

CoMPSeT: A Framework for Comparing Multiparty Session Types

Telmo Ribeiro

telmo.ribeiro@fc.up.pt

Department of Computer Science

Faculty of Sciences, University of Porto
Portugal

José Proença

jose.proenca@fc.up.pt

CISTER & Department of Computer Science

Faculty of Sciences, University of Porto
Portugal

Mário Florido

amflorid@fc.up.pt

LIACC & Department of Computer Science

Faculty of Sciences, University of Porto
Portugal

Concurrent systems are often complex and difficult to design. Choreographic languages, such as Multiparty Session Types (MPST), allow the description of global protocols of interactions by capturing valid patterns of interactions between participants. Many variations of MPST exist, each one with its rather specific features and idiosyncrasies. Here we propose a tool – **CoMPSeT** – that provides clearer insights over different features in existing MPST. We select a representative set of MPST examples and provide mechanisms to combine different features and to animate and compare the semantics of concrete examples. **CoMPSeT** is open-source, compiled into JavaScript, and can be directly executed from any browser, becoming useful both for researchers who want to better understand the landscape of MPST and for teachers who want to explain global choreographies.

1 Introduction

Communicating systems can be described by a variety of formalisations, often differing in subtle but significant ways – such as their treatment of concurrency, message ordering, or assumptions about synchrony – which makes their analysis non-trivial. These challenges build upon the inherent difficulties in the architecture of such systems, where numerous execution flows and behaviours must be understood to ensure the absence of communication errors [19].

This paper focuses on *Multiparty Session Types (MPST)*, a typing discipline that guarantees communication safety and liveness in concurrent systems [10], originally formulated by Honda et al. [14]. *Session Types* denote a formalism capable of verifying correctness in concurrent programs, where the well-behaviour of a protocol can be asserted through the well-typedness of its participants. The *Multiparty* aspect generalizes the earlier concept of *Binary Session Types* [13], which considered communications between only two parties.



Figure 1: The classical Multiparty Session Types’ framework

The *classical* MPST framework, illustrated in Fig. 1, begins with the specification of a *global type*. The global type describes the expected communication behaviour of a system from a global perspective, detailing how participants interact and how the session evolves over time, as exemplified in Example 1.1.¹

Example 1.1. *A possible global type describing a session where a controller assigns a task (Work) to two workers, worker_A and worker_B, in this particular order. The workers are then expected to reply with a completion message (Done) in any order.*

$$\begin{aligned} & \text{controller} \rightarrow \text{worker}_A : \text{Work} ; \text{controller} \rightarrow \text{worker}_B : \text{Work} ; \\ & (\text{worker}_A \rightarrow \text{controller} : \text{Done} \parallel \text{worker}_B \rightarrow \text{controller} : \text{Done}) \end{aligned}$$

From the global specification, *local types* – also known as *session types* – may be derived via a *projection* operation. Each local type reflects the perspective of a single participant and contains only the actions in which that participant is involved – either as sender or as receiver – similar to the depiction in Example 1.2. Local types can then be used to statically type-check processes implementing corresponding participants.

Example 1.2. *The local types derived from the projection of Example 1.1, where worker_{AB} is used to reference both worker_A and worker_B, as they share an identical structure.*

$$\begin{aligned} L_{\text{controller}} &= \text{worker}_A ! \text{Work} ; \text{worker}_B ! \text{Work} ; (\text{worker}_A ? \text{Done} \parallel \text{worker}_B ? \text{Done}) \\ L_{\text{worker}_{AB}} &= \text{controller} ? \text{Work} ; \text{controller} ! \text{Done} \end{aligned}$$

Formally, let G be a well-formed global type involving participants p_1, \dots, p_n . If, for each $1 \leq i \leq n$, there exists a process P_i such that $\vdash P_i : (G|_{p_i})$ – where $|$ denotes the projection of G onto participant p_i – then the composed concurrent system $(P_1 \mid \dots \mid P_n)$ is guaranteed to be both safe and live [7]. Here, the projection operation must be a partial function undefined for global types that do not meet the conditions required to ensure these guarantees.

We developed CoMPSeT, a tool for comparing MPST sessions and semantics through hands-on experimentation and visualisation. The core contribution of CoMPSeT lies in its ability to support not only the comparison of distinct sessions under the same semantic but also of semantics employing different formalisms, such as synchronous versus asynchronous communication models. Furthermore, it enables users to configure the underlying semantics according to a selected set of features (see Fig. 2) and to immediately observe the practical effects of varying formalisation choices.

Our tool pivots CAOS [20, 21], a framework that enables programmers to test, define, and animate both sessions and structural operational semantics (SOSs) by defining widget (builders) – visual and/or interactive blocks of extendable functionality – called upon as functions. CoMPSeT instantiates these builders with appropriate parameters to generate the visual and interactive elements configured according to the user’s choice. Importantly, while CAOS is a general-purpose framework for SOSs and does not specifically target MPST, CoMPSeT is designed with a modular architecture defining all MPST-specific components, including projections, operational semantics, and well-formedness conditions. These components are then parametrised against the configurations selected by the user and animated through the CAOS framework.

¹We employ \rightarrow to denote the communication of a data type between two participants, while $;$ and \parallel represent the sequential and parallel composition, respectively. Additionally, $!$ and $?$ highlight the direction of a communication, marking sending and receiving actions.

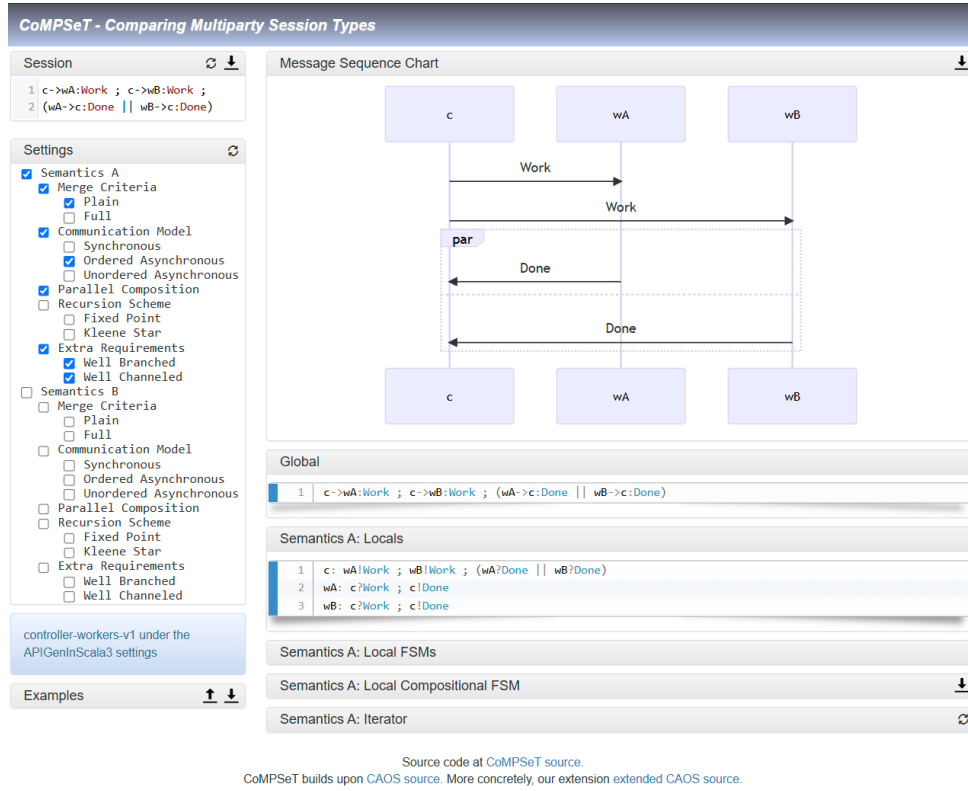


Figure 2: CoMPSeT representations for Example 1.1 and Example 1.2, where `controller`, `workerA`, and `workerB` are abbreviated as `c`, `wA`, and `wB`, respectively

Remarks on the scope

We avoid establishing new lemmas and theorems, as our focus is neither on the development of new formulations nor on a comprehensive survey of existing ones, but rather on the introduction of a system capable of effectively comparing MPST variation points. Although we often provide formalisms to support our notations, they are largely grounded in existing contributions.

