# API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3

**Guillermina Cledou**
HASLab, INESC TEC, Porto, Portugal
University of Minho, Braga, Portugal

**Luc Edixhoven**
Open University of the Netherlands, Heerlen, The Netherlands
NWO-I, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

**Sung-Shik Jongmans**[1]
Open University of the Netherlands, Heerlen, The Netherlands
NWO-I, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

**José Proença**
CISTER, ISEP, Polytechnic Institute of Porto, Portugal

## Abstract

Construction and analysis of distributed systems is difficult. Multiparty session types (MPST) constitute a method to make it easier. The idea is to use type checking to statically prove deadlock freedom and protocol compliance of communicating processes. In practice, the premier approach to apply the MPST method in combination with mainstream programming languages has been based on API generation. In this paper (pearl), we revisit and revise this approach.

Regarding our "revisitation", using Scala 3, we present the existing API generation approach, which is based on deterministic finite automata (DFA), in terms of both the existing states-as-classes encoding of DFAs as APIs, and a new states-as-type-parameters encoding; the latter leverages match types in Scala 3. Regarding our "revision", also using Scala 3, we present a new API generation approach that is based on sets of pomsets instead of DFAs; it crucially leverages match types, too. Our fresh perspective allows us to avoid two forms of combinatorial explosion resulting from implementing concurrent subprotocols in the DFA-based approach. We implement our approach in a new API generation tool.

**2012 ACM Subject Classification** Software and its engineering → Software notations and tools

**Keywords and phrases** Concurrency, pomsets (partially ordered multisets), match types, Scala 3

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2022.27

**Supplementary Material** *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.19`

**Funding** *G. Cledou and J. Proença*: European Regional Development Fund (ERDF), Operational Programme for Competitiveness and Internationalisation (COMPETE 2020): POCI-01-0145-FEDER-029946 (DaVinci). *S. Jongmans*: Netherlands Organisation of Scientific Research: 016.Veni.192.103. *J. Proença*: Fundação para a Ciência e a Tecnologia (FCT), within the CISTER Research Unit: UIDP/UIDB/04234/2020. ERDF and FCT, Portugal 2020 Partnership Agreement, Norte Portugal Regional Operational Programme (NORTE 2020): NORTE-01-0145-FEDER-028550 (REASSURE). ECSEL Joint Undertaking (JU): grant agreement No 876852 (VALU3S).

[1] Corresponding author





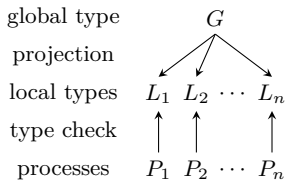



**Figure 1** MPST method.

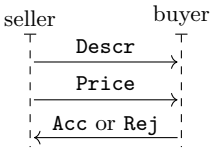



**Figure 2** Seller–buyer protocol (Example 1).

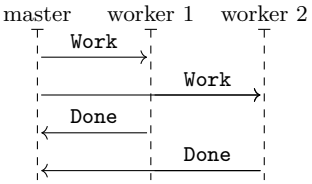



**Figure 3** Master–workers protocol (Example 2).

## 1 Introduction

**Background.** Construction and analysis of distributed systems is difficult. One of the key challenges is to verify absence of communication errors, by proving *deadlock freedom* (i.e., the processes can always terminate or reduce) and *protocol compliance* (i.e., if the processes can terminate or reduce, then the protocol allows it). *Multiparty session types* (MPST) [18, 19] constitute a method to overcome these challenges. The idea is visualised in Figure 1:

1. First, a *protocol* among *roles* $r_1, \ldots, r_n$ is implemented as a *session* of *processes* $P_1, \ldots, P_n$ (concrete), while it is specified as a *global type* $G$ (abstract). The global type models the behaviour of all processes, collectively, from their shared perspective (e.g., "first, a number from Alice to Bob; next, a boolean from Bob to Carol").
2. Next, $G$ is decomposed into local types $L_1, \ldots, L_n$, by *projecting* $G$ onto every role. Every local type models the behaviour of one process, individually, from its own perspective (e.g., for Bob, "first, he receives a number from Alice; next, he sends a boolean to Carol").
3. Last, absence of communication errors is verified, by *type-checking* every process $P_i$ against local type $L_i$. MPST theory guarantees that well-typedness at compile-time (statically) implies deadlock freedom and protocol compliance at execution-time (dynamically).

The following two examples [6, 34] further illustrate global/local types in the MPST method.

▶ **Example 1** (seller–buyer [6]). In the *seller–buyer protocol*, visualised in Figure 2, first, the *seller* (**s**) tells the *buyer* (**b**) the description of an item (`Descr`) and a price (`Price`); next, the buyer tells the seller whether it accepts the offer (`Acc`) or rejects it (`Rej`). The following global type specifies the protocol from the seller's and the buyer's shared perspective:

$$G = \mathbf{s} \twoheadrightarrow \mathbf{b}{:}\mathtt{Descr} \,.\, \mathbf{s} \twoheadrightarrow \mathbf{b}{:}\mathtt{Price} \,.\, \mathbf{s} \twoheadrightarrow \mathbf{b}{:}\{\mathtt{Acc}, \mathtt{Rej}\} \,.\, \mathbf{end}$$

In this notation, $p \twoheadrightarrow q{:}\{t_i.G_i\}_{1 \le i \le n}$ specifies the communication of a value of type $t_i$ from role $p$ to role $q$, followed by $G_i$, for some $1 \le i \le n$. We write $p \twoheadrightarrow q{:}\{t_i\}_{1 \le i \le n}.G$ as a macro for $p \twoheadrightarrow q{:}\{t_i.G\}_{1 \le i \le n}$, and we omit braces when $n = 1$. The following local types (projected from the global type) specify the protocol from the seller's and the buyer's own perspectives:

$$L_{\mathbf{s}} = \mathbf{sb}!\mathtt{Descr} \,.\, \mathbf{sb}!\mathtt{Price} \,.\, \mathbf{sb}?\{\mathtt{Acc}, \mathtt{Rej}\} \,.\, \mathbf{end} \quad L_{\mathbf{b}} = \ldots$$

In this notation, $pq!\{t_i.L_i\}_{1 \le i \le n}$ and $pq?\{t_i.L_i\}_{1 \le i \le n}$ specify the send and receive of a value of type $t_i$ from role $p$ to role $q$, followed by $L_i$, for some $1 \le i \le n$. We write $pq!\{t_i\}_{1 \le i \le n}.L$ and $pq?\{t_i\}_{1 \le i \le n}.L$ as macros for $pq!\{t_i.L\}_{1 \le i \le n}$ and $pq?\{t_i.L\}_{1 \le i \le n}$, and we omit braces when $n = 1$. Henceforth, also, we usually omit "$.\mathbf{end}$". ⌟


**Figure 4** Workflow of API generation (the first three arrows are performed automatically by the tool; the last arrow is performed manually by the programmer).

