctness by construction*, i.e. they accept only correct protocols, while we can accept protocols that fail, and we simply prevent them to generate incorrect traces. More precisely, the authors statically ensure that there is a satisfiable path, which prevents some valid runs to be accepted. For instance, consider the following type:

$$\mathsf{A} \to \mathsf{B}\,\{\ell_1(x : \mathtt{int} \models x < 10).\mathsf{B} \to \mathsf{A}\,\{\ell_2(y : \mathtt{int} \models x > y \wedge y > 6).\mathtt{end}\}\}$$

This type would be rejected in [4] since if A sends $x = 5$ (which is allowed by $x < 10$), then there is no y that satisfies $5 > y \wedge y > 6$. By rejecting this, they also reject all possibly valid runs (e.g. if A sends $x = 9$ and B replies with $y = 7$). A follow-up on this work is [3] which introduces local states, i.e. the authors allow participants to have local variables, which can be updated during process execution. The session types reflect those elements and contain predicates on exchanged variables and local variables.

With respect to these two papers, our criteria for the validity of refinements (expressed as a property of the generated trace) is decoupled from the semantics of the model. This approach allows us to be more flexible than enforcing statically the refinements, and to lower the cost of adopting refinements, in particular to retrofit refinements into existing systems. For instance, using our framework, one can simply use the centralised semantics at first, which is very expressive, without having to prove the correctness of the implementation. In a second step, users can then develop different verification or analysis techniques which can be plugged-in transparently. For instance, switching from Vanilla Rumpsteak to Refined Rumpsteak does not involve changes in the implementation, as the modifications do not happen in the programming interface. Also, compared to these papers, our framework is not bound to MPST only, and provide an actual implementation of our framework.

Design-by-Contract in Choreography Automata. Choreography Automata (CA) are graphs that represent the global behaviour of a concurrent system. The behaviour of individual participants is obtained by *projecting* well-formed CA, i.e. erasing all actions that do not concern a given participant. The result is a FSM which, after determinising and minimising, is used as a CFSM. The projection of all participants leads to a CS. Notice that CA accept some protocols that would be rejected by MPST, and vice-versa.

Gheri et al. [16] study the verification of CA with assertions. Their work and ours are distinct with respect to the following aspects:

(i) the communication semantics;
(ii) the choices;
(iii) the logic for predicates; and
(iv) the implementation presented in [16] is limited to CA without assertions (i.e., the design-by-contract approach was not implemented and left as their future work).

Regarding Item i, Gheri et al. [16] defines choreography automata with *synchronous* communication semantics, while the one we developed in this work is asynchronous. Gheri et al. [16, Section 7] discusses asynchronous semantics but it remains future works.

Regarding Item ii, we are constrained by the syntax of RMPST, in which choices can only happen between two selected participants, while choreography automata accept protocols with choices where a (single) participant A sends to multiple receivers (B and C) [16, Definition 4.15]. Explicit connections [22] is an extension of MPST that accommodates with choices with multiple receivers.

Regarding Item iii, we kept our refinement logic abstract, while it is fixed in choreography automata, with a form of first order logic. Besides, predicates are handled differently in both frameworks as well: Gheri et al. [16] require choreography automata to be *history-sensitive* [4], a definition which serves a similar purpose to our definition of *variable localisation* (Section 5 and [35]), which constrains our decentralised semantics. Our centralised semantics (Definition 10) is not constrained by variable localisation. For instance, the RMPST $\mathsf{A} \to \mathsf{B}\,\{\ell_1(x : \mathtt{int} \models \top).\mathsf{C} \to \mathsf{D}\,\{\ell_2(y : \mathtt{int} \models x = y).\mathtt{end}\}\}$ produces valid traces with our centralised semantics, while the corresponding choreography automata would be rejected.

Besides, our work introduces a general framework that can accommodate refined CA in addition to RMPST. We show [35] a possible way to do so.

Implementations of Refinements in MPST. Neykova et al. [29] develop an F# library for static verification of MPST with refinements. They present a compiler plugin which uses an SMT solver (Z3) to statically verify some refinements. They use a notion of similar to our variable localisation criterion (which they call *variable knowledge*), and a variant of CFSM with refinements that is similar to ours. In their work, refinements that are statically asserted by the SMT solver are pruned in the CFSM, while the rest of refinements are kept in the CFSM and are dynamically checked. Similarly, [41, 42] develop a framework for multiparty session types with refinements in F*. They delegate the management of refinements to F* type system (which internally uses an SMT solver). They define refinements on global types, which are then projected onto local types. They show that a global type and its projection are trace equivalent. Those two works focus on the *implementation* of MPST with refinements. [29] does not focus on the theory of refinements and the theory developed in [42] is tightly coupled to F*. For instance, they do not present a *correctness* criterion such as *valid refined traces* we present. Contrary to both works, our correctness criteria (based on valid refined traces) is *decoupled from* (i.e. independent of) any target type theory, programming language or model of computation: we only require an LTS labelled with actions. Besides, the logic used for refinements is also a parameter of our framework, and users could use alternatives, leading to a greater expressivity of our framework.