Contributions Our primary contribution is a prototype open-source tool called CoMPSeT, available to be executed online at <https://telmoriibeiro.github.io/CoMPSeT>. This tool uses a dedicated input language for specifying global types (see Section 2.1) and supports configurations over how the language is interpreted, such as choosing between synchronous and asynchronous communication or selecting what constructs are permissible (see Section 3). Users can configure and compare two different semantics within the same session, with additional analysis available through branching bisimulation checkers. Sessions are visually represented using message sequence charts (MSCs) and projections of the global type, while identifying and reporting errors when projections are undefined. Finally, it provides semantic animations, either through step-by-step execution or by rendering the entire state space. As a complementary contribution, we extended the CAOS framework to overcome two identified limitations: (1) lack of runtime widget variability and (2) absence of user-driven parametrisation. These extensions were crucial for supporting the configurability and interactivity of CoMPSeT and are made available as an independent fork at <https://github.com/TelmoRibeiro/CAOS>.

Organisation of the paper Section 1 introduces the motivation, problem statement, and contributions of this work. Section 2 provides an overview of the MPST framework, including definitions of global and local types, projection mechanisms, and semantics for different communication models. Section 3 surveys existing formalisms while describing our notions of *feature* and *base semantics*. Section 4 details the extensions made to the CAOS framework to support dynamic widget behaviour and user-configurability, enabling the innovative aspects leveraged in CoMPSeT. Section 5 defines CoMPSeT, describing its interface, features, and use cases through illustrative examples while highlighting how it enables interactive and visual comparisons of different MPST formalisms. Finally, Section 6 summarises the main contributions and findings and outlines directions for future research and development.

2 Multiparty Session Types in a Nutshell

This section serves simultaneously as a gentle introduction to the MPST framework and as the theoretical foundation for understanding the core principles behind our tool.

2.1 Global Types

Let \mathbb{P} be the set of all *participants*, ranged over by p, q, r ; let \mathbb{T} be the set of all *data types*, ranged over by t ; and let \mathbb{G} be the set of all *global types*, ranged over by G . The syntax of a global type G is given by the following grammar.

$$G ::= p \rightarrow q : \{t_i ; G_i\}_{1 \leq i \leq n} \mid G_1 ; G_2 \mid G_1 \parallel G_2 \mid \mu X. G \mid X \mid (G)^* \mid \text{skip}$$

The syntax and their informal interpretations derive primarily from the descriptions provided by Cledou et al. [5], as well as Jongmans and Proença [17], while acknowledging that both developments reference Daniélou and Yoshida [7] as their primary source concerning their own syntax.

- $p \rightarrow q : \{t_i ; G_i\}_{1 \leq i \leq n}$ specifies the *communication* of a label t_i from the participant p to the participant q , followed by the global type G_i , for some $1 \leq i \leq n$. As an additional well-formedness requirement, we stipulate that (1) $p \neq q$ (i.e., no self-communications) and (2) the labels in t_i must be pairwise distinct (i.e., deterministic continuations). Moreover, we write $p \rightarrow q : \{t_i\}_{1 \leq i \leq n} ; G$ as a shorthand for $p \rightarrow q : \{t_i ; G\}_{1 \leq i \leq n}$.
- $G_1 ; G_2$ specifies the *sequential composition* of G_1 and G_2 .
- $G_1 \parallel G_2$ specifies the *parallel composition* of G_1 and G_2 .
- $\mu X. G$ and X specify (3) guarded and (4) bounded *recursive protocols* achieved through *fixed point* notation.
- $(G)^*$ specifies (3) guarded *recursive protocols* achieved through *Kleene star* notation.
- skip specifies *sequence identity*.

Regarding the fixed point notation, we take the equi-recursive viewpoint, not distinguishing between $\mu X. G$ and its unfolding $G[\mu X. G/X]$, as is the usual case within MPST [14].

Furthermore, the constructs adopted are not extensive of the literature, which contains instances such as the *universal quantification* in the work of Daniélou and Yoshida [7], and *session delegation* established by Bejleri and Yoshida [2].

Example 2.1. The global type below captures a session where the controller delegates a task and the worker responds with a completion note. This communication pattern can then be repeated.

$$\begin{aligned} &\mu X. \text{controller} \rightarrow \text{worker} : \{ \\ &\quad \text{Work} ; \text{worker} \rightarrow \text{controller} : \text{Done} ; X, \\ &\quad \text{Quit} \\ &\} \end{aligned}$$

2.2 Local Types & Projections

Local types are defined by the following grammar.

$$L ::= \text{pq}!\{\mathbf{t}_i ; L_i\}_{1 \leq i \leq n} \mid \text{pq}?\{\mathbf{t}_i ; L_i\}_{1 \leq i \leq n} \mid L_1 ; L_2 \mid L_1 \parallel L_2 \mid \mu X.L \mid X \mid (L)^* \mid \text{skip}$$

The informal meaning of the local types is such that:

- $\text{pq}!\{\mathbf{t}_i ; L_i\}_{1 \leq i \leq n}$ specifies a *sending* of a label \mathbf{t}_i from the participant \mathbf{p} to the participant \mathbf{q} , followed by the local type L_i , for some $1 \leq i \leq n$. Moreover, we write $\text{pq}!\{\mathbf{t}_i\}_{1 \leq i \leq n} ; L$ as a shorthand for $\text{pq}!\{\mathbf{t}_i ; L\}_{1 \leq i \leq n}$.
- $\text{pq}?\{\mathbf{t}_i ; L_i\}_{1 \leq i \leq n}$ specifies the *reception* of a label \mathbf{t}_i expected by the participant \mathbf{q} from the participant \mathbf{p} , followed by the local type L_i , for some $1 \leq i \leq n$. Moreover, we write $\text{pq}?\{\mathbf{t}_i\}_{1 \leq i \leq n} ; L$ as a shorthand for $\text{pq}?\{\mathbf{t}_i ; L\}_{1 \leq i \leq n}$.
- the remaining constructs – sequencing, parallel composition, recursion, and skip – mirror their global type counterparts.

The local types are obtained through the *projection* of the global type through each participant. This notion is formalised in Fig. 3, which in turn is based upon the definition from Daniélou and Yoshida [7] as well as Yoshida and Gheri [25].

Remarks on the syntax

We make the assumption that merging a single local type (see Fig. 3) returns the same local type, in which case, we omit the braces commonly used to denote several branching continuations.

Whenever it is made clear by the context, we omit the subject of the sending (!) and receiving (?) action. This approach is not followed in the formulations, which adhere to the literature motifs. For instance, we omitted `controller` in $L_{\text{controller}}$ (see Example 1.2).

When that is not possible, the notation for communication actions can be made explicit between participants, to avoid ambiguous naming and improve readability. For example, we use `controller!worker:Work`, instead of `controllerworker!Work`.

For the sake of simplicity, this syntax does not discriminate between payload types and message labels, a characteristic that is not commonly observed in the literature [14, 15, 2, 7, 8, 23, 6, 25]. Rather, following the main sources for the syntax, it uses data types, which may be used to refer interchangeably to either notion.

All the above decisions are transposed to CoMPSeT.

