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* denotes 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.

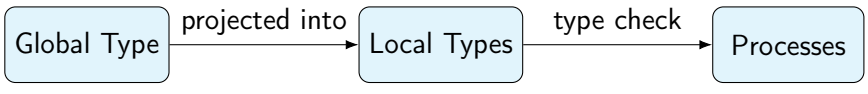

Figure 1: The classical Multiparty Session Types' framework

Submitted to:
EXPRESS/SOS 2025

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,* $\mathsf{worker_A}$ *and* $\mathsf{worker_B}$*, in this particular order. The workers are then expected to reply with a completion message (*Done*) in any order.*

$$\mathsf{controller} \to \mathsf{worker_A} : \mathsf{Work}\,;\mathsf{controller} \to \mathsf{worker_B} : \mathsf{Work};$$
$$(\mathsf{worker_A} \to \mathsf{controller} : \mathsf{Done} \,\|\, \mathsf{worker_B} \to \mathsf{controller} : \mathsf{Done})$$

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* $\mathsf{worker_{AB}}$ *is used to reference both* $\mathsf{worker_A}$ *and* $\mathsf{worker_B}$*, as they share an identical structure.*

$$L_{\mathsf{controller}} = \mathsf{worker_A}!\mathsf{Work}\,;\mathsf{worker_B}!\mathsf{Work}\,;(\mathsf{worker_A}?\mathsf{Done} \,\|\, \mathsf{worker_B}?\mathsf{Done})$$
$$L_{\mathsf{worker_{AB}}} = \mathsf{controller}?\mathsf{Work}\,;\mathsf{controller}!\mathsf{Done}$$

Formally, let G be a well-formed global type involving participants $\mathsf{p_1}, \ldots, \mathsf{p_n}$. If, for each $1 \leq i \leq n$, there exists a process P_i such that $\vdash P_i : (G\!\upharpoonright_{\mathsf{p_i}})$ – where $\upharpoonright$ denotes the projection of G onto participant $\mathsf{p_i}$ – then the composed concurrent system $(P_1 \mid \ldots \mid P_n)$ 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.

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

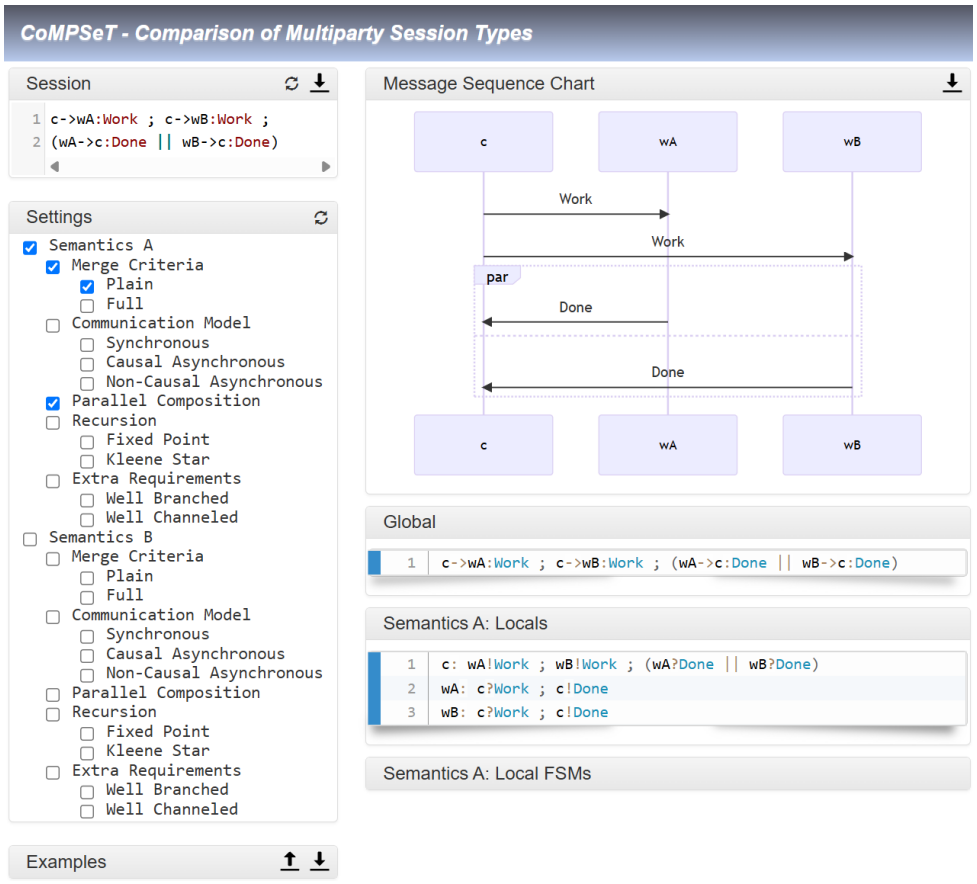

Figure 2: CoMPSeT representations for Example 1.1 and Example 1.2, where controller, worker_A, and worker_B are abbreviated as c, wA, and wB, respectively

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://telmoribeiro.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 – CoMPSeT.

2.1 Global Types

Let $\mathbb{P}$ be the set of all *participants*, ranged over by $\mathtt{p}$, $\mathtt{q}$, $\mathtt{r}$; let $\mathbb{T}$ be the set of all *data types*, ranged over by $\mathtt{t}$; 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.

$$G ::= \mathtt{p} \to \mathtt{q} : \{\mathtt{t}_i\,; G_i\}_{1 \leq i \leq n} \mid G_1\,; G_2 \mid G_1 \parallel G_2 \mid \mu X.G \mid X \mid (G)^* \mid \mathit{skip}$$

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.