▶ **Example 2** (master–workers [34]). In the *master–workers protocol*, visualised in Figure 3, first, the *master* (`m`) tells two *workers* ($\mathtt{w}_1, \mathtt{w}_2$) to perform work (`Work`); next, the workers tell the master that they are done (`Done`). The following global/local types specify the protocol:

$$G = \mathtt{m} \to \mathtt{w}_1 : \mathtt{Work} \,.\, \mathtt{m} \to \mathtt{w}_2 : \mathtt{Work} \,.\, \mathtt{w}_1 \to \mathtt{m} : \mathtt{Done} \,.\, \mathtt{w}_2 \to \mathtt{m} : \mathtt{Done}$$

$$L_{\mathtt{m}} = \mathtt{m}\mathtt{w}_1 ! \mathtt{Work} \,.\, \mathtt{m}\mathtt{w}_2 ! \mathtt{Work} \,.\, \mathtt{w}_1\mathtt{m} ? \mathtt{Done} \,.\, \mathtt{w}_2\mathtt{m} ? \mathtt{Done}$$

$$L_{\mathtt{w}_1} = \mathtt{m}\mathtt{w}_1 ? \mathtt{Work} \,.\, \mathtt{w}_1\mathtt{m} ! \mathtt{Done} \qquad L_{\mathtt{w}_2} = \ldots$$

In practice, the premier approach to apply the MPST method in combination with mainstream programming languages has been based on *API generation*. The main ideas, originally conceived by Deniélou, Hu, and Yoshida, are based on the following insights: **(a)** local types can be interpreted "operationally" as *deterministic finite automata* (DFA) [11,12]; **(b)** DFAs can be encoded as *application programming interfaces* (API), such that well-typed usage of the APIs at compile-time implies deadlock freedom and protocol compliance at execution-time (cf. step 3 of the MPST method) [20, 21]. The corresponding workflow is visualised in Figure 4. API generation has been influential: it is used in the majority of tools that support the MPST method, including Scribble [20], its many dialects/extensions [7,25,27,31,33,37,46], νScr [45], and `mpstpp` [23].

**Unsolved: concurrent subprotocols in MPST practice.** The global/local types in Example 1 and Example 2 specify *sequential* protocols: there is only a single static order in which the roles are allowed to communicate. Intuitively, however, imposing such a single static order is needlessly restrictive: in the seller–buyer protocol, there is no apparent reason why the `Descr`-message and the `Price`-message should be sent by the seller in that order, while in the master–workers protocol, there is no apparent reason why the `Done`-messages should be received by the master in "worker-id-order". Thus, the specified protocols in these examples are not just sequential; they are *oversequentialised*.

In general, oversequentialisation in global/local types should be avoided for two reasons:

- *Some ordering decisions can be made only at implementation-time*, based on implementation details that are unknown at specification-time. In Example 1, it may be known only at implementation-time that the seller first computes the contents of the `Price`-message and next of the `Descr`-message. Thus, to maximise throughput, the seller should be able to send these messages in this alternative static order as well (forbidden in Example 1).
- *Some ordering decisions can be made only at execution-time*, based on execution details that are unknown both at specification-time and at implementation-time. In Example 2, it is known only at execution-time in which order the workers send the `Done`-messages (depending on how much actual time performing the work takes). Thus, to improve throughput, the master should be allowed to receive those messages in any dynamic order (forbidden in Example 2), which may be different in different executions.

To allow ordering decisions to be made at implementation-time and/or execution-time, oversequentialisation at specification-time should be avoided. In recognition of this issue, several papers on MPST theory feature a more relaxed version of global/local types in which

1. **State explosion:** Interpretations of local types as DFAs (i.e., second arrow in Figure 4) may suffer from combinatorial explosion: in the presence of parallel composition, DFAs may consist of *an exponential number of states* (e.g., with $n$ workers, the DFA of local type $L_{\mathtt{m}}$ in Example 4 has $2^n + n$ states). As a result, both the time to generate APIs, and the space to store them, are prohibitively long/large for many local types with $\|$.
2. **Branch explosion:** Usages of APIs in processes (i.e., fourth arrow in Figure 4) may suffer from combinatorial explosion, too: in the presence of parallel composition, processes may consist of *an exponential number of branches* to achieve well-typedness. As a result, APIs are prohibitively cumbersome to use for many local types with $\|$.

**Figure 5** Complications of supporting concurrent subprotocols in MPST practice.

*concurrent subprotocols* can be specified (e.g., [6,10,11,23,28]). The idea is to supplement the basic prefix operators $p \twoheadrightarrow q{:}\{t_i.G_i\}_{1\leq i\leq n}$, $pq!\{t_i.L_i\}_{1\leq i\leq n}$, and $pq?\{t_i.L_i\}_{1\leq i\leq n}$ in global/local type calculi with operators for *parallel composition* to express free interleaving (i.e., "fork" subprotocols) and *sequential composition* (i.e., "join" subprotocols).

▶ **Example 3** (seller–buyer, relaxed)**.** The following global/local types specify a relaxed version of the seller–buyer protocol in Example 1:

$$G = (\mathtt{s} \twoheadrightarrow \mathtt{b}{:}\mathtt{Descr} \parallel \mathtt{s} \twoheadrightarrow \mathtt{b}{:}\mathtt{Price}) \cdot \mathtt{s} \twoheadrightarrow \mathtt{b}{:}\{\mathtt{Acc}, \mathtt{Rej}\}$$
$$L_{\mathtt{s}} = (\mathtt{sb}!\mathtt{Descr} \parallel \mathtt{sb}!\mathtt{Price}) \cdot \mathtt{sb}?\{\mathtt{Acc}, \mathtt{Rej}\} \quad L_{\mathtt{b}} = \ldots$$

In this notation, $G_1 \parallel G_2$ (resp. $L_1 \parallel L_2$) specifies the parallel composition of $G_1$ and $G_2$ (resp. $L_1$ and $L_2$) that freely interleaves their communications (resp. sends/receives), while $G_1 \cdot G_2$ (resp. $L_1 \cdot L_2$) specifies the sequential composition of $G_1$ and $G_2$ (resp. $L_1$ and $L_2$). ⌟

▶ **Example 4** (master–workers, relaxed)**.** The following global/local types specify a relaxed version of the master–workers protocol in Example 2:

$$G = \mathtt{m} \twoheadrightarrow \mathtt{w}_1{:}\mathtt{Work} \,.\, \mathtt{m} \twoheadrightarrow \mathtt{w}_2{:}\mathtt{Work} \,.\, (\mathtt{w}_1 \twoheadrightarrow \mathtt{m}{:}\mathtt{Done} \parallel \mathtt{w}_2 \twoheadrightarrow \mathtt{m}{:}\mathtt{Done})$$
$$L_{\mathtt{m}} = \mathtt{mw}_1!\mathtt{Work} \,.\, \mathtt{mw}_2!\mathtt{Work} \,.\, (\mathtt{w}_1\mathtt{m}?\mathtt{Done} \parallel \mathtt{w}_2\mathtt{m}?\mathtt{Done})$$
$$L_{\mathtt{w}_1}, L_{\mathtt{w}_2} = \ldots \text{ (as in Example 2)}$$