The function *participants* returns a set of all participants in a given global type G . On the other hand, *merge* is a partial function that combines a set of local types L_i into a single one, respecting a strategy that will depend on a *merge criterion*, a point of discussion in Section 3.

$skip _r = skip$	
$X _r = X$	
$(\mu X.G) _r = \mu X.(G _r)$	if $r \in participants\{G\}$
$(\mu X.G) _r = skip$	if $r \notin participants\{G\}$
$(G)^* _r = (G _r)^*$	if $r \in participants\{G\}$
$(G)^* _r = skip$	if $r \notin participants\{G\}$
$p \rightarrow q : \{t_i ; G_i\}_{1 \leq i \leq n} _r = pq!\{t_i ; (G_i _r)\}_{1 \leq i \leq n}$	if $p = r \neq q$
$p \rightarrow q : \{t_i ; G_i\}_{1 \leq i \leq n} _r = pq?\{t_i ; (G_i _r)\}_{1 \leq i \leq n}$	if $p \neq r = q$
$p \rightarrow q : \{t_i ; G_i\}_{1 \leq i \leq n} _r = merge(\{G_i _r\}_{1 \leq i \leq n})$	if $p \neq r \neq q$
$(G_1 ; G_2) _r = (G_1 _r) ; (G_2 _r)$	
$(G_1 \parallel G_2) _r = (G_1 _r) \parallel (G_2 _r)$	
undefined	otherwise

Figure 3: Projection $G|_r$ of a global type G^2 to a participant r , using functions *participants* and *merge*

Remarks on the Kleene star

Notably, the Kleene star is not disclosed in the papers referenced for our syntax. Instead, the construct was originally introduced by Castagna et al. [3], although it was not fully integrated into their theoretical framework, as it was reduced to a fixed-point operator during projection. More recently, Jongmans and Proença incorporated support for the Kleene star in their tool implementation, but without an accompanying formal definition in their paper. In this work, we formalise the rule applied in their implementation. Importantly, as shown by Charalambides et al. [4], the Kleene star can be used to define protocols whose projections are unsafe – meaning they produce local types that are not compliant with the original global specification. For instance, the authors present the type $(a \rightarrow b : m ; b \rightarrow c : m')^* ; c \rightarrow d : m''$ and explain that c cannot determine whether it should wait for m' from b , or skip directly to sending m'' to d . In the same work, the authors propose a Kleene star projectability criterion (*KP*) to ensure that such ambiguity does not arise. Informally, the criterion requires that all participants in global types of the form $(G_1)^* ; G_2$ are able to distinguish between G_1 and G_2 . As such, we assume well-defined projections of the Kleene star to be, additionally, *KP*-consistent.

Example 2.2. The two local types below capture the projections of the global type from Example 2.1. We present this example before any concrete formulation of *merge*, under the guarantee that all implementations discussed throughout this paper yield the same result for this instance.

$$\begin{aligned}
L_{\text{controller}} &= \mu X. \text{worker}! \{ \\
&\quad \text{Work} ; \text{worker?Done} ; X, \quad \text{Quit} \\
&\quad \} \\
L_{\text{worker}} &= \mu X. \text{controller?} \{ \\
&\quad \text{Work} ; \text{controller!Done} ; X, \quad \text{Quit} \\
&\quad \}
\end{aligned}$$

²We assume *KP*-consistent [4] global types, following the discussion in *Remarks on the Kleene star*.

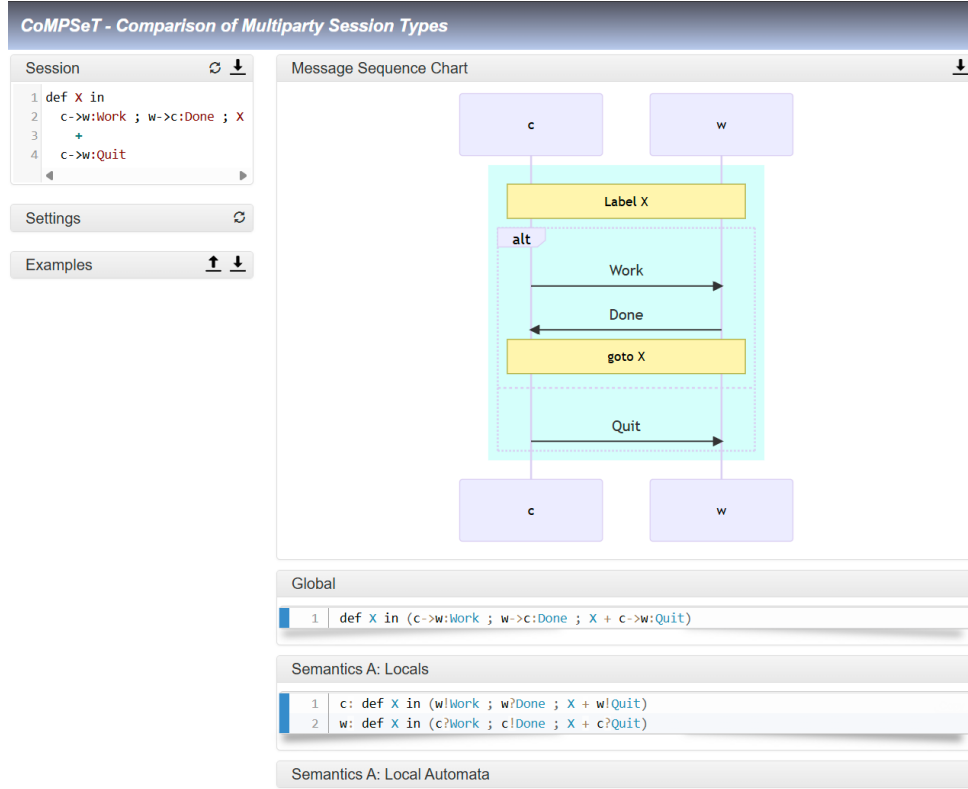


Figure 4: CoMPSeT representations for Example 2.1 and Example 2.2, where *controller* and *worker* are abbreviated as *c* and *w*, respectively

Both Example 2.1 and Example 2.2 are represented in Fig. 4, which illustrates their translation into visual widgets in CoMPSeT.

2.3 Running Local Types

We define a *Multiparty Session*, denoted by M , as a concurrent composition of local types, written $(L_{p_1} \mid \dots \mid L_{p_n})$, where each L_p is a well-formed local type for some participant $p \in M$. This definition is based on the following considerations: (1) local types can mimic the evolution of well-typed processes, and (2) the employment of semantics over local descriptions is common in choreographic languages beyond MPST. Our semantics deviate from the standard approach, which typically relies on variants or extensions of π -calculus to establish reduction rules over processes.

The reduction rules are defined over *configurations* of the form $\langle M, p \rangle$, consisting of a Multiparty Session M (or, occasionally, a single local type) and a collection p representing possible pending communications. The realisation of p is established during the definitions of concrete communication rules, when it is assumed to be shared among the remaining reduction rules. Multisets or first-in first-out (FIFO) queues would be possible realisations of p .

As with syntax, CoMPSeT supports the full set of reduction rules presented throughout this section, only with minor deviations such as the employment of $env : X \rightarrow L$ – where X ranges over recursion variables and L over local types – as mappings for recursion fixed points, which are omitted from the formulations for simplicity.

We describe the communication rules for three semantics, also omitting structural congruence definitions and shared reduction rules, which are only disclosed in the extended version [22].