- $\mathtt{p} \to \mathtt{q} : \{\mathtt{t}_i\,; G_i\}_{1 \leq i \leq n}$ specifies the *communication* of a label $\mathtt{t}_i$ from the participant $\mathtt{p}$ to the participant $\mathtt{q}$, followed by the global type G_i, for some $1 \leq i \leq n$. As an additional well-formedness requirement, we stipulate that **(1)** $\mathtt{p} \neq \mathtt{q}$ (i.e., no self-communications) and **(2)** the labels in $\mathtt{t}_i$ must be pairwise distinct (i.e., deterministic continuations). Moreover, we write $\mathtt{p} \to \mathtt{q} : \{\mathtt{t}_i\}_{1 \leq i \leq n}\,; G$ as a shorthand for $\mathtt{p} \to \mathtt{q} : \{\mathtt{t}_i\,; G\}_{1 \leq i \leq n}$.
- $G_1\,; G_2$ 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].

Remarks on the syntax

We make the assumption that merging a single local type 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 $L_{\mathsf{controller}}$ (see Example 1.2).
When that is not possible, the notation for communication actions may be explicited 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, 22, 6, 24]. 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.

Example 2.1. *The global type below captures a session where the controller delegates a task that the worker needs to execute promptly, and which may be replicated afterwards.*

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

2.2 Local Types & Projections

Local types are defined by the following grammar.

$$L ::= \mathsf{pq}!\{\mathsf{t_i}\,;L_i\}_{1\leq i\leq n} \mid \mathsf{pq}?\{\mathsf{t_i}\,;L_i\}_{1\leq i\leq n} \mid L_1\,;L_2 \mid L_1 \parallel L_2 \mid \mu X.L \mid X \mid (L)^* \mid \mathit{skip}$$

The informal meaning of the local types is such that:

- $\mathsf{pq}!\{\mathsf{t_i}\,;L_i\}_{1\leq i\leq n}$ specifies a *sending* of a label $\mathsf{t_i}$ from the participant p to the participant q, followed by the local type L_i, for some $1 \leq i \leq n$. Moreover, we write $\mathsf{pq}!\{\mathsf{t_i}\}_{1\leq i\leq n}\,;L$ as a shorthand for $\mathsf{pq}!\{\mathsf{t_i}\,;L\}_{1\leq i\leq n}$.
- $\mathsf{pq}?\{\mathsf{t_i}\,;L_i\}_{1\leq i\leq n}$ specifies the *reception* of a label $\mathsf{t_i}$ expected by the participant q from the participant p, followed by the local type L_i, for some $1 \leq i \leq n$. Moreover, we write $\mathsf{pq}?\{\mathsf{t_i}\}_{1\leq i\leq n}\,;L$ as a shorthand for $\mathsf{pq}?\{\mathsf{t_i}\,;L\}_{1\leq i\leq n}$.
- 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 [24].

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.

$$
\begin{array}{ll}
skip\upharpoonright_{\mathsf{r}} = skip & \\
X\upharpoonright_{\mathsf{r}} = X & \\
(\mu X.G)\upharpoonright_{\mathsf{r}} = \mu X.(G\upharpoonright_{\mathsf{r}}) & \text{if } \mathsf{r} \in participants\{G\} \\
(\mu X.G)\upharpoonright_{\mathsf{r}} = skip & \text{if } \mathsf{r} \notin participants\{G\} \\
(G)^*\upharpoonright_{\mathsf{r}} = (G\upharpoonright_{\mathsf{r}})^* & \text{if } \mathsf{r} \in participants\{G\}^2 \\
(G)^*\upharpoonright_{\mathsf{r}} = skip & \text{if } \mathsf{r} \notin participants\{G\} \\
\mathsf{p} \to \mathsf{q} : \{\mathsf{t}_i\,; G_i\}_{1\le i\le n}\upharpoonright_{\mathsf{r}} = \mathsf{pq}!\{\mathsf{t}_i\,; (G_i\upharpoonright_{\mathsf{r}})\}_{1\le i\le n} & \text{if } \mathsf{p} = \mathsf{r} \neq \mathsf{q} \\
\mathsf{p} \to \mathsf{q} : \{\mathsf{t}_i\,; G_i\}_{1\le i\le n}\upharpoonright_{\mathsf{r}} = \mathsf{pq}?\{\mathsf{t}_i\,; (G_i\upharpoonright_{\mathsf{r}})\}_{1\le i\le n} & \text{if } \mathsf{p} \neq \mathsf{r} = \mathsf{q} \\
\mathsf{p} \to \mathsf{q} : \{\mathsf{t}_i\,; G_1\}_{1\le i\le n}\upharpoonright_{\mathsf{r}} = merge(\{G_i\upharpoonright_{\mathsf{r}}\}_{1\le i\le n}) & \text{if } \mathsf{p} \neq \mathsf{r} \neq \mathsf{q} \\
(G_1\,; G_2)\upharpoonright_{\mathsf{r}} = (G_1\upharpoonright_{\mathsf{r}})\,; (G_2\upharpoonright_{\mathsf{r}}) & \\
(G_1 \parallel G_2)\upharpoonright_{\mathsf{r}} = (G_1\upharpoonright_{\mathsf{r}}) \parallel (G_2\upharpoonright_{\mathsf{r}}) & \\
\text{undefined} & \text{otherwise}
\end{array}
$$

Figure 3: Projection $G\upharpoonright_{\mathsf{r}}$ of a global type G 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 $(\mathsf{a} \to \mathsf{b} : \mathsf{m}\,; \mathsf{b} \to \mathsf{c} : \mathsf{m}')^*\,; \mathsf{c} \to \mathsf{d} : \mathsf{m}''$ 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.*