We note that the local types of $L_{\mathtt{w}_1}$ and $L_{\mathtt{w}_2}$ are exactly the same as in Example 2. Thus, the relaxation affects only the master. We also note that the protocol can be relaxed even further by allowing the master to send to the workers in any order; we skip it for simplicity of later examples in this paper (in which we revisit the relaxed master–workers protocol). ⌟

However, while the importance of supporting concurrent subprotocols to avoid over-sequentialisation has been duly recognised in MPST theory [6, 10, 11, 23, 28], almost none of the API generation tools offer it in MPST practice [7, 20, 25, 27, 31, 33, 37, 46]. Figure 5 explains two major complications; they pertain to the second arrow in Figure 4 ("interpret as") and the fourth arrow ("use in"). The only API generation tool that features parallel composition does not at all address these complications [23] (i.e., it suffers from both forms of combinatorial explosion in Figure 5). Thus, while concurrent subprotocols are supported in MPST theory, they are effectively unsupported in API-generation-based MPST practice.

We note that concurrent subprotocols are effectively supported by some tools that are based on *runtime verification*. For instance, the tool by Demangeon et al. [9] uses an optimised DFA representation (in which "inner" DFAs for subprotocols can be nested inside states of an "outer" DFA) to compactly represent parallel composition; process behaviour

is dynamically monitored against optimised DFAs. Alternatively, the tool by Hamers and Jongmans [16] computes traces of DFAs on-the-fly by dynamically interpreting global types, guided by process behaviour at execution-time, so the full state space is never computed.

**Contributions.** In this paper (pearl), we present a fresh perspective on API generation: we show how to effectively support concurrent subprotocols for the first time in MPST practice. To achieve this, we leverage two recent advances:

- On the theoretical side, we take advantage of Guanciale–Tuosto's pomset framework [14] to interpret local types as *sets of pomsets* (SOPs) instead of as DFAs. The key benefit of SOPs over DFAs is that parallel composition can be represented in linear time and space. In this way, both complications in Figure 5 can be avoided. Our usage of Guanciale–Tuosto's pomset framework in API generation is novel.
- On the practical side, we take advantage of Scala 3's *match types* (i.e., "lightweight form of dependent typing") [1] to encode SOPs into APIs. The key benefit of match types is that they enable type-level programming; our encoding pivotally relies on these advanced static capabilities (e.g., the encoding cannot be ported to Java). Our usage of Scala 3's match types in API generation is novel. (We note that Scalas et al. also use new features in Scala 3 to support the MPST method [40, 41], but not match types.)

In § 2, we summarise a version MPST theory that includes parallel composition and sequential composition. In § 3, we ***revisit*** API generation by presenting the existing DFA-based version in Scala 3. Besides the existing "states-as-classes" encoding of DFAs, we also present a new "states-as-type-parameters" encoding that uses match types. In § 4, we ***revise*** API generation to avoid the complications in Figure 5 by presenting a new SOP-based version that also uses match types. In § 5, we give a brief overview of our tool.

## 2 MPST Theory in a Nutshell

In this section, we summarise a *minimal, loop-free core* of Deniélou–Yoshida's version of MPST theory, which includes parallel composition and sequential composition [10]. That is, given the aim of this paper, we omit orthogonal and/or more advanced features from this section (e.g., dynamic channel creation, dynamic process creation, delegation). Regarding "loop-free", we note that many practically relevant protocols do not require loops (e.g., the auction protocol in [10], the ATM protocol in [14], and the OAuth protocol in [22]).

**Global types.** Let $\mathbb{R} = \{\texttt{alice}, \texttt{bob}, \texttt{carol}, \ldots, \texttt{s}, \texttt{c}, \texttt{m}, \texttt{w}, \ldots\}$ denote the set of all *roles*, ranged over by $p, q, r$. Let $\mathbb{T} = \{\texttt{Unit}, \texttt{Bool}, \texttt{Nat}, \ldots, \texttt{Descr}, \texttt{Price}, \texttt{Acc}, \texttt{Rej}, \ldots\}$ denote the set of all *data types*, ranged over by $t$. Let $\mathbb{G}$ denote the set of all *global types*, ranged over by $G$:

$$G ::= \textbf{end} \mid p \twoheadrightarrow q : \{t_i.G_i\}_{1 \leq i \leq n} \mid G_1 \parallel G_2 \mid G_1 \cdot G_2$$

Informally, these forms of global types have the following meaning:

- Global type **end** specifies the **empty protocol**.
- Global type $p \twoheadrightarrow q : \{t_i.G_i\}_{1 \leq i \leq n}$ specifies the **asynchronous communication** of a value of type $t_i$ through the buffered channel from role $p$ to role $q$ (unbounded), followed by $G_i$, for some $1 \leq i \leq n$. As additional well-formedness requirements, we stipulate: **(1)** $p \neq q$ (i.e., no self-communication); **(2)** $t_i \neq t_j$, for every $1 \leq i < j \leq n$ (i.e., deterministic continuations). Singleton types (e.g., $\texttt{Acc}, \texttt{Rej}$) can serve as *labels* to communicate choices.

$$\mathbf{end} \upharpoonright r = \mathbf{end}$$

$$p \twoheadrightarrow q{:}\{t_i.G_i\}_{1\le i\le n} \upharpoonright r = \begin{cases} pq!\{t_i.(G_i \upharpoonright r)\}_{1\le i\le n} & \text{if } p = r \neq q \\ pq?\{t_i.(G_i \upharpoonright r)\}_{1\le i\le n} & \text{if } p \neq r = q \\ G_1 \upharpoonright r & \text{if } p \neq r \neq q \text{ and } G_1 \upharpoonright r = \cdots = G_n \upharpoonright r \end{cases}$$

$$(G_1 \oplus G_2) \upharpoonright r = (G_1 \upharpoonright r) \oplus (G_2 \upharpoonright r)$$

**Figure 6** Projection of global types.

- Global type $G_1 \parallel G_2$ specifies the **parallel composition** of $G_1$ and $G_2$ that freely interleaves their communications. As an additional well-formedness requirement [10], we stipulate $\mathsf{comm}(G_1) \cap \mathsf{comm}(G_2) = \emptyset$ (i.e., distinct communications in distinct subprotocols), where $\mathsf{comm} : \mathbb{G} \to 2^{\mathbb{R}\times\mathbb{R}\times\mathbb{T}}$ is a function that maps every global type to the communications that occur in it, represented as triples of the form $(p, q, t)$.
- Global type $G_1 \cdot G_2$ specifies the **sequential composition** of $G_1$ and $G_2$.