The main syntactical difference between our RMPST and those developed in [42] is that we attach refinements to the messages of the protocol, while [42] attach refinements to the payload value. This is due to a different approach: correctness in [42] is related to payload types being inhabited while our criteria of correctness (developed in Definition 7) relies on actions being allowed. In binary linear logic-based session types, [9] study the metatheory of binary session types with arithmetic refinements. In particular, they focus on the type equality, showing that added refinements make the type equality undecidable (they provide a sound but incomplete algorithm for type equality). [10] also implement a library for session types with refinements, although it only accounts for arithmetic refinements.

Other Related Works. There are various papers on the dynamic verification of MPST. For instance [2] present a framework that allows for both static and dynamic verification of MPST. This paper introduces a theory for (dynamically) monitoring assertions on messages (i.e. the equivalent of our refinements). Furthermore, the authors introduce theoretical tools (bisimulations) to relate monitored processes with correct unmonitored processes. This paper, however, suffers a few limitations. First, it focuses on *monitorable* types (which intuitively correspond to types satisfying our conditions for decentralised verification Definition 29). Second, it focuses on dynamic verification of assertions. The paper is compatible with statically verified processes (which allows turning off the dynamic monitoring), but it does not present techniques for static verification in itself.

On the other hand, our paper takes a different approach, by decoupling the correctness criterion from the verification technique. This allows us to have a more general framework (our framework accept types that are not localisable/monitorable, although not all semantics can accommodate those), as well as to develop static verification techniques.

In Rust, the `refinement` crate [11] provides refinement data types. Their approach of refinements is similar to ours, with a `Predicate` trait that provides a method to perform the predicate verification (at runtime). Refinement data types have also been implemented in multiple languages (e.g. F*, Haskell [36], etc.). On the practical side, we can note the similarities between typestates and session types [20]. [14] implements typestates in Rust with a DSL to verify protocol conformance. While Rumpsteak does not use their library, it internally uses similar constructs.

Regarding implementations of session types in Rust, there are several frameworks beside Rumpsteak. [25] first integrate binary session types in Rust, but their implementation suffers a few drawbacks (see [26, Section 3] for a detailed explanation). Sesh [26] and Ferrite [6] are two Rust libraries for *binary* session types, and they implement synchronous and asynchronous ones, respectively. MultiCrusty [27] implements synchronous MPST on top of Sesh, with a mesh of binary sessions. Compared to MultiCrusty, Rumpsteak implements directly MPST instead of wrapping them into binary sessions, and focuses on asynchronous MPST. None of the aforementioned tools develops refinements. It would be an interesting future work to apply our criteria to extend their tools with refinements.

Finally, we note the proximity between (MP)ST with refinements and dependent (MP)ST. For instance, [33] introduce a session type calculus with label-dependency (their approach does not explicitly account for payload value refinement). Other approaches exist, for instance, an intuitionistic linear logic-based type theory for building value-dependent session types [34], and separation logic-based work for reasoning about session types [17].

Future Work While, in our work, we consider MPST with payloads (some variants only consider messages with labels), we restrict our MPST with a single payload (i.e. *monadic* MPST, where each message carries a single value). The extension to polyadic MPST, where a message can carry multiple values, is straightforward, by adapting the RCS rules ([GRSnd] and [GRRec], Definition 14).

We presented two optimisations, in order to illustrate the flexibility of our theoretical framework. Regarding the decentralised verification (Section 5), there is room for an extension, e.g. with specific domains (i.e. some class of protocols with specific refinements). Regarding the static elision of redundant refinements, we envision improving the technique with use of SMT solvers could be promising. The main difficulty lies in asynchronous communications: one would need to consider all possible message orderings before solving constraints.

References

1 Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2020. doi:10.1007/978-3-030-50029-0_6.

2 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theoretical Computer Science*, 669:33–58, 2017. doi:10.1016/j.tcs.2017.02.009.

3 Laura Bocchi, Romain Demangeon, and Nobuko Yoshida. A Multiparty Multi-Session Logic. In *7th International Symposium on Trustworthy Global Computing*, volume 8191 of *LNCS*, pages 111–97. Springer, 2012.

4 Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, pages 162–176, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-15375-4_12.

5 Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, April 1983. doi:10.1145/322374.322380.

6 Ruofei Chen and Stephanie Balzer. Ferrite: A Judgmental Embedding of Session Types in Rust, 2021. (repository is found at https://github.com/ferrite-rs/ferrite). arXiv:2009.13619.

7 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in rust with multiparty session types. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 246–261, New York, NY, USA, April 2022. Association for Computing Machinery. doi:10.1145/3503221.3508404.

8 Gérard Cécé and Alain Finkel. Verification of programs with half-duplex communication. *Information and Computation*, 202(2):166–190, November 2005. doi:10.1016/j.ic.2005.05.006.