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

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

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

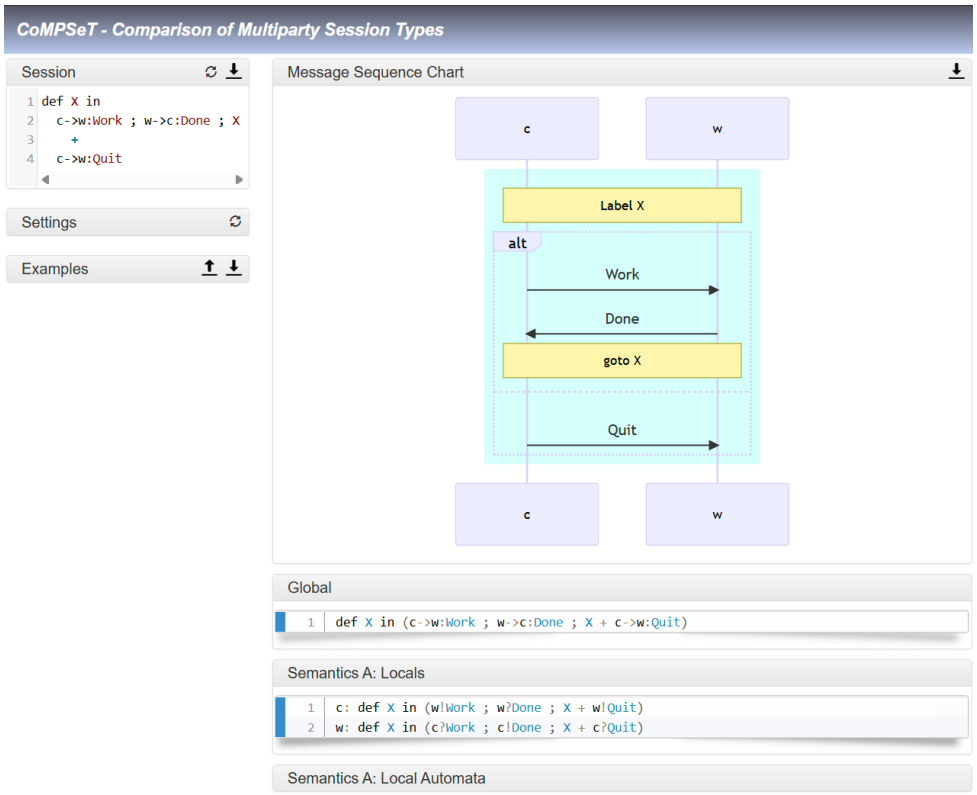

Figure 4: CoMPSeT representations for Example 2.1 and Example 2.2, where controller and worker are abbreviated as c and w, respectively

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 $(L_{\mathsf{p}_1} \mid \cdots \mid L_{\mathsf{p}_n})$, where each L_{p} is a well-formed local type for some participant $\mathsf{p} \in M$. This assumption is based on the following considerations: **(1)** local types can mimic the evolution of well-typed processes, and **(2)** the employment of semantics over projections 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 exact data structure 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 concrete 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 $env : X \to L$ – 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 Appendix A and Appendix B, respectively.

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

$$\langle \mathsf{pq}!\{\mathsf{t_i}\,;L_{1_i}\}_{1\le i\le m_i} \mid \mathsf{pq}?\{\mathsf{t_j}\,;L_{2_j}\}_{1\le j\le m_j} \mid M\rangle \xrightarrow{\mathsf{p}\to\mathsf{q}:\mathsf{t_k}} \langle L_{1_k} \mid L_{2_k} \mid M\rangle \qquad \text{(communication)}$$

Causal Asynchronous Semantics Here, each pair of participants represented simply by pq has an unbounded FIFO queue for messages. A configuration is established by a Multiparty Session M and a buffer $p : (\mathbb{P}\times\mathbb{P}) \to \mathbb{T}^*$, 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 $\mathsf{t_i}\cdot ts$ is used to highlight the prefix element while conversely, $ts\cdot\mathsf{t_i}$ emphasizes the suffix.

$$\langle \mathsf{pq}!\{\mathsf{t_i}\,;L_i\}_{1\le i\le m} \mid M\,,\, p\cup\{\mathsf{pq}\mapsto ts\}\rangle \xrightarrow{\mathsf{pq}!\mathsf{t_k}} \langle L_k \mid M\,,\, p\cup\{\mathsf{pq}\mapsto ts\cdot\mathsf{t_k}\}\rangle \qquad \text{(send)}$$

$$\langle \mathsf{pq}?\{\mathsf{t_i}\,;L_i\}_{1\le i\le m} \mid M\,,\, p\cup\{\mathsf{pq}\mapsto \mathsf{t_k}\cdot ts\}\rangle \xrightarrow{\mathsf{pq}?\mathsf{t_k}} \langle L_k \mid M\,,\, p\cup\{\mathsf{pq}\mapsto ts\}\rangle \qquad \text{(receive)}$$

Non-Causal Asynchronous Semantics Here the configurations are established as pairs of Multiparty Sessions M and a multiset $p \in \mathscr{M}(\mathbb{P}\times\mathbb{P}\times\mathbb{T})$, where $\mathscr{M}(X)$ 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 $p\cup(\mathsf{p},\mathsf{q},\mathsf{t_i})$ a new multiset achieved by joining the tuple comprising p,q, and $\mathsf{t_i}$ to an existing multiset p.

$$\langle \mathsf{pq}!\{\mathsf{t_i}\,;L_i\}_{1\le i\le m} \mid M\,,\, p\rangle \xrightarrow{\mathsf{pq}!\mathsf{t_k}} \langle L_k \mid M\,,\, p\cup(\mathsf{p},\mathsf{q},\mathsf{t_k})\rangle \qquad \text{(send)}$$

$$\langle \mathsf{pq}?\{\mathsf{t_i}\,;L_i\}_{1\le i\le m} \mid M\,,\, p\cup(\mathsf{p},\mathsf{q},\mathsf{t_k})\rangle \xrightarrow{\mathsf{pq}?\mathsf{t_k}} \langle L_k \mid M\,,\, p\rangle \qquad \text{(receive)}$$