Synchronous Semantics The synchronous communication rule assumes no buffering mechanism, hence p is always empty and absent in the notation. We assume that $\mathbf{t}_k \in \bigcup_{i=1}^{m_i} \mathbf{t}_i \cap \bigcup_{j=1}^{m_j} \mathbf{t}_j$.

$$\langle \mathbf{pq}!\{\mathbf{t}_i ; L_{1_i}\}_{1 \leq i \leq m_i} \mid \mathbf{pq}?\{\mathbf{t}_j ; L_{2_j}\}_{1 \leq j \leq m_j} \mid M \rangle \xrightarrow{\mathbf{p} \rightarrow \mathbf{q}:\mathbf{t}_k} \langle L_{1_k} \mid L_{2_k} \mid M \rangle \quad (\text{communication})$$

Ordered Asynchronous Semantics Here, each pair of participants represented simply by \mathbf{pq} has an unbounded FIFO queue for messages. A configuration is established by a Multiparty Session M and a buffer $p : (\mathbb{P} \times \mathbb{P}) \rightarrow \mathbb{T}^*$, mapping each pair *sender-receiver* to a sequence of data types in \mathbb{T} . The main operational rules to evolve a configuration – send and receive – are presented below. Here, ts represents a queue of data types, where $\mathbf{t}_i \cdot ts$ is used to highlight the prefix element while conversely, $ts \cdot \mathbf{t}_i$ emphasizes the suffix.

$$\begin{aligned} \langle \mathbf{pq}!\{\mathbf{t}_i ; L_i\}_{1 \leq i \leq m} \mid M, p \cup \{\mathbf{pq} \mapsto ts\} \rangle &\xrightarrow{\mathbf{pq}!\mathbf{t}_k} \langle L_k \mid M, p \cup \{\mathbf{pq} \mapsto ts \cdot \mathbf{t}_k\} \rangle & (\text{send}) \\ \langle \mathbf{pq}?\{\mathbf{t}_i ; L_i\}_{1 \leq i \leq m} \mid M, p \cup \{\mathbf{pq} \mapsto \mathbf{t}_k \cdot ts\} \rangle &\xrightarrow{\mathbf{pq}?\mathbf{t}_k} \langle L_k \mid M, p \cup \{\mathbf{pq} \mapsto ts\} \rangle & (\text{receive}) \end{aligned}$$

Unordered Asynchronous Semantics Here the configurations are established as pairs of Multiparty Sessions M and a multiset $p \in \mathcal{M}(\mathbb{P} \times \mathbb{P} \times \mathbb{T})$, where $\mathcal{M}(X)$ denotes the set of all finite multisets over the set X . The main operational rules to evolve a configuration are presented below. We denote by $p \cup (\mathbf{p}, \mathbf{q}, \mathbf{t}_i)$ a new multiset achieved by joining the tuple comprising \mathbf{p}, \mathbf{q} , and \mathbf{t}_i to an existing multiset p .

$$\begin{aligned} \langle \mathbf{pq}!\{\mathbf{t}_i ; L_i\}_{1 \leq i \leq m} \mid M, p \rangle &\xrightarrow{\mathbf{pq}!\mathbf{t}_k} \langle L_k \mid M, p \cup (\mathbf{p}, \mathbf{q}, \mathbf{t}_k) \rangle & (\text{send}) \\ \langle \mathbf{pq}?\{\mathbf{t}_i ; L_i\}_{1 \leq i \leq m} \mid M, p \cup (\mathbf{p}, \mathbf{q}, \mathbf{t}_k) \rangle &\xrightarrow{\mathbf{pq}?\mathbf{t}_k} \langle L_k \mid M, p \rangle & (\text{receive}) \end{aligned}$$

While *unordered asynchronous* models are atypical in the MPST literature, we include them to showcase the flexibility of the proposed tool, a point elaborated further in the following Section 3.

3 Variations in Multiparty Session Types Formalisms

This section identifies a set of variation points between similar MPST formalisations, regarding their semantics and expressiveness, which we categorise as *features*. More formally, it defines a feature as a modular aspect in the literature that can vary independently across different semantic implementations. For example, the realisation of the *merge* partial function (see Fig. 3), used to reconcile the projection of branches. This analysis is inspired by the methodology behind the *essential features* of Bejleri et al. [1], which compartmentalises structural motifs in MPST. In contrast, our focus lies specifically on the expressiveness of global and local types while disregarding process-oriented variations. Additionally, we define *base semantics* as the operational behaviour of our MPST syntax after selecting an explicit set of features.

The following variation points are considered.

Merge criteria This feature specifies the *merge* implementation, which handles the projected behaviour of a non-participating role observing branching communication between two others. We borrow the

nomenclature from Scalas and Yoshida [23], thus contemplating the *plain merge* and the *full merge*. Informally, plain merge is only defined when all branching communications have the *same* continuation, in which case, that continuation is yielded. The intuition is that if the continuations are the same, the projecting participant does not need to distinguish the branches. Conversely, full merge – as introduced by Yoshida et al. [24, 8] – extends this notion by allowing distinct yet *compatible* [23, 25] communications.

Communication models This feature captures the underlying communication system between participants. We consider three models: (1) *synchronous* where senders and receivers communicate in a lock-step; (2) *ordered asynchronous* with an unbounded FIFO queue for pending communications; and (3) *unordered asynchronous* which offers no guarantees on the ordering of pending messages. The concrete implementations referenced in the literature may differ from those introduced in Section 2.3, which focused on exemplifying how local types could be run natively and introducing a baseline for the comparisons observed in CoMPSeT. For instance, Coppo et al. [6] employ a single message queue with additional structural congruence rules to enable reordering, rather than assigning separate queues to each pair [sender-receiver](#).

Parallel Composition This feature captures whether local parallel composition is explicitly supported in the type system, exemplified by Daniélou and Yoshida [7], Cledou et al. [5], and Jongmans and Proença [17]. Notably, some formalisations allow for parallelism at the global type level but not on local types, as evidenced by Bejleri and Yoshida [2]. All constructs supported in CoMPSeT are done so throughout the complete type system, however, operators concerning only the global types could be established under the new extension discussed in Section 4.

Recursion Scheme This feature describes whether and how recursion is supported – either as a fixed point construct or via Kleene star notation, representing zero or more repetitions of a term in sequence. As remarked in Section 2.2, some systems project the Kleene star as fixed points, forfeiting its native support at the local type level. A detailed discussion on the expressiveness of different recursion constructs falls outside the scope of this paper. Yet, recursion in MPST, often blurs the line between expressiveness and syntactic sugar, where we contrast the previous case with the tail-recursive fixed point implementations that could be reduced to Kleene star. This distinction motivates our decision to implement a local Kleene star with dedicated reduction rules.

Well-formedness requirements This feature regards additional requirements that are conditionally imposed. For instance, as an additional requirement for the parallel composition, Daniélou and Yoshida [7] require that sub-protocols be *well channelled*, i.e., that their communications do not overlap: $\text{comm}(G_1) \cap \text{comm}(G_2) = \emptyset$. Here, $\text{comm} : \mathbb{G} \rightarrow 2^{\mathbb{P} \times \mathbb{P} \times \mathbb{T}}$ maps a global type to the set of its communications, represented as triples (p, q, t) .

Remarks on the selected papers

The reasoning behind the inclusion of [25], [6], [5], and [17] derives partially from their role as major references while establishing our own formalisms. The introductory nature of [25] and [6] is shared by the previous sections and allowed for simpler implementations. Meanwhile, [5] and [17] describe tools with design principles shared with CoMPSeT, e.g., being accomplished atop CAOS. In fact, readers familiar with both works can notice how closely our system emulates their original semantics. In contrast, [11] was included because of its unordered asynchronous communications. It simultaneously allows for greater variability and stands as motivation for future extensions, since it describes a communication model found in choreographic languages [11, 12] outside the MPST scope.