**Local types and projection.** Let $\mathbb{L}$ denote the set of all *local types*, ranged over by $L$:

$$L ::= \mathbf{end} \mid \underbrace{pq!\{t_i.L_i\}_{1\le i\le n}}_{\text{send}} \mid \underbrace{pq?\{t_i.L_i\}_{1\le i\le n}}_{\text{receive}} \mid L_1 \parallel L_2 \mid L_1 \cdot L_2$$

These forms of local types have a similar meaning as the corresponding forms of global types. Henceforth, let $\dagger \in \{!, ?\}$ and $\oplus \in \{\parallel, \cdot\}$.

Let $G \upharpoonright r$ denote the *projection* of $G$ onto $r$. Formally, $\upharpoonright$ is the smallest partial function induced by the equations in Figure 6. The projections of **end**, $G_1 \parallel G_2$, and $G_1 \cdot G_2$ are easy. The projection of $p \twoheadrightarrow q{:}\{t_i.G_i\}_{1\le i\le n}$ onto $r$ depends on the contribution of $r$ to the communication: if $r$ is sender (resp. receiver), then the projection specifies a send (resp. receive); if $r$ does not contribute to the communication, and if $r$ has a unique continuation, then the projection is that continuation. The latter means that $r$ is insensitive to which type was communicated (which, as a non-contributor to the communication, $r$ does not know). We note that projection is partial: if the projection of a global type onto one of its roles is undefined, then the global type is unsupported. We also note that, for simplicity and because it does not affect this paper, we use the "plain merge" instead of the "full merge" [39].

**Processes and typing rules.** Let $\mathbb{V}$ denote the set of all *values*, ranged over by $v$. Let $\mathbb{X}$ denote the set of all *variables*, ranged over by $x$. Let $\mathbb{E}$ denote the set of all *expressions*, ranged over by $e$. Let $\mathbb{P}$ denote the set of all *processes*, ranged over by $P$:

$$P ::= \mathbf{0} \mid pq!e.P \mid pq?\{x_i{:}t_i.P_i\}_{1\le i\le n} \mid P_1 \parallel P_2 \mid P_1 \cdot P_2$$

Informally, these forms of processes have the following meaning:

- Process $\mathbf{0}$ implements the **empty role**.
- Process $pq!e.P$ implements the **asynchronous send** of the value of expression $e$ through the buffered channel from role $p$ to role $q$, followed by $P$. Asynchronous sends can be combined with conditional choices to implement *internal choices* by a process. For instance, the following process implements the second part of the buyer in Example 1:

  **if** `goodOffer(descr,price)` (**bs**`!Acc().`$\mathbf{0}$) (**bs**`!Rej().`$\mathbf{0}$)

$$\frac{\Gamma \vdash e : t_k \quad \Gamma \vdash P : L_k}{\Gamma \vdash pq!e.P : pq!\{t_i.L_i\}_{1\leq i\leq n}}\ [\textsc{Send}] \qquad \frac{\Gamma, x_i : t_i \vdash P_i : L_i \text{ for every } 1 \leq i \leq n}{\Gamma \vdash pq?\{x_i{:}t_i.P_i\}_{1\leq i\leq n} : pq?\{t_i.L_i\}_{1\leq i\leq n}}\ [\textsc{Recv}]$$

$$\frac{}{\Gamma \vdash \mathbf{0} : \mathbf{end}}\ [\textsc{Empty}] \qquad \frac{\Gamma \vdash P_1 : L_1 \quad \Gamma \vdash P_2 : L_2}{\Gamma \vdash P_1 \parallel P_2 : L_1 \parallel L_2}\ [\textsc{Par}] \qquad \frac{\Gamma \vdash P_1 : L_1 \quad \Gamma \vdash P_2 : L_2}{\Gamma \vdash P_1 \cdot P_2 : L_1 \cdot L_2}\ [\textsc{Seq}]$$

**Figure 7** Well-typedness of processes.

- Process $pq?\{x_i{:}t_i.P_i\}_{1\leq i\leq n}$ implements the **asynchronous receive** of a value of type $t_i$ into variable $x_i$ through the buffered channel from role $p$ to role $q$, followed by $P_i$ (i.e., type switch on the received value), for some $1 \leq i \leq n$. Asynchronous receives can be used to implement *external choices* by the environment of a process. For instance, the following process implements the second part of the seller in Example 1: $\texttt{bs}?\{\texttt{x:Acc}.\mathbf{0}, \texttt{x:Rej}.\mathbf{0}\}$. Thus, through an internal choice and a reciprocal external choice, the sender can "select" a value of a particular type to control whereto the receiver "branches off".

- Process $P_1 \parallel P_2$ implements the parallel composition of $P_1$ and $P_2$. We note that $P_1 \parallel P_2$ is intended to implement *one* role (i.e., there is no communication between $P_1$ and $P_2$); the only purpose of parallel composition is to allow the sends and receives of $P_1$ and $P_2$ to be ordered dynamically at execution-time.

- Process $P_1 \cdot P_2$ implements the sequential composition of $P_1$ and $P_2$.

Let $\Gamma \vdash e : t$ denote *well-typedness* of expression $e$ by data type $t$ in environment $\Gamma$. Let $\Gamma \vdash P : L$ denote *well-typedness* of process $P$ by local type $L$ in environment $\Gamma$. Formally, $\vdash$ is the smallest relation induced by the rules in Figure 7. Rule [Empty] states that the empty role is well-typed by the empty protocol. Rule [Send] states that a send is well-typed by $pq!\{t_i.L_i\}_{1\leq i\leq n}$ if, for some $1 \leq k \leq n$, the value to send is well-typed by $t_k$ and the continuation is well-typed by $L_k$. Dually, rule [Recv] states that a receive is well-typed by $pq?\{t_i.L_i\}_{1\leq i\leq n}$ if, for every $1 \leq i \leq n$, the continuation is well-typed by $L_i$ under the additional assumption that the received value is well-typed by $t_i$. Thus, a well-typed process needs to be able to consume all specified inputs (i.e., *input-enabledness*), but produce only one specified output. Rules [Par] and [Seq] state that a parallel and sequential composition are well-typed if their operands are.

▶ **Theorem 5** (Deniélou–Yoshida [10]). *If $G$ is a well-formed global type in which roles $r_1, \ldots, r_n$ occur, and if $\vdash P_i : (G \upharpoonright r_i)$ for every $1 \leq i \leq n$, then the session of $P_1, \ldots, P_n$ is deadlock-free and protocol-compliant with respect to $G$.*

## 3 DFA-based API Generation

In this section, we ***revisit*** API generation by presenting the existing DFA-based version in Scala 3, using concepts and notation of the previous section. First, we show how local types can be interpreted operationally as DFAs (§ 3.1). Next, we show how DFAs can be encoded as APIs using the existing *"states-as-classes" encoding* (§ 3.2). Last, we also show how DFAs can be encoded as APIs using a new *"states-as-type-parameters" encoding* (§ 3.3). The value of this second encoding is twofold: it yields APIs with a smaller memory footprint, and it gently introduces match types to set the stage for the next section.