While *non-causal 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 [22], 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.[23, 8] – extends this notion by allowing distinct yet *compatible* [22, 24] 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)** *causal asynchronous* with an unbounded FIFO queue for pending communications; and **(3)** *non-causal 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 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: $comm(G_1) \cap comm(G_2) = \emptyset$. Here, $comm : \mathbb{G} \to 2^{\mathbb{P}\times\mathbb{P}\times\mathbb{T}}$ 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 **[24]**, **[6]**, **[5]**, and **[17]** derives partially from their role as major references while establishing our own formalisms. The introductory nature of **[24]** 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 non-causal 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	Well-formedness requirements
[24]	plain & full	synchronous	×	fixed point	
[6]	plain	causal asynchronous	×	fixed point	
[5]	plain	causal asynchronous	✓	×	well-channelled
[17]	plain	causal asynchronous	✓	**Kleene star** & fixed point	well-channelled
[11]	N/S	non-causal 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: *lts* – 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

[3]Notation 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.

[4]Given 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 becomes apparent in the manner widgets are defined, stopping CAOS from dynamically modifying them.

```
/** Main widgets, on the right hand side of the screen*/
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 (see Appendix C); 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 (see Appendix D).

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 (see Appendix E and Appendix F).

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 (see Appendix G).

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.

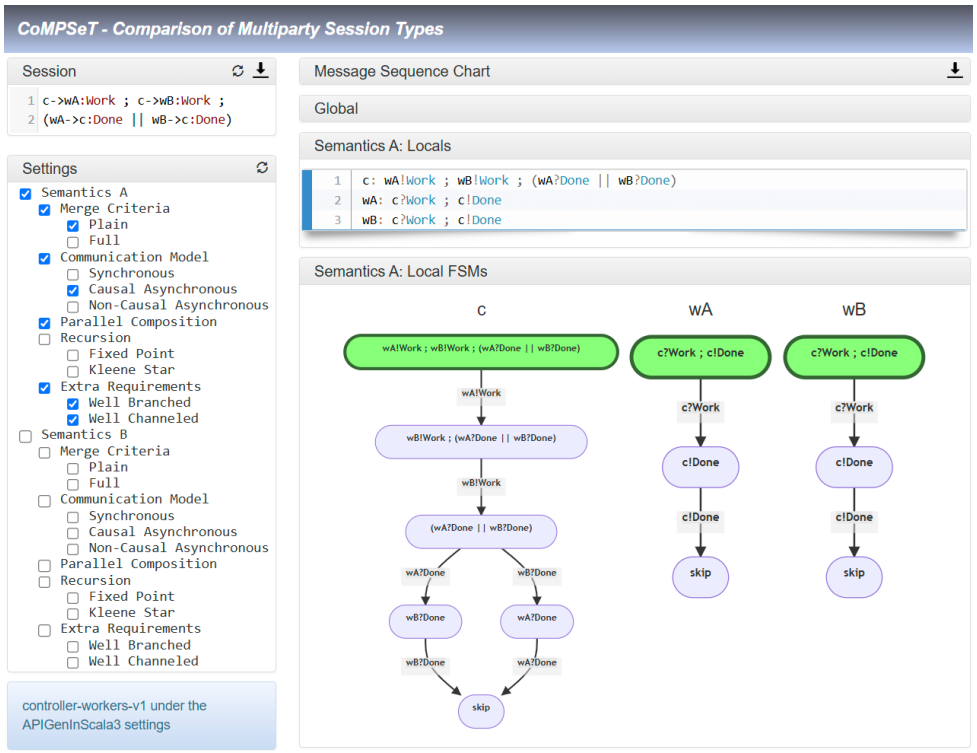

Figure 6: Locals and Local FSMs for APIGenInScala3 (base semantics for [5]) – controller-workers session from Example 1.1

The configurable settings and their consequent impacts are as follows.

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, causal asynchronous, or non-causal 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.

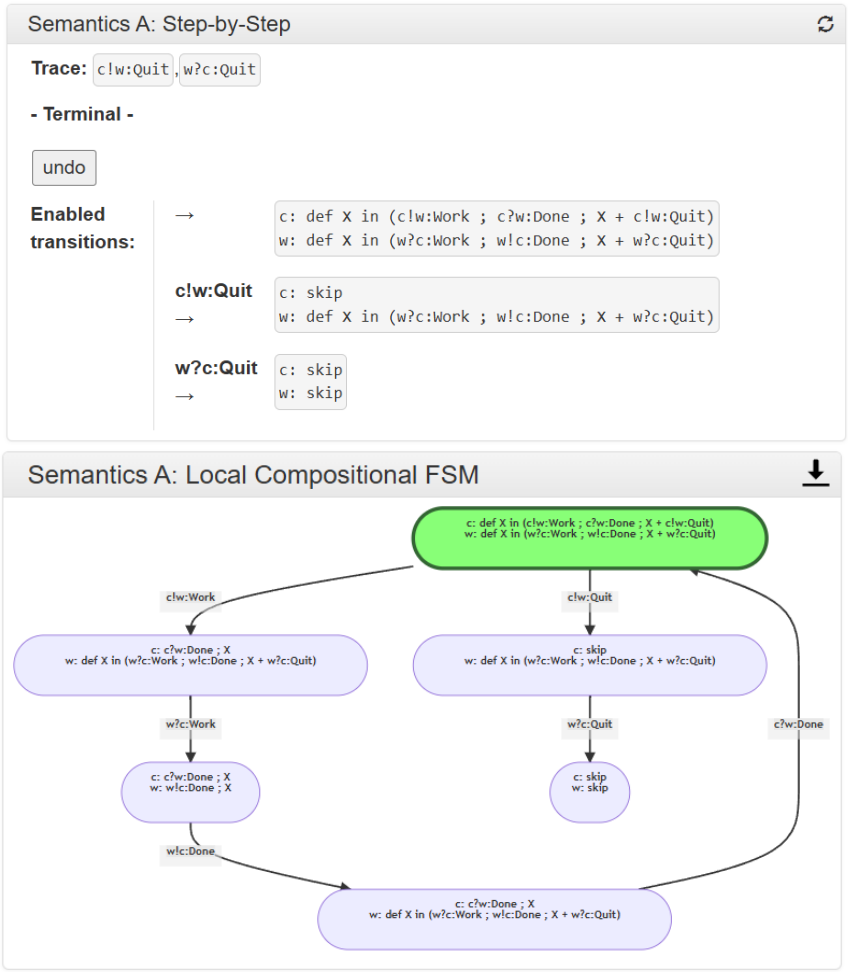

Figure 7: Step-by-Step evaluation (top) and Local Compositional FSM (bottom) for GentleIntroMPAsyncST (base semantics for [6]) – recursive controller-worker session from Example 2.1

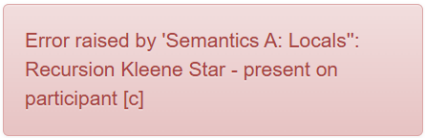

Figure 8: Check yielding an error describing the presence of Kleene star recursion for VeryGentleIntroMPST (base semantics for [24]), using a session described by $(\mathsf{c} \to \mathsf{w} : \mathsf{Work}\,;\mathsf{w} \to \mathsf{c} : \mathsf{Done})^*$

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.