Table 1: Features mapping – ✓ (present), ✗ (absent) or N/S (not specified)³

Paper	Merge criteria	Communication model	Parallel composition	Recursion scheme	Well-formedness requirements
[25]	plain & full	synchronous	✗	fixed point	
[6]	plain	ordered asynchronous	✗	fixed point	
[5]	plain	ordered asynchronous	✓	✗	well-channelled
[17]	plain	ordered asynchronous	✓	Kleene star & fixed point	well-channelled
[11]	N/S	unordered asynchronous	N/S	N/S	N/S

Table 1 illustrates how selected MPST systems combine different features, indicating in the left column a reference to a paper on multiparty communications – here defined as the conjoined works on MPST and choreographies – and in the other columns the selection of features used by them. Notably, in [17] (*ST4MP*) we reference decisions for both the paper and the accompanying implementation (in **bold**), motivated by the previous discussion on recursion.

4 Extending CAOS

CAOS [20, 21] is defined both as a methodology and a programming framework for computer-aided design of SOSs for formal models. It supports simultaneous development of semantic foundations and corresponding interactive tools, enabling developers to define reduction rules, use cases structured as examples, and interface elements to visualise and interact with established sessions and operational semantics. This integration facilitates early detection of incongruences in formalisations, particularly during the modelling and verification of formal semantics.

We selected **CAOS** as the foundation for CoMPSeT due to its comprehensive collection of widget builders and its support for visual, interactive means – a key requirement for the tool. In particular, we leveraged the following core widgets: *Its* – to visualise local types or their compositional behaviour under specific communication models; *steps* – to allow users to interactively compute traces through step-by-step evaluations; and *compareBranchingBisim* – to determine whether two states under different semantics are branching bisimilar.⁴

Despite its flexibility, **CAOS** presented shortcomings for our implementation which other developers may equally face when extending the framework, concerning how to select a set of analysis without overwhelming themselves and/or the user. For instance, a subtle limitation lies in the way internal configurability is handled. Although **CAOS** supports multiple forms of semantical equivalence checking – such as the aforementioned branching bisimulation, but also strong bisimulation and trace equivalence – its usage is often limited to small sets of semantics. While it is technically possible to implement parametrised or configurable semantics and compare them through those tools, each configuration needs

³Notation used for concrete variation points disregarded in our analysis. This decision was motivated by the paper describing a choreographic language in which our sole interest was the communication model.

⁴Given the possibility of infinite behaviour, **CAOS** constrains bisimulation checking with a depth bound, set to 100 in CoMPSeT.

to be explicitly accounted for, which can quickly lead to scenarios where the number of semantics is just too large to feasibly maintain.

This inflexibility is exemplified by the manner widgets are defined, stopping **CAOS** from dynamically modifying them.

```
1 /** Main widgets, on the right hand side of the screen*/
2 val widgets: Iterable[(String, WidgetInfo[Stx])]
```

Figure 5: Signature for *widgets* in **CAOS**

As shown in Fig. 5, the selection of widgets is declared as an iterable collection of pairs, each containing the name of a widget and a structure encapsulating its functionality, like the previously discussed *steps*. This selection of widgets is immutable, reflected by the keyword *val* and by the use of an immutable iterable structure, hence it cannot be updated at runtime to adapt to different configurations. As a consequence, if one wishes to create a widget that interprets *steps* under synchronous semantics for one example set and asynchronous semantics for another, two distinct widgets must be declared. Under the current design, this will clutter the web interface with widgets yielding meaningless results or throwing exceptions depending on the input.

To address these limitations, we propose a threefold extension where we: (1) refactored core components of the current implementation to support runtime widget variability; (2) implemented a new configurable input widget – **Settings** – whose structure is defined by the developer through a lightweight domain-specific language (DSL) and can be interactively modified by users via checkboxes [22]; and (3) defined an application programming interface (API) over this structure, providing methods for accessing and updating it, alongside general-purpose filters and auxiliary functions [22].

Those extensions were critical to the innovations observed in **CoMPSeT**. They enable the programmer to establish widgets that are parametrised by user-selected configurations and which dynamically adapt their behaviour. More concretely, in **CoMPSeT**, each widget is instantiated by binding its logic to the current state of **Settings**, where we then leverage our modular definitions that capture key aspects of the MPST framework, including global type projection, operational semantics, and well-formedness verification [22].

5 Comparing Multiparty Session Types With **CoMPSeT**

This section presents the **CoMPSeT** tool by describing its applicability, later presenting motivational use cases. All examples referenced throughout this section – among others – are included in the tool [22].

5.1 Running **CoMPSeT**

Each setting in the interface (**CAOS**) corresponds directly to a feature (literature) identified in Table 1. Users configure these settings through associated checkboxes, observing the effects on the widgets described.

Two widgets are kept visible and unchanged regardless of configuration: (1) **Message Sequence Chart**, which offers a graphical representation of the session, and (2) **Global** (type), which displays the session specification via text.

The configurable settings and their consequent impacts are as follows.

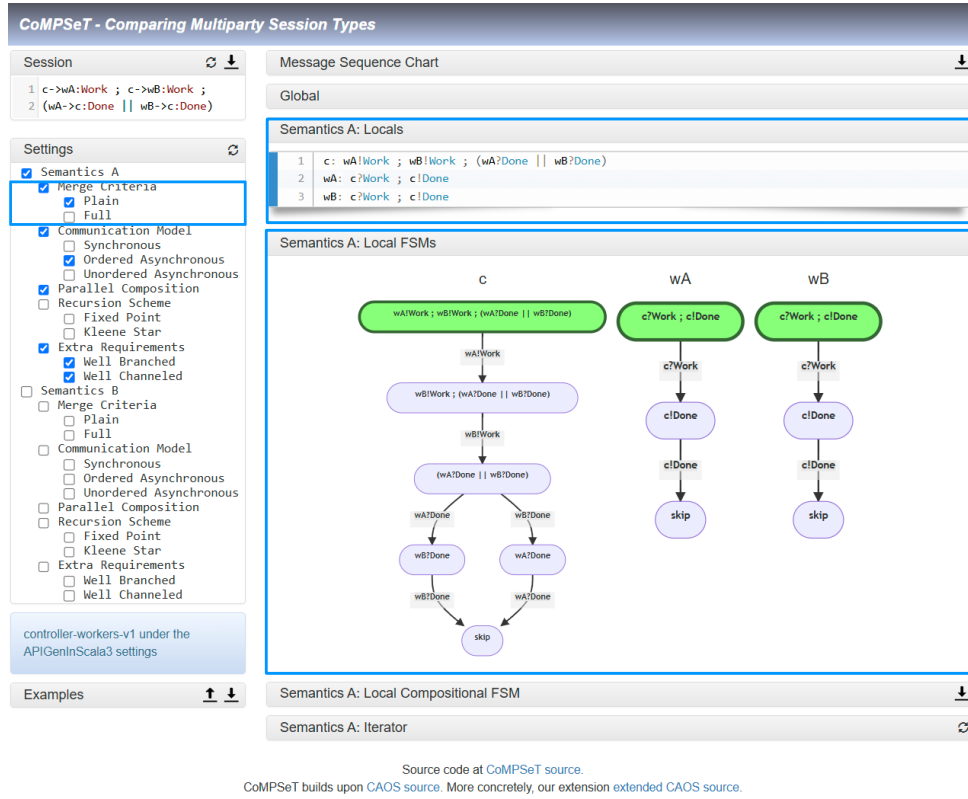


Figure 6: **Locals** and **Local FSMs** for **APIGenInScala3** (base semantics for [5]) – **controller-workers** session from Example 1.1

Merge Criteria This setting controls how projection handles branching interactions for non-communicating participants. CoMPSeT implements the plain merge following Honda et al. [14, 15], and the full merge according to Dezani-Ciancaglini et al. [9]. These determine whether projection is well-defined, manifested through **Locals** – a textual description for the local types – and **Local FSMs** – its graphical counterpart – as exemplified by Fig. 6.