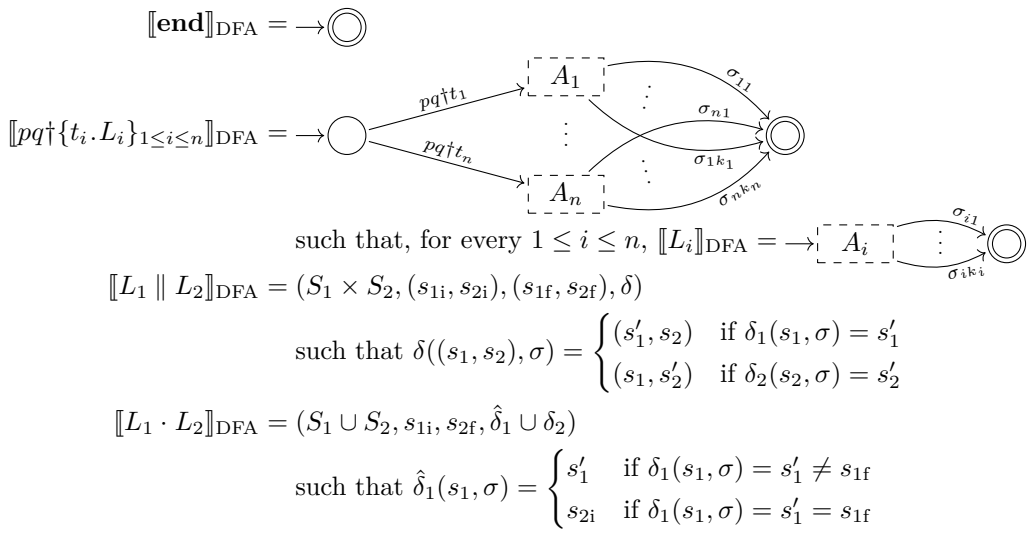



**Figure 8** Interpretation of local types as DFAs.

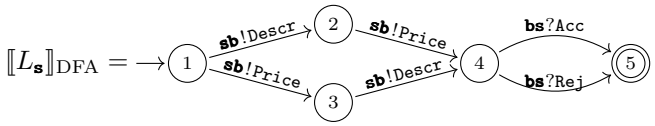
## 3.1 From Local Types to DFAs

The first interpretation of local types as DFAs was discovered by Deniélou and Yoshida [11]. The key insight is that a local type for a role essentially defines a regular language, each of whose words represents an admissible execution of the role's implementation.

**DFAs, formally.** Let $\mathbf{\Sigma} = \{pq!t \mid p \neq q\} \cup \{pq?t \mid p \neq q\}$ denote the set of all *type-level actions* ("the alphabet"), ranged over by $\sigma$. Let $\mathbb{A}$ denote the set of all *deterministic finite automata* (DFA) over $\mathbf{\Sigma}$, ranged over by $A$. Formally, a DFA is a tuple $(S, s_{\mathrm{i}}, s_{\mathrm{f}}, \delta)$, where $S$ denotes a set of *states*, $s_{\mathrm{i}}, s_{\mathrm{f}} \in S$ denote the *initial state* and the *final state*, and $\delta : S \times \mathbf{\Sigma} \rightharpoonup S$ denotes a *transition function*.

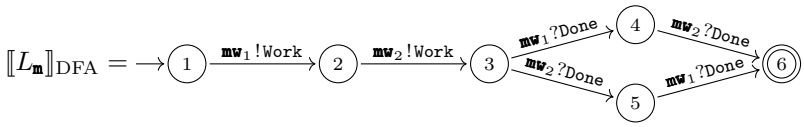
**Interpretation.** Let $[\![L]\!]_{\mathrm{DFA}}$ denote the *interpretation* of local type $L$ as a DFA. Formally, $[\![\text{-}]\!]_{\mathrm{DFA}} : \mathbb{L} \to \mathbb{A}$ is the smallest function induced by the equations in Figure 8. The interpretation of **end** is the DFA that accepts the empty language. The interpretation of $pq\dagger\{t_i.L_i\}_{1\le i\le n}$ is the DFA that accepts the language of words that begin with $pq\dagger t_i$ and continue with a word accepted by the interpretation of $L_i$; the visualisation is intended to convey that the final states of the interpretations of $L_1, \ldots, L_n$ are "superimposed" to form a single new final state. The interpretations of $L_1 \parallel L_2$ and $L_1 \cdot L_2$ are the DFAs that accept the shuffle and the concatenation of the languages accepted by the interpretations of $L_1$ and $L_2$. We note that the transition function of $[\![L_1 \parallel L_2]\!]_{\mathrm{DFA}}$ is well-defined due to the well-formedness requirement (§ 2) that the sets of sends and receives that occur in $L_1$ and $L_2$ are disjoint (i.e., $\delta_1(s_1, \sigma) = s_1'$ and $\delta_2(s_2, \sigma) = s_2'$ cannot both be true).

▶ **Example 6.** The following DFA is the interpretation of $L_{\mathbf{s}}$ in Example 3:




⌟

▶ **Example 7.** The following DFA is the interpretation of $L_{\mathtt{m}}$ in Example 4:




## 3.2 From DFAs to APIs – Using Classes

The first encoding of DFAs as APIs was developed by Hu and Yoshida [20]. The key insight is that an input-enabled process $P$ is well-typed by local type $L$ (§ 2) if, and only if, every possible sequence of sends and receives by $P$ forms a word accepted by $[\![L]\!]_{\mathrm{DFA}}$. The "trick", then, is to structure the API in such a way that when the compiler successfully type-checks the API's usage, it has effectively performed an *accepting run* of $[\![L]\!]_{\mathrm{DFA}}$ for every possible sequence of sends and receives by $P$. In the rest of this subsection, we show how to achieve this in Scala 3 using the existing states-as-classes encoding of DFAs as APIs; experts may skip this subsection, or quickly browse through it only to familiarise themselves with notation.

Suppose that $(S, s_{\mathrm{i}}, s_{\mathrm{f}}, \delta)$ is the interpretation of the local type for role $r$:

- Every state $s \in S$ is encoded as class $\langle r \rangle \mathtt{\$} \langle s \rangle$ in the API, where $\langle r \rangle$ and $\langle s \rangle$ are identifiers for $r$ and $s$ (and `$` is a meaningless separator).
- Every transition $\delta(s, \sigma)$ is encoded as a method of class $\langle r \rangle \mathtt{\$} \langle s \rangle$ to perform action $\sigma$ and provide an instance of class $\langle r \rangle \mathtt{\$} \langle \delta(s, \sigma) \rangle$, as detailed shortly.