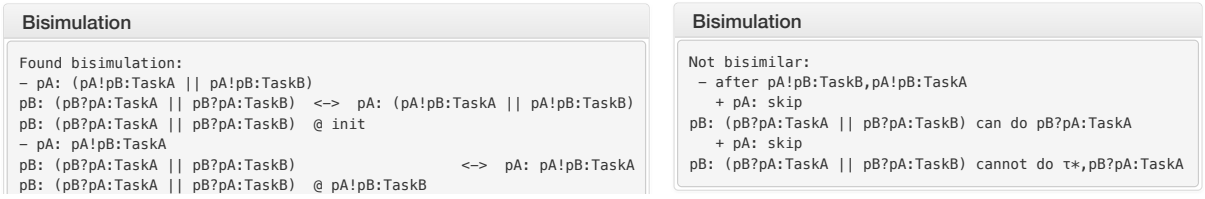

Figure 9: Bisimulations comparing the APIGenInScala3 semantics either against the ST4MP semantics (left – only the first seven lines) or against a non-causal asynchronous system (right), using a session described by $\mathsf{pA} \to \mathsf{pB} : \mathsf{TaskA} \parallel \mathsf{pA} \to \mathsf{pB} : \mathsf{TaskB}$

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 a non-causal 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 $(\mathsf{pA} \to \mathsf{pB} : \mathsf{TaskA}\,;\mathsf{pB} \to \mathsf{pC} : \mathsf{TaskA}) + (\mathsf{pA} \to \mathsf{pB} : \mathsf{TaskB}\,;\mathsf{pB} \to \mathsf{pC} : \mathsf{TaskB})$ 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

[5]Notably, 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.

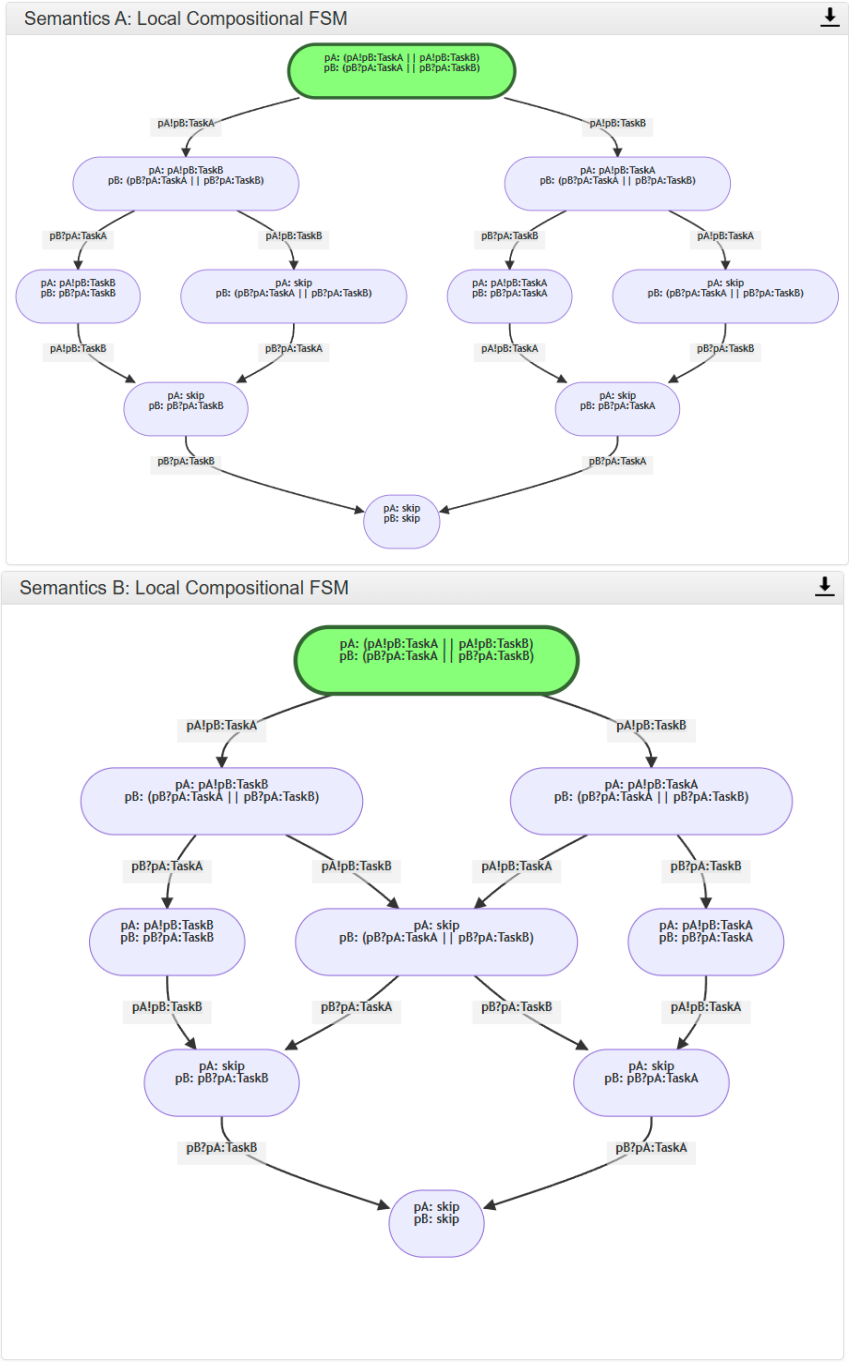

Figure 10: Local Compositional FSM for APIGenInScala3 (top) and a non-causal asynchronous model (bottom) – simple task delegation session from Fig. 9

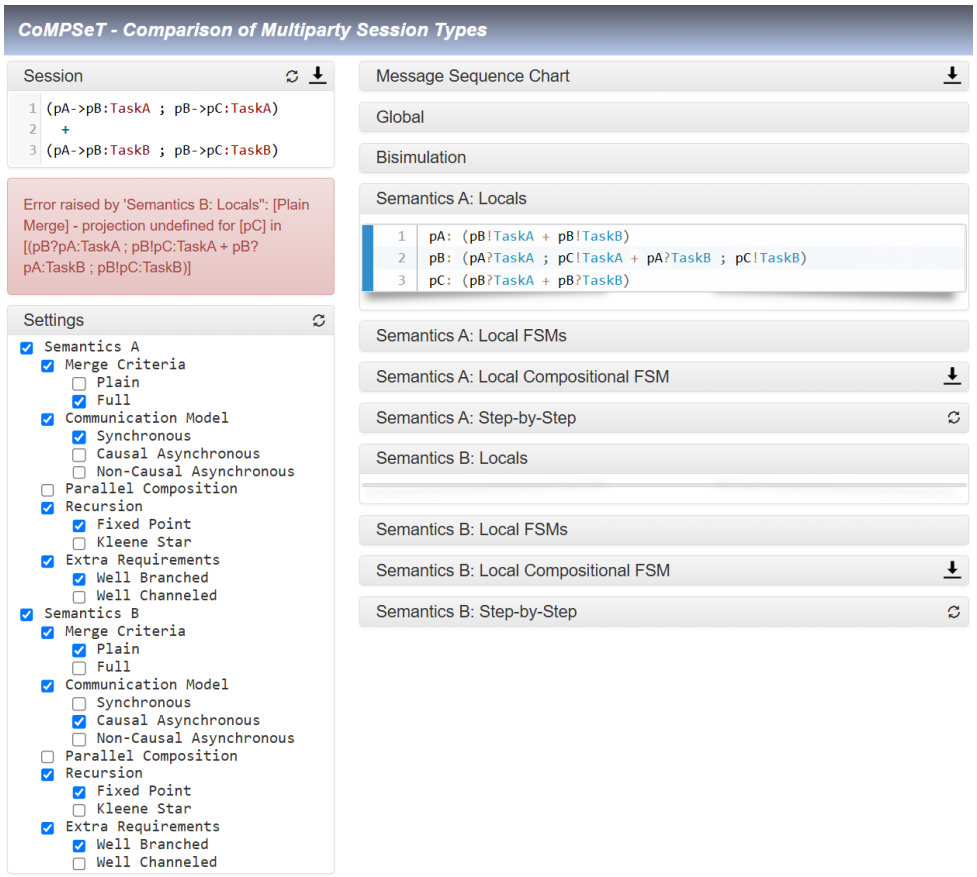

Figure 11: Locals for VeryGentleIntroMPST (Semantics A) and GentleIntroMPAsyncST (Semantics B), using the branching session previously described

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 inclusion of relaxed branching communications and a non-causal asynchronous communication model, 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:10.4204/EPTCS.91.2. Available at `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 Deniélou & 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 Deniélou, 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:10.4204/EPTCS.203.3. Available at `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, ISoLA 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] 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.

[23] Nobuko Yoshida, Pierre-Malo Deniélou, 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.

[24] 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.

A Structural Congruence

We assume the structural congruence relation shown below, adapted from Yoshida and Gheri [24], while incorporating additional rules targeting local types.

$$
\begin{array}{l}
skip\,;L \equiv L \quad L\,;skip \equiv L \quad skip \parallel L \equiv L \quad L \parallel skip \equiv L \\
(L\,;L')\,;L'' \equiv L\,;(L'\,;L'') \quad (L \parallel L') \parallel L'' \equiv L \parallel (L' \parallel L'') \\
L \equiv L' \Rightarrow L \mid M \equiv L' \mid M \\
M \mid M' \equiv M' \mid M \quad (M \mid M') \mid M'' \equiv M \mid (M' \mid M'') \\
\mu X.L \equiv L \text{ if } X \notin fv(L)
\end{array}
$$

Here, fv denotes the set of free variables within a local type.

All operational rules presented in this section are defined modulo structural congruence.

B Shared Reduction Rules

We define *Multiparty Sessions*, *configurations*, and *pending collections* in accordance with Section 2.3.