Communication Model Users can select between synchronous, ordered asynchronous, or unordered asynchronous communication models, following their descriptions in Section 2.3. In turn, **Step-by-Step** – an interactive semantic iterator – and **Local Compositional FSM** – a graphical representation of the composed behaviour – will be rendered on the interface in accordance with Fig. 7.

Parallel Composition and Recursion These settings control whether constructs like parallel composition and the supported forms of recursion are allowed. When disabled, sessions containing these operators will raise errors via **Check** – a widget that runs pre-defined conditions over the system while staying invisible if they do not hold – as exemplified by Fig. 8. Enabling the corresponding setting suppresses the error.

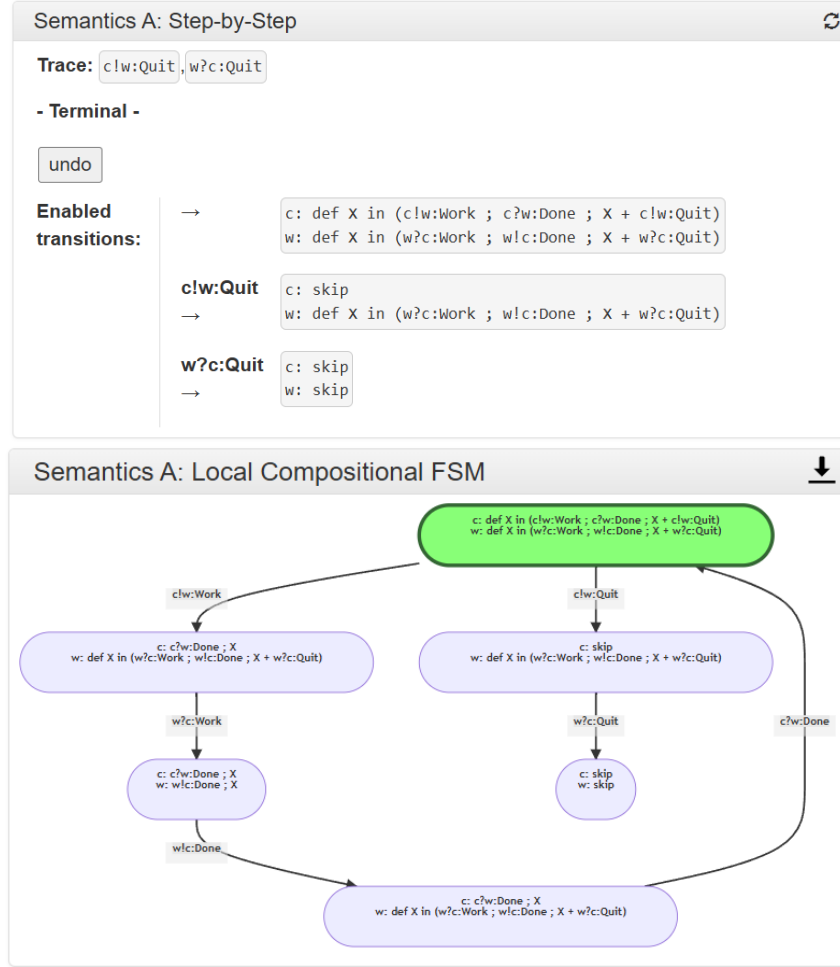


Figure 7: **Step-by-Step** evaluation (top) and **Local Compositional FSM** (bottom) for **GentleIntroMPASyncST** (base semantics for [6]) – recursive **controller-worker** session from Example 2.1

Error raised by 'Semantics A: Locals':
Recursion Kleene Star - present on
participant [c]

Figure 8: **Check** yielding an error describing the presence of Kleene star recursion for **VeryGentleIntroMPST** (base semantics for [25]), using a session described by $(c \rightarrow w : \text{Work} ; w \rightarrow c : \text{Done})^*$

Extra Requirements This setting enables additional well-formedness checks, such as *well-channelled* (see Section 3). Additionally, **CoMPSeT** also enjoys a relaxed form of branching, expressed as $L_A + L_B$ for local types and $G_A + G_B$ for global types. To ensure compatibility with classical MPST, we introduce a well-formedness condition named *well-branched*, verifying whether the relaxed form can be rewritten in the canonical syntax. This decision advances future support for general choreographic languages. These checks do not suppress errors like the previous settings but add extra syntactic validations.

5.2 Comparing Semantics

As a motivational use case, we compare the semantics for **APIGenInScala3** and **ST4MP** (base semantics for [17])⁵ which differ only in their recursion treatment, otherwise aligning closely for sessions not foreseeing this construct. This is evidenced in Fig. 9 (left), where **Bisimulation** is a widget describing their behavioural equivalence.

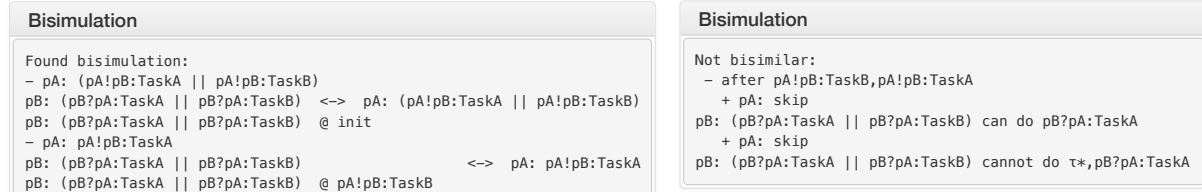


Figure 9: **Bisimulations** comparing the **APIGenInScala3** semantics either against the **ST4MP** semantics (left – only the first seven lines) or against an unordered asynchronous system (right), using a session described by $pA \rightarrow pB : \text{TaskA} \parallel pA \rightarrow pB : \text{TaskB}$

However, the previous behavioural equivalence was partially rooted in both semantics sharing the same communication model. For the same session, a semantic similar with **ST4MP** yet over an unordered asynchronous model would no longer be bisimilar to **APIGenInScala3**, as the new communication model does not enforce rules over message delivery. This nuance is also captured by **CoMPSeT** as illustrated on the right of Fig. 9. Additionally, the graphical representation for the compositional behaviour (see Fig. 10) further evidences their distinction.

As an additional motivational case, we note how the session defined as $(pA \rightarrow pB : \text{TaskA} ; pB \rightarrow pC : \text{TaskA}) + (pA \rightarrow pB : \text{TaskB} ; pB \rightarrow pC : \text{TaskB})$ is well defined under the full merge, but not under the plain merge, as pC would require the same communication action in both branches. This distinction is captured in **CoMPSeT** – exemplified in Fig. 11 – as **VeryGentleIntroMPST**, which adopts the full merge, successfully produces the corresponding local type, whereas **GentleIntroMPASyncST**, which relies on the plain merge, fails to do so.

Readers are encouraged to engage with the examples made accessible online and to configure their own sessions and semantic setups using the presently defined settings.

6 Conclusion and Future Work

This paper introduced **CoMPSeT**, a novel tool designed to compare Multiparty Session Types (MPST) formalisms through dynamic, user-configurable settings, available at <https://telmoribeiro.github.io/CoMPSeT>.

Built atop the **CAOS** framework, **CoMPSeT** leverages its modular widget architecture to define, test, and animate structural operational semantics (SOSs) while exploiting pre-defined notions established by this framework, such as branching bisimulation checkers. Recognising current limitations in the original **CAOS** framework – specifically the lack of support for dynamic widget behaviour and runtime configurability – we extended it to: (1) support runtime variability in widgets; (2) structure configurations through

⁵Notably, in **ST4MP** we adopt the implementation semantics from [17] – with the Kleene star – instead of the original formulation, following the discussion of Section 3 and envisioning greater variability. The formalised version can be experimented upon by selecting instead the *Fixed Point* setting.