To use the API, the idea is to define a function `f` that consumes an "initial state object" `s` of type $\langle r \rangle \mathtt{\$} \langle s_{\mathrm{i}} \rangle$ as input and produces a "final state object" of type $\langle r \rangle \mathtt{\$} \langle s_{\mathrm{f}} \rangle$ as output. Inside of `f`, initially, the only protocol-related actions that can be performed, are those for which `s` has a method. When such a method is called on `s`, an action is performed and a fresh "successor state object" `sNext` is provided. Subsequently, the only protocol-related actions that can be performed, are those for which `sNext` has a method. When such a method is called on `sNext`, another action is performed, and another fresh "successor successor state object" `sNextNext` is provided. This goes on until the final state object is provided.[2] To ensure that every state object is used at most once (see also footnote 2), every state class extends the following trait:

```
trait UseOnce:
  var used = false
  def use = if used then throw new Exception() else used = true
```

When a method is called on a state object `s`, inside of it, method `use` is first called to ensure that `s` has not been used before. Otherwise, an exception is thrown. This technique was first used by Tov and Pucella [42] and has since been adopted in tools for both binary sessions (e.g., [35, 38]) and multiparty sessions (e.g., [7, 20, 23, 31, 33, 37, 45]). We note that `used` in `UseOnce` should actually be declared `private` to shield it from external modification; we omit such access modifiers in our listings for simplicity.

[2] When a method is called on an instance of class $\langle r \rangle \mathtt{\$} \langle s \rangle$, *but the method does not exist*, it means that: **(1)** state $s$ in the DFA does not have a corresponding transition; **(2)** hence, the local type for $r$ does not specify the corresponding action; **(3)** hence, the action is not allowed in the protocol. The compiler statically detects this while type-checking and reports an error. As successor state objects become available only after predecessor state objects are used, *and assuming that every state object is used exactly once*, well-typed usage of the API implies protocol compliance. Moreover, as a final state object must have been provided upon termination, *and assuming that there are no other sources of non-terminating or exceptional behaviour*, well-typed usage of the API also implies deadlock freedom. We note that these two additional assumptions cannot be statically enforced using Scala 3's type system (just as with many existing tools [7, 20, 23, 31, 33, 37, 45]): checking the first assumption requires a form of substructural typing, while checking the second assumption is generally undecidable. However, the first assumption can be dynamically monitored using lightweight checks at execution-time.

Next, we explain in more detail how states and transitions can be encoded as classes and methods. As usual in DFA-based API generation, we require that every state in the DFA-interpretation of a local type is either an *output state* with only send transitions, or an *input state* with only receive transitions. In the absence of parallel composition, this requirement is satisfied by construction (§ 3.2), but in the presence of parallel composition, it needs to be checked separately. The classes for output states and input states look as follows:

- Suppose that $(S, s_\mathrm{i}, s_\mathrm{f}, \delta)$ is the interpretation of the local type for role $p$. Every output state $s \in S$, with transitions $\delta(s, pq_1!t_1), \ldots, \delta(s, pq_n!t_n)$, is encoded as follows:

```
class ⟨p⟩$⟨s⟩(net: Network) extends UseOnce: // output state
  def send(q: ⟨q1⟩, e: ⟨t1⟩): ⟨p⟩$⟨δ(s, pq1!t1)⟩ = { use; ... /* real send */ }
  ...
  def send(q: ⟨qn⟩, e: ⟨tn⟩): ⟨p⟩$⟨δ(s, pqn!tn)⟩ = { use; ... /* real send */ }
```

  Parameter `net` of class $\langle p \rangle\$\langle s \rangle$ encapsulates the underlying communication infrastructure (e.g., shared-memory channels); it is used inside of every `send` method to perform the "real send". Parameter `q` of every `send` method is the identifier of the receiver, parameter `e` is the value to send, and the return value is a fresh successor state object. Roughly, these methods in Scala 3 are typed versions of $pq!e.P$ in the process calculus (§ 2).

- Suppose that $(S, s_\mathrm{i}, s_\mathrm{f}, \delta)$ is the interpretation of the local type for role $q$. Every input state $s \in S$, with transitions $\delta(s, p_1q?t_1), \ldots, \delta(s, p_nq?t_n)$, is encoded as follows:

```
class ⟨q⟩$⟨s⟩(net: Network) extends UseOnce: // input state
  def recv(f1: (⟨p1⟩, ⟨t1⟩, ⟨q⟩$⟨δ(s, p1q?t1)⟩)) => ⟨q⟩$⟨sf⟩,
           ...,
           fn: (⟨pn⟩, ⟨tn⟩, ⟨q⟩$⟨δ(s, pnq?tn)⟩)) => ⟨q⟩$⟨sf⟩) = { use; ... /* real recv */ }
```

  Parameter `f`$i$ of method `recv` is the $i$-th *continuation*; it is called with the identifier of the sender, the value to receive, and a fresh successor state object after the "real receive". Roughly, this method in Scala 3 is $pq?\{x_i{:}t_i.P_i\}_{1 \leq i \leq n}$ in the process calculus (§ 2).

▶ **Example 8.** The following API is the states-as-classes encoding of $[\![L_\mathbf{s}]\!]_\mathrm{DFA}$ in Example 6:

```
class S$1(net: Network) extends UseOnce:
  def send(q: B, e: Descr): S$2 = ...
  def send(q: B, e: Price): S$3 = ...
class S$2(net: Network) extends UseOnce:
  def send(q: B, e: Price): S$4 = ...
class S$3(net: Network) extends UseOnce:
  def send(q: B, e: Descr): S$4 = ...
```

```
class S$4(net: Network) extends UseOnce:
  def recv(
    f1: (B, Acc, S$5) => S$5,
    f2: (B, Rej, S$5) => S$5): S$5 = ...
class S$5(net: Network) extends UseOnce

type S$Initial = S$1
type S$Final   = S$5
```

The following well-typed function implements the seller:

```
def seller(s: S$Initial): S$Final = s
  .send(B, new Descr).send(B, new Price).recv(
    (q: B, x: Acc, s) => { println("offer accepted"); s },
    (q: B, x: Rej, s) => { println("offer rejected"); s }
```

We note that the two sends in the implementation of the seller can be swapped (i.e., first the price, second the description): the resulting code would still be well-typed, indicating to the programmer that the protocol is not violated. We also note that the type of parameter `s` in the continuations on the last two lines is inferred by the compiler. This demonstrates that the programmer does not need to know how states are represented. The types of parameters `q` and `x` can be inferred as well, so the annotations are redundant; we added them here for clarity of presentation (but will omit them from now on). For details, see: `https://scastie.scala-lang.org/779xP1Z8QwC1DLKDFYAsJg`. ⌟