Additionally, we use α to range over both communication interactions of the form $\mathsf{p} \to \mathsf{q} : \mathsf{t_k}$, or actions such as $\mathsf{pq}!\mathsf{t_k}$ and $\mathsf{pq}?\mathsf{t_k}$, for $1 \leq k \leq m$ where m is the number of branching data types.

We consider the shared reduction rules that follow.

$$\frac{\langle L_1,\ p\rangle \xrightarrow{\alpha} \langle L_1',\ p'\rangle}{\langle L_1\,;L_2 \mid M,\ p\rangle \xrightarrow{\alpha} \langle L_1'\,;L_2 \mid M,\ p'\rangle} \quad \text{(SEQUENCE-LEFT)}$$

$$\frac{canskip(L_1) \wedge \langle L_2,\ p\rangle \xrightarrow{\alpha} \langle L_2',\ p'\rangle}{\langle L_1\,;L_2 \mid M,\ p\rangle \xrightarrow{\alpha} \langle L_2' \mid M,\ p'\rangle} \quad \text{(SEQUENCE-RIGHT)}$$

$$\frac{\langle L_1,\ p\rangle \xrightarrow{\alpha} \langle L_1',\ p'\rangle}{\langle L_1 \parallel L_2 \mid M,\ p\rangle \xrightarrow{\alpha} \langle L_1' \parallel L_2 \mid M,\ p'\rangle} \quad \text{(PARALLEL-LEFT)}$$

$$\frac{canskip(L_2) \wedge \langle L_1,\ p\rangle \xrightarrow{\alpha} \langle L_1',\ p'\rangle}{\langle L_1 \parallel L_2 \mid M,\ p\rangle \xrightarrow{\alpha} \langle L_1' \mid M,\ p'\rangle} \quad \text{(PARALLEL-LEFT W/ SKIP)}$$

$$\frac{\langle L_2,\ p\rangle \xrightarrow{\alpha} \langle L_2',\ p'\rangle}{\langle L_1 \parallel L_2 \mid M,\ p\rangle \xrightarrow{\alpha} \langle L_1 \parallel L_2' \mid M,\ p'\rangle} \quad \text{(PARALLEL-RIGHT)}$$

$$\frac{canskip(L_1) \wedge \langle L_2,\ p\rangle \xrightarrow{\alpha} \langle L_2',\ p'\rangle}{\langle L_1 \parallel L_2 \mid M,\ p\rangle \xrightarrow{\alpha} \langle L_2' \mid M,\ p'\rangle} \quad \text{(PARALLEL-RIGHT W/ SKIP)}$$

$$\frac{\langle L,\ p\rangle \xrightarrow{\alpha} \langle L',\ p'\rangle}{\langle L^* \mid M,\ p\rangle \xrightarrow{\alpha} \langle L' ; L^* \mid M,\ p'\rangle} \quad \text{(KLEENE STAR)}$$

$$\frac{\langle L[\mu X.L/X],\ p\rangle \xrightarrow{\alpha} \langle L',\ p'\rangle}{\langle \mu X.L \mid M,\ p\rangle \xrightarrow{\alpha} \langle L' \mid M,\ p'\rangle} \quad \text{(FIXED POINT)}$$

We say *canskip* holds if L can immediately reduce to *skip* without performing any communication. For instance, the *skip* itself, $skip ; skip$, and L^* always satisfy *canskip*.

Remarks on semantical validation

The semantics presented adapt reduction rules from works like Daniélou and Yoshida [7], Jongmans and Proença [17], Yoshida and Gheri [24], among others, aiming for a minimal, modular set applicable across different communication models.
These semantics have been validated through empirical testing as they stand extracted from the current implementation in CoMPSeT. Although formal soundness and completeness proofs are still lacking, we notice that our tool offers semantic animation and visualisation features, helping users detect inconsistencies. Additionally, CoMPSeT includes several examples hinting towards the well-behavedness of the semantics.

C Specifying Settings

The core building block of the new Settings extension is an instance of the case class *Setting*.