9 Ankush Das and Frank Pfenning. Session Types with Arithmetic Refinements. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CONCUR.2020.13.

10 Ankush Das and Frank Pfenning. Rast: A Language for Resource-Aware Session Types. *Logical Methods in Computer Science*, Volume 18, Issue 1, January 2022. doi:10.46298/lmcs-18(1:9)2022.

11 Brady Dean and Joey Ezechiëls. refinement crate, 2021. (repository is found at https://github.com/2bdkid/refinement).

12 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Session Types Meet Communicating Automata. In *21st European Symposium on Programming*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.

13 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 174–186, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39212-2_18.

14 José Duarte and António Ravara. Retrofitting Typestates into Rust. In *25th Brazilian Symposium on Programming Languages*, pages 83–91, Joinville Brazil, September 2021. ACM. doi:10.1145/3475061.3475082.

15 Francisco Ferreira, Fangyi Zhou, Simon Castellan, and Benito Echarren. NuScr, 2019. URL: https://github.com/nuscr/nuscr.

16 Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. Design-By-Contract for Flexible Multiparty Session Protocols. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2022.8`.

17 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, January 2020. `doi:10.1145/3371074`.

18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, January 2008. `doi:10.1145/1328897.1328472`.

19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63(1):9:1–9:67, March 2016. `doi:10.1145/2827695`.

20 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 329–353, Berlin, Heidelberg, June 2010. Springer-Verlag.

21 Raymond Hu and Nobuko Yoshida. Hybrid Session Verification Through Endpoint API Generation. In Perdita Stevens and Andrzej Wąsowski, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 401–418, Berlin, Heidelberg, 2016. Springer. `doi:10.1007/978-3-662-49665-7_24`.

22 Raymond Hu and Nobuko Yoshida. Explicit Connection Actions in Multiparty Session Types. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 116–133, Berlin, Heidelberg, 2017. Springer. `doi:10.1007/978-3-662-54494-5_7`.

23 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 516–541, Berlin, Heidelberg, 2008. Springer. `doi:10.1007/978-3-540-70592-5_22`.

24 International Telecommunication Union. Z.120 : Message Sequence Chart (MSC), February 2011.

25 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, pages 13–22, Vancouver BC Canada, August 2015. ACM. `doi:10.1145/2808098.2808100`.

26 Wen Kokke. Rusty Variation: Deadlock-free Sessions with Failure in Rust. *Electronic Proceedings in Theoretical Computer Science*, 304:48–60, 2019. (repository is found at `https://github.com/wenkokke/sesh`). `doi:10.4204/eptcs.304.4`.

27 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1868-8969. `doi:10.4230/LIPIcs.ECOOP.2022.4`.

28 Bertrand Meyer. Design by Contract. *Advances in Object-Oriented Software Engineering*, pages 1–35, 1991.

29 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in F#. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 128–138, New York, NY, USA, February 2018. Association for Computing Machinery. `doi:10.1145/3178372.3179495`.

30 Davide Sangiorgi. *An Introduction to Bisimulation and Coinduction.* Cambridge University Press, Cambridge ; New York, 2012.

31 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proceedings of the ACM on Programming Languages*, 3(POPL):30:1–30:29, January 2019. `doi:10.1145/3290343`.

32 Felix Stutz. Asynchronous Multiparty Session Type Implementability is Decidable - Lessons Learned from Message Sequence Charts. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ECOOP.2023.32*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2023. `doi:10.4230/LIPIcs.ECOOP.2023.32`.

33 Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, January 2020. `doi:10.1145/3371135`.

34 Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, pages 161–172, Odense Denmark, July 2011. ACM. `doi:10.1145/2003476.2003499`.

35 Martin Vassor and Nobuko Yoshida. Refinements for multiparty message-passing protocols: Specification-agnostic theory and implementation, 2024. Full version on Arxiv.

36 Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, University of California, San Diego, USA, 2016. URL: `http://www.escholarship.org/uc/item/8dm057ws`.

37 Jules Villard. *Heaps and Hops*. PhD thesis, Laboratoire Spécification et Vérification, École Normale Supérieure de Cachan, France, February 2011.

38 Nobuko Yoshida and Lorenzo Gheri. A Very Gentle Introduction to Multiparty Session Types. In Dang Van Hung and Meenakshi D´Souza, editors, *Distributed Computing and Internet Technology*, Lecture Notes in Computer Science, pages 73–93, Cham, 2020. Springer International Publishing. `doi:10.1007/978-3-030-36987-3_5`.

39 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, pages 22–41, Cham, 2014. Springer International Publishing.

40 Erik Zhang. Crepe, 2022. URL: `https://crates.io/crates/crepe`.

41 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically Verified Refinements for Multiparty Protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. `doi:10.1145/3428216`.

42 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically Verified Refinements for Multiparty Protocols. *arXiv:2009.06541 [cs]*, September 2020. arXiv: 2009.06541. `arXiv:2009.06541`.