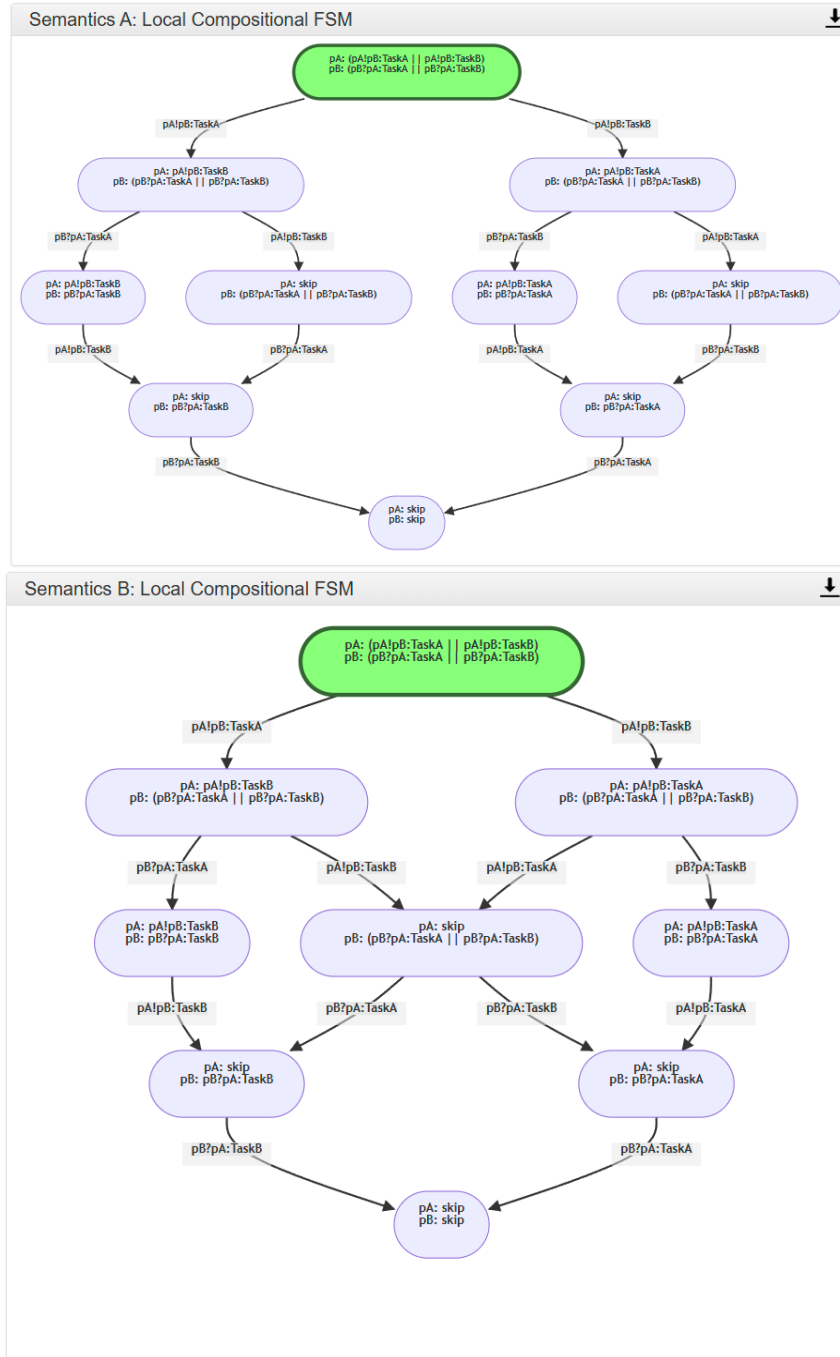





Figure 10: Local Compositional FSM for **APIGenInScala3** (top) and an unordered asynchronous model (bottom) – simple task delegation session from Fig. 9

CoMPSeT - Comparing Multiparty Session Types

Session  

- 1 (pA→pB:TaskA ; pB→pC:TaskA)
- 2 +
- 3 (pA→pB:TaskB ; pB→pC:TaskB)

Error raised by 'Semantics B: Locals':
[Plain Merge] - projection undefined for [pC] in [(pB?pA.TaskA ; pBlpC.TaskA + pB?pA.TaskB ; pBlpC.TaskB)]

Settings 



Semantics A


- ☒ Merge Criteria
 - ☐ Plain
 - ☒ Full
- ☒ Communication Model
 - ☒ Synchronous
 - ☐ Ordered Asynchronous
 - ☐ Unordered Asynchronous
- ☐ Parallel Composition
- ☒ Recursion Scheme
 - ☒ Fixed Point
 - ☐ Kleene Star
- ☒ Extra Requirements
 - ☒ Well Branched
 - ☐ Well Channeled

Semantics B

- ☒ Merge Criteria
 - ☒ Plain
 - ☐ Full
- ☒ Communication Model
 - ☐ Synchronous
 - ☒ Ordered Asynchronous
 - ☐ Unordered Asynchronous
- ☐ Parallel Composition
- ☒ Recursion Scheme
 - ☒ Fixed Point
 - ☐ Kleene Star
- ☒ Extra Requirements
 - ☒ Well Branched
 - ☐ Well Channeled

simple branching - v2 compared for both VeryGentleIntroMPST and GentleIntroMPASyncST

Examples  

Message Sequence Chart 


Global


Bisimulation Checker

Semantics A: Locals

- 1 pA: (pB!TaskA + pB!TaskB)
- 2 pB: (pA?TaskA ; pC!TaskA + pA?TaskB ; pC!TaskB)
- 3 pC: (pB?TaskA + pB?TaskB)


Semantics A: Local FSMs

Semantics A: Local Compositional FSM 

Semantics A: Iterator 

Semantics B: Locals

Semantics B: Local FSMs

Semantics B: Local Compositional FSM 


Semantics B: Iterator 

Figure 11: Locals for VeryGentleIntroMPST (Semantics A) and GentleIntroMPASyncST (Semantics B), using the branching session previously described

a dedicated domain specific language (DSL); and **(3)** establish a concise application programming interface (API) that abstracts the definition of compound widget behaviour and facilitates parametrisation. These extensions are also available as open-source at <https://github.com/TelmoRibeiro/CAOS>.

Building on these extensions, **CoMPSeT** benefits from the new **Settings** widget, enabling users to configure the semantics supporting a session and observe, in practice, how different formalism implementations affect system behaviour. Moreover, it supports side-by-side comparisons of two distinct semantics, providing immediate visual and interactive feedback on their differences. **CoMPSeT** thus enables researchers and educators to visualise, animate, and compare MPST formalisations, helping to clarify subtle semantic variations within aspects such as communication models, recursion schemes, and merge strategies.

By offering detailed automata visualisation, interactive trace exploration, and bisimulation checking, **CoMPSeT** serves both as an exploratory research platform and as a pedagogical tool for teaching concurrent communication sessions.

7 Future Work

Although our tool already supports a range of MPST features, several extensions could broaden its applicability.

- **Additional features** – Further feature assimilation would widen the range of reproducible systems. Following the already included form of relaxed branching and unordered asynchronous communications, supplementary adoption of *choreographic features* would extend the scope of the tool allowing for comparisons between different concurrent communication systems;
- **API generation** – MPST tooling often focuses on application programming interface (API) generation [16, 18, 5, 17, 10], allowing for endpoints implementing different participants to benefit from the guarantees ensured by this typing discipline. **CoMPSeT** could be made a foundational layer in a broader pipeline or incorporate this aspect natively, enabling both semantical comparisons and yielding session compliant APIs.