```
1  case class Setting(name: String = null,
2                     children: List[Setting] = List.empty,
3                     checked: Boolean = false,
4                     options: List[String] = List.empty)
```

Figure 12: *Setting* signature in CAOS

As shown in the Fig. 12, each *Setting* has: a *name*, used both as an internal reference to the structure and as a display label in the user interface (UI); a list of *children*, which are themselves *Setting* instances; a *checked* flag indicating whether the setting is currently enabled; and a list of *options*, which allows attaching tags (as strings) to account for categories or properties.

The *options* field allows the programmer to introduce extra configurability to her project. For example, our extension comes with two predefined tags: *allowOne* and *allowAll*. These determine whether at most one child – exclusive choice – or any number of children – multiple choice – can be selected within a given branch.

To define Settings concisely, we developed a lightweight DSL. It allows developers to declare structured settings in a declarative style. This structure is rendered in the UI through a dedicated *SettingWidget*, following the design of the other input widgets. Once rendered, users interact with it via checkboxes, which reflect and update the *checked* fields of individual nodes. On the developer's side, builders such as *steps* or *lts* can be conditionally defined based on the state of Settings. This enables reactive behaviour where, for example, a widget may be omitted if certain parameters are disabled (unchecked), resulting in it not being rendered.

In Fig. 13, the operators || and *&&* build groups of sibling settings, with implied semantics: the parent node will be tagged with either *allowOne* or *allowAll*, respectively. The -> operator renames a subtree, assigning the label on the left-hand side to the root node of the tree produced on the right.