▶ **Example 9.** The following API is the states-as-classes encoding of $[\![L_{\texttt{m}}]\!]_{\text{DFA}}$ in Example 7:

```
class M$1(net: Network) extends UseOnce:
  def send(q: W1, e: Work): M$2 = ...
class M$2(net: Network) extends UseOnce:
  def send(q: W2, e: Work): M$3 = ...
class M$3(net: Network) extends UseOnce:
  def recv(
    f1: (W1, Done, M$4) => M$6,
    f2: (W2, Done, M$5) => M$6): M$6 = ...
```

```
class M$4(net: Network) extends UseOnce:
  def recv(f2: ... => M$6): M$6 = ...
class M$5(net: Network) extends UseOnce:
  def recv(f1: ... => M$6): M$6 = ...
class M$6(net: Network) extends UseOnce

type M$Initial = M$1
type M$Final = M$6
```

The following well-typed function implements the master:

```
def master(s: M$Initial): M$Final = s
  .send(W1, new Work).send(W2, new Work).recv(
    (_, _, s) => s.recv((_, _, s) => { println("first #1, second #2"); s }),
    (_, _, s) => s.recv((_, _, s) => { println("first #2, second #1"); s }))
```

We note that the two sends in the implementation of the master cannot be swapped (i.e., first to worker 2, second to worker 1): the resulting code would not be well-typed, indicating to the programmer that the protocol is violated. We also note that we omitted all type annotations for parameters in continuations; they can be inferred by the compiler. For details, see: `https://scastie.scala-lang.org/Lg2ZNlw8T7eTfcef7k3yIw`. ⌟

## 3.3 From DFAs to APIs – Using Type Parameters

In the previous subsection, we showed how the existing *states-as-classes* encoding of DFAs can be used to "trick" the compiler into performing type-level accepting runs. Attractively, states-as-classes requires only basic features of the underlying type system; as a result, it can be applied in combination with a wide range of programming languages (e.g., F# [33], F⋆ [46], Go [7], Java [20], OCaml [45], PureScript [25], Rust [27], Scala [37], TypeScript [31]). In this subsection, we present a new *states-as-type-parameters* encoding of DFAs that leverages an advanced feature of Scala 3's type system: *match types* [1]. While the primary aim of this subsection is to gently explain match types and set the stage for §4, states-as-type-parameters has a technical advantage as well: it is more space-efficient (i.e., states-as-classes requires all specific state classes to be loaded in memory, which can be many, while states-as-type-parameters requires only one generic state class to be loaded.)

**Match types.** As a brief digression, consider the following example to introduce match types: suppose that we need to write a function that converts `Int`s to `Boolean`s and vice versa. Naively, we could write the signature of this function as follows:

```
type IntOrBoolean = Int | Boolean // type alias for a union type
def convert(x: IntOrBoolean): IntOrBoolean = x match
  case i: Int     => i == 1
  case b: Boolean => if b then 1 else 0
```

The trouble with this first attempt is that the return type, `IntOrBoolean`, is insufficiently precise. For instance, the compiler fails to prove that expression `convert(5) && false` is safe, as it cannot infer that `convert(5)` is of type `Boolean`. Essentially, what the compiler is missing, is a relation between the actual type of `x` (e.g., `Int`) and the return type (e.g., `Boolean`). Match types allow us to define such relations.

1. First, we redefine the signature of `convert` as follows:

   ```
   def convert[T <: IntOrBoolean](x: T): Convert[T] = ... // same as before
   ```

   That is, we introduce a type parameter `T`, which must be a subtype of `IntOrBoolean` (generally, `A` and `B` are subtypes of `A|B`), and we declare `x` to be of type `T`. Furthermore, we declare the return value of the function to be of match type `Convert[T]`. The idea is to define `Convert[T]` in such a way that the intended relation between the actual type of `x` and the return type can be inferred.

2. Next, we define `Convert[T]` as follows:

```
type Convert[T] = T match
  case Int     => Boolean
  case Boolean => Int
```

The compiler *reduces* every occurrence of `Convert[T]` to `Int` or `Boolean`, depending on the instantiation of `T` (e.g., `Convert[Int]` reduces to `Boolean`).

3. Last, for instance, the compiler succeeds/fails to type-check the following expressions:

```
convert(5) + 6          // fail
convert(5) && false     // succeed
```

```
convert(true) + 6       // succeed
convert(true) && false  // fail
```

Thus, match types constitute a "lightweight form of dependent typing" to perform "type-level programming" [1]. In the rest of this subsection, we show how to leverage them in the states-as-type-parameters encoding of DFAs as APIs.

**Encoding of DFAs as APIs.** The idea behind the states-as-type-parameters encoding is to generate, for every role $r$, a generic state class $\langle r \rangle$`$State`; it has one type parameter, called `N`. Every instantiation of `N` with a *numeric literal type* (e.g., in Scala, symbol "`5`" denotes both value 5 and a type with value 5 as its only inhabitant) specialises the generic state class into a specific one. For instance, $\langle r \rangle$`$State[1]`, $\langle r \rangle$`$State[2]`, and $\langle r \rangle$`$State[3]` represent three different states, identified by types `1`, `2`, and `3`. Thus, $\langle r \rangle$`$State` has the following header:

```
class ⟨r⟩$State[N](n: N, net: Network) extends UseOnce:
```

This class has two methods: `send` and `recv`. At execution-time, when `send` or `recv` is called, an action is performed and a fresh successor state object is returned. At compile-time, to check that this method call is actually allowed in the current state, the compiler tries to reduce a match type: if it succeeds, the call is allowed; if it fails, it is not.

Regarding `send`, the idea is to use a match type $\langle r \rangle$`$SendReturn` for the *return value*; it has three type parameters, called `N`, `Q`, and `E`, which identify the current state $s$, the receiver $q$, and the type $t$ of the value to send. If the DFA has a send transition $\delta(s, rq!t) = s'$, then the compiler succeeds to reduce $\langle r \rangle$`$SendReturn[`$\langle s \rangle$`,`$\langle q \rangle$`,`$\langle t \rangle$`]` to $\langle r \rangle$`$State[`$\langle s' \rangle$`]`; this is the type of the fresh successor state object after sending. In contrast, if the DFA does not have such a transition, then the compiler fails to reduce and yields an error. Thus, `send` looks as follows:

```
  def send[Q, E](q: Q, e: E): ⟨r⟩$SendReturn[N, Q, E] = { use; ... }
```

Regarding `recv`, the idea is to use a match type $\langle r \rangle$`$RecvArgument` for the *argument values*; it has one type parameter, called `N`, which identifies the current state $s$. If the DFA has receive transitions $\delta(s, p_1 r?t_1) = s'_1, \ldots, \delta(s, p_n r?t_n) = s'_n$, then the compiler succeeds to reduce $\langle r \rangle$`$RecvArgument[`$\langle s \rangle$`]` to a tuple of function types each of which is of the form `(`$\langle p_i \rangle$`,`$\langle t_i \rangle$`,`$\langle r \rangle$`$State[`$\langle s'_i \rangle$`]) => `$\langle r \rangle$`$Final`; these are the types of the continuations after receiving. In contrast, if the DFA does not have such transitions, then the compiler fails to reduce and yields an error. Thus, `recv` looks as follows:

```
  def recv(fff: ⟨r⟩$RecvArgument[N]): ⟨r⟩$Final = { use; ... }
```

The following examples demonstrate these match types.