By continuing to develop **CoMPSeT**, we aim to lower the barriers to understanding MPST theory and make the impact of different semantic choices more transparent and accessible.

References

- [1] Andi Bejleri, Elton Domnori, Malte Viering, Patrick Eugster & Mira Mezini (2019): *Comprehensive Multiparty Session Types*. *Art Sci. Eng. Program.* 3(3), p. 6, doi:10.22152/PROGRAMMING-JOURNAL.ORG/2019/3/6.
- [2] Andi Bejleri & Nobuko Yoshida (2008): *Synchronous Multiparty Session Types*. In Vasco T. Vasconcelos & Nobuko Yoshida, editors: *Proceedings of the First Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES@DisCoTec 2008, Oslo, Norway, June 7, 2008, Electronic Notes in Theoretical Computer Science* 241, Elsevier, pp. 3–33, doi:10.1016/J.ENTCS.2009.06.002.
- [3] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini & Luca Padovani (2012): *On Global Types and Multi-Party Session*. *Log. Methods Comput. Sci.* 8(1), doi:10.2168/LMCS-8(1:24)2012.

- [4] Minas Charalambides, Peter Dinges & Gul Agha (2012): *Parameterized Concurrent Multi-Party Session Types*. In Natallia Kokash & António Ravara, editors: *Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, FOCLASA 2012, Newcastle, U.K., September 8, 2012, EPTCS 91*, pp. 16–30, doi:<https://doi.org/10.4204/EPTCS.91.2>.
- [5] Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans & José Proença (2022): *API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3*. In Karim Ali & Jan Vitek, editors: *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany, LIPIcs 222, Schloss Dagstuhl - Leibniz-Zentrum für Informatik*, pp. 27:1–27:28, doi:10.4230/LIPICS.ECOOP.2022.27.
- [6] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani & Nobuko Yoshida (2015): *A Gentle Introduction to Multiparty Asynchronous Session Types*. In Marco Bernardo & Einar Broch Johnsen, editors: *Formal Methods for Multicore Programming - 15th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2015, Bertinoro, Italy, June 15-19, 2015, Advanced Lectures, Lecture Notes in Computer Science 9104*, Springer, pp. 146–178, doi:10.1007/978-3-319-18941-3_4.
- [7] Pierre-Malo Deniérou & Nobuko Yoshida (2011): *Dynamic multirole session types*. In Thomas Ball & Mooly Sagiv, editors: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, ACM*, pp. 435–446, doi:10.1145/1926385.1926435.
- [8] Pierre-Malo Deniérou, Nobuko Yoshida, Andi Bejleri & Raymond Hu (2012): *Parameterised Multiparty Session Types*. *Log. Methods Comput. Sci.* 8(4), doi:10.2168/LMCS-8(4:6)2012.
- [9] Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic & Nobuko Yoshida (2015): *Precise subtyping for synchronous multiparty sessions*. In Simon Gay & Jade Alglave, editors: *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015, EPTCS 203*, pp. 29–43, doi:<https://doi.org/10.4204/EPTCS.203.3>.
- [10] Francisco Ferreira & Sung-Shik Jongmans (2023): *Oven: Safe and Live Communication Protocols in Scala, using Synthetic Behavioural Type Analysis*. In René Just & Gordon Fraser, editors: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023, ACM*, pp. 1511–1514, doi:10.1145/3597926.3604926.
- [11] Roberto Guanciale & Emilio Tuosto (2018): *Realisability of Pomsets via Communicating Automata*. In Massimo Bartoletti & Sophia Knight, editors: *Proceedings 11th Interaction and Concurrency Experience, ICE 2018, Madrid, Spain, June 20-21, 2018, EPTCS 279*, pp. 37–51, doi:10.4204/EPTCS.279.6.
- [12] Roberto Guanciale & Emilio Tuosto (2019): *Realisability of pomsets*. *J. Log. Algebraic Methods Program.* 108, pp. 69–89, doi:10.1016/J.JLAMP.2019.06.003.
- [13] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings, Lecture Notes in Computer Science 1381*, Springer, pp. 122–138, doi:10.1007/BFB0053567.
- [14] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In George C. Necula & Philip Wadler, editors: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, ACM*, pp. 273–284, doi:10.1145/1328438.1328472.
- [15] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty Asynchronous Session Types*. *J. ACM* 63(1), pp. 9:1–9:67, doi:10.1145/2827695.
- [16] Raymond Hu & Nobuko Yoshida (2016): *Hybrid Session Verification Through Endpoint API Generation*. In Perdita Stevens & Andrzej Wasowski, editors: *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016, Held as Part of the European Joint Conferences on Theory and Practice*

- of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings, Lecture Notes in Computer Science 9633, Springer, pp. 401–418, doi:10.1007/978-3-662-49665-7_24.
- [17] Sung-Shik Jongmans & José Proença (2022): *ST4MP: A Blueprint of Multiparty Session Typing for Multilingual Programming*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISO LA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I*, Lecture Notes in Computer Science 13701, Springer, pp. 460–478, doi:10.1007/978-3-031-19849-6_26.
 - [18] Nicolas Lagaillardie, Rumyana Neykova & Nobuko Yoshida (2020): *Implementing Multiparty Session Types in Rust*. In Simon Bliudze & 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*, Lecture Notes in Computer Science 12134, Springer, pp. 127–136, doi:10.1007/978-3-030-50029-0_8.
 - [19] Simone Orlando, Vairo Di Pasquale, Franco Barbanera, Ivan Lanese & Emilio Tuosto (2021): *Corinne, a Tool for Choreography Automata*. In Gwen Salaün & Anton Wijs, editors: *Formal Aspects of Component Software - 17th International Conference, FACS 2021, Virtual Event, October 28-29, 2021, Proceedings*, Lecture Notes in Computer Science 13077, Springer, pp. 82–92, doi:10.1007/978-3-030-90636-8_5.
 - [20] José Proença & Luc Edixhoven (2023): *Caos: A Reusable Scala Web Animator of Operational Semantics*. In Sung-Shik Jongmans & Antónia Lopes, editors: *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*, Lecture Notes in Computer Science 13908, Springer, pp. 163–171, doi:10.1007/978-3-031-35361-1_9.
 - [21] José Proença & Luc Edixhoven (2025): *The CAOS framework for Scala: Computer-aided design of SOS*. *Sci. Comput. Program.* 240, p. 103222, doi:10.1016/J.SCICO.2024.103222.
 - [22] Telmo Ribeiro, José Proença & Mário Florido (2025): *CoMPSeT: A Framework for Comparing Multiparty Session Types (Extended Version)*, doi:https://doi.org/10.5281/zenodo.17188157.
 - [23] Alceste Scalas & Nobuko Yoshida (2019): *Less is more: multiparty session types revisited*. *Proc. ACM Program. Lang.* 3(POPL), pp. 30:1–30:29, doi:10.1145/3290343.
 - [24] Nobuko Yoshida, Pierre-Malo Deniérou, Andi Bejleri & Raymond Hu (2010): *Parameterised Multiparty Session Types*. In C.-H. Luke Ong, editor: *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, Lecture Notes in Computer Science 6014, Springer, pp. 128–145, doi:10.1007/978-3-642-12032-9_10.
 - [25] Nobuko Yoshida & Lorenzo Gheri (2020): *A Very Gentle Introduction to Multiparty Session Types*. In Dang Van Hung & Meenakshi D’Souza, editors: *Distributed Computing and Internet Technology - 16th International Conference, ICDCIT 2020, Bhubaneswar, India, January 9-12, 2020, Proceedings*, Lecture Notes in Computer Science 11969, Springer, pp. 73–93, doi:10.1007/978-3-030-36987-3_5.