```
    private def mkSemantics: Setting =
        "Merge" ->
            ("Plain" || "Full")
        && "Communication Model" ->
            ("Sync" || "Causal Async" || "Non-Causal Async")
        && "Recursion" ->
            ("Kleene Star" || "Fixed Point")
        && "Parallel"
        && "Extra Requirements" ->
            ("Well Branched" && "Well Channeled")
```

Figure 13: Builder for a semantic branch in CoMPSeT established through the new DSL

D Specifying Configurable Widgets

Building on the proposed extensions, the programmer can now define a configurable structure that users influence via checkboxes in the UI. These widgets can dynamically adjust their behaviour at runtime, including being conditionally hidden.

This pattern assumes the presence of *getters* – methods that retrieve the current state of a data structure – and possibly *setters* – methods that modify it – defined for *Setting*. This allows the programmer to, for instance, pattern-match on its values and adjust widget behaviour accordingly. However, relying solely on these two operations places the burden on the developer to define compound behaviour.

To better support this flexibility, we defined a *Setting* API that includes not only methods for accessing and updating nodes, but also general-purpose filters and auxiliary functions.

```
    def allFromInclusive(setting: Setting,
                         filterCondition: Setting => Boolean = _ => true
                         ): Set[Setting] =
        val filteredSetting = if filterCondition(setting)
                              then Set(setting)
                              else Set.empty
        filteredSetting ++ setting.children.flatMap {
            allFromInclusive(_, filterCondition)
        }
```

Figure 14: Modular filtering method from new new API

Fig. 14 shows one of the methods present in the API, which returns all settings reachable from a given root (including itself) that satisfies *filterCondition*. By default, the condition holds for any *Setting*, meaning the method returns the entire subtree when the programmer does not supply it with a condition of her own. This method serves as a building block for defining other meaningful queries. For instance, by specifying a condition that returns the *checked* field of each setting, the method will yield all settings toggled from a given root. Conversely, a condition checking that *children* is empty yields all leaves attainable from the root. By verifying both conditions, we obtain all checked leaves, which is particularly useful in scenarios where the internal nodes serve purely as labels and leaves encode actual decisions, a pattern shared by CoMPSeT. All the previous scenarios are contemplated as methods supplied by the API.

E CoMPSeT Structure

Fig. 15 presents the high-level organisation of the source code implementing CoMPSeT. Rather than exposing every class and object, the diagram abstracts most of the implementation details to highlight the different modules and their purposes.

The design supports widget parametrisation, in line with our goals of configurability. It includes independent modules for syntax definition, projection, operational semantics, among others. At the core lies the *frontend* module, acting as an interface between CoMPSeT-specific logic and CAOS.

```
compset/
├── frontend/
│   ├── auxiliary/ ..... auxiliary notions for the frontend
│   │   ├── Examples.scala
│   │   └── Widgets.scala
│   ├── CoMPSeTConfigurator.scala
│   └── Main.scala
├── operational-semantics/ ..... objects describing the different operational semantics
├── projections/ ..... standard projection rules and merge criteria implementations
├── syntax/ ..... language parser and structural congruence rules
├── utility/ ..... implementations for recursive environments and multisets
└── well-fromedness/ ..... modular well-formedness conditions
```

Figure 15: CoMPSeT directory structure

F CoMPSeT Implementation

```
1  private def mkStepsByStep(root: String): Option[WidgetInfo[Global]] =
2      Option.when(enabledCommunicationModels(root).nonEmpty) {
3          val (semantics, initialState, showState) =
4              getArguments(root, enabledCommunicationModels(root).head)
5          steps(initialState, semantics, showState)
6      }
7  end mkStepsByStep
```

Figure 16: A builder for an interactive step-by-step widget in CoMPSeT

Fig. 16 exemplifies – through a concrete widget definition in CoMPSeT– a builder for widgets that describe the interactive exploration of semantic execution traces. The challenge lies in ensuring that the widget complies with the parameters selected by the user, while remaining hidden when no suitable configuration is chosen.

This definition is conditionally described using *Option.when*, meaning the widget is only defined when at least one communication model is selected by the user. Additionally, the function *getArguments* abstracts the algorithm that involves filtering for checked leaves from the root – following the logic contemplated in Fig. 14 – and establishing the necessary arguments for *steps* under the chosen communication model: the initial configuration state, the object defining the operational semantics, and a pretty-printer for configuration visualisation.

This structure generalises across all widget types. By filtering the checked leaves in Settings – through methods defined in the new API – and subsequently pattern-matching on them, CoMPSeT identifies the selected feature and extracts the appropriate arguments for the native builder.

G Available Examples

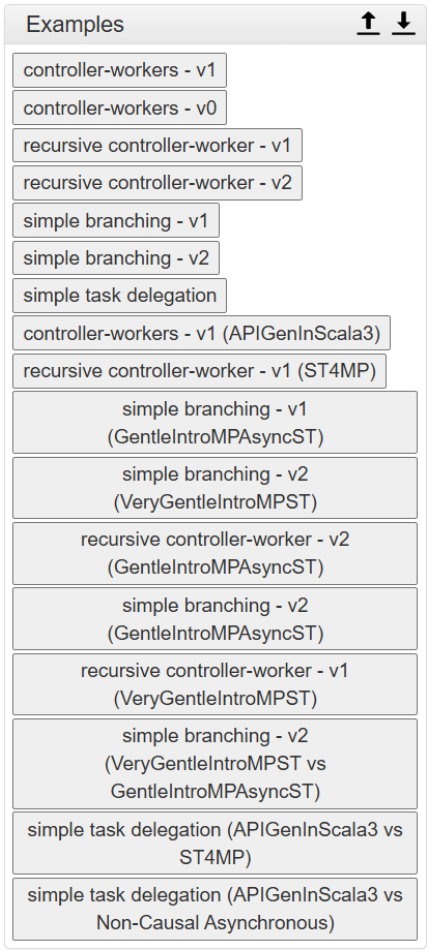

Figure 17: Interface view of the available example set in CoMPSeT

Fig. 17 displays the examples tab of CoMPSeT, showcasing the predefined set of session descriptions currently included with the tool. These examples were selected to cover a broad range of MPST features and variations in semantic configurations.

The available examples are organized into three groups:

- The first seven examples define simple yet meaningful sessions without predefined semantics, allowing users to experiment with different configurations.
- The next seven examples associate the previous sessions with specific base semantics, described in Table 1.

- The final three examples feature sessions being compared under distinct semantic settings, highlighting CoMPSeT's capability to perform side-by-side semantic comparisons.