▶ **Example 10.** The following API (excerpt) is the states-as-type-parameters encoding of $[\![L_{\mathtt{s}}]\!]_{\mathrm{DFA}}$ in Example 6 (cf. the states-as-classes encoding in Example 8):

```
type S$SendReturn[N, Q, E] =
  (N, Q, E) match
    case (1, B, Descr) => S$State[2]
    case (1, B, Price) => S$State[3]
    case (2, B, Price) => S$State[4]
    case (3, B, Descr) => S$State[4]
```

```
type S$RecvArgument[N] = N match
  case 4 => (
    (B, Acc, S$State[5]) => S$State[5],
    (B, Rej, S$State[5]) => S$State[5])
```

Every case in match type `S$SendReturn` encodes a send transition out of states 1–3 in the DFA in Example 6. Similarly, the single case in match type `S$RecvArgument` encodes the set of receive transitions out of state 4. We note that exactly the same function that implements the seller in Example 8 is also well-typed using the API in this example. For details, see: `https://scastie.scala-lang.org/YB9G1KuVQxGkbwgqm7LKLw`. ⌟

▶ **Example 11.** The following API (excerpt) is the states-as-type-parameters encoding of $[\![L_{\mathtt{m}}]\!]_{\mathrm{DFA}}$ in Example 7 (cf. the states-as-classes encoding in Example 9):

```
type M$SendReturn[N, Q, E] = (N, Q, E) match
  case (1, W1, Work) => M$State[2]
  case (2, W2, Work) => M$State[3]

type M$RecvArgument[N] = N match
  case 3 => ((W1, Done, M$State[4]) => M$State[6],
             (W2, Done, M$State[5]) => M$State[6])
  case 4 => ((W2, Done, M$State[6]) => M$State[6])
  case 5 => ((W1, Done, M$State[6]) => M$State[6])
```

Every case in match type `M$SendReturn` encodes a send transition out of states 1–2 in the DFA for the master in Example 7. Similarly, every case case in match type `M$RecvArgument` encodes a set of receive transitions out of states 3–5. We note that exactly the same functions that implement the master and worker 1 in Example 9 are also well-typed using the API in this example. For details, see: `https://scastie.scala-lang.org/HHy4TUYLREeQYc8yW6UwHw`. ⌟

We note that states-as-type-parameters fully supports recursive protocols, and it never gives rise to non-terminating compile-time reductions of match types: every reduction only checks if a call to `send` or `recv` on a state object is allowed by considering its finitely many outgoing transitions; possibly infinitely long paths through the DFA are not considered.

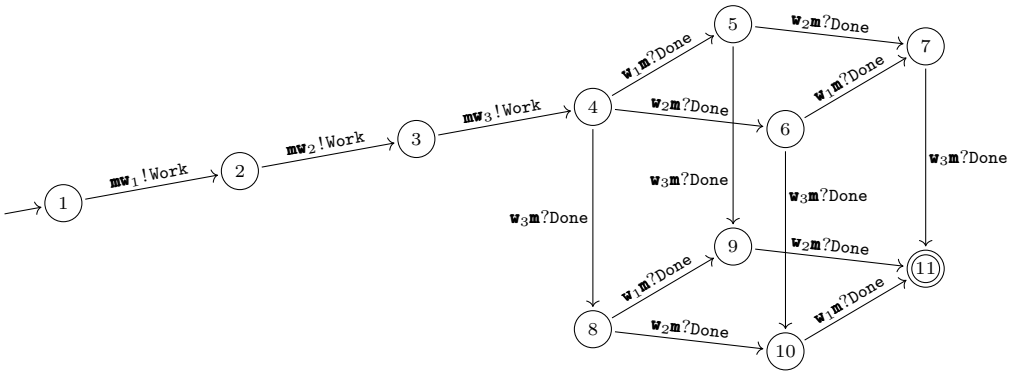
## 4 SOP-based API Generation

To more clearly demonstrate the complications of supporting concurrent subprotocols with DFA-based API generation (Figure 5), we consider another example.

▶ **Example 12.** The following local type specifies the master in the relaxed master–workers protocol (Example 4), but with three workers instead of two:

$$L_{\mathtt{m}} = \mathtt{mw}_1\,!\mathtt{Work} \,.\, \mathtt{mw}_2\,!\mathtt{Work} \,.\, \mathtt{mw}_3\,!\mathtt{Work} \,.\, (\mathtt{w}_1\mathtt{m}\,?\mathtt{Done} \parallel \mathtt{w}_2\mathtt{m}\,?\mathtt{Done} \parallel \mathtt{w}_3\mathtt{m}\,?\mathtt{Done})$$

The following DFA is the interpretation of the local type:




**This DFA demonstrates complication 1 in Figure 5 (i.e., state explosion):** it has $O(2^n)$ states, where $n = 3$ is the number of unordered receives.

The following well-typed function implements the master, using an API that encodes the DFA (whether states-as-classes or states-as-type-parameters is used, is irrelevant here):

```
def master(s: M$Initial): M$Final = s
  .send(W1, new Work).send(W2, new Work).send(W3, new Work).recv(
    (_, _, s) => s.recv(
        (_, _, s) => s.recv(
          (_, _, s) => { println("#1, #2, #3"); s }),
        (_, _, s) => s.recv(
          (_, _, s) => { println("#1, #3, #2"); s })),
    (_, _, s) => s.recv(
        (_, _, s) => s.recv(
          (_, _, s) => { println("#2, #1, #3"); s }),
        (_, _, s) => s.recv(
          (_, _, s) => { println("#2, #3, #1"); s })),
    (_, _, s) => s.recv(
        (_, _, s) => s.recv(
          (_, _, s) => { println("#3, #1, #2"); s }),
        (_, _, s) => s.recv(
          (_, _, s) => { println("#3, #2, #1"); s })))
```

**This implementation demonstrates complication 2 in Figure 5 (i.e., branch explosion):** it has $O(n!)$ branches (each of which implements a distinct order in which the receives might dynamically take place), where $n = 3$ is the number of unordered receives. ⌟

In this section, we ***revise*** API generation to avoid the complications in Figure 5 by presenting a new version based on *sets of pomsets* (SOP). First, we show how local types can be interpreted operationally as SOPs (§ 4.1). Next, we show how SOPs can be encoded as APIs using match types (§ 4.2).

## 4.1 From Local Types to SOPs

Our interpretation of local types as SOPs is based on the recent pomset framework by Guanciale and Tuosto [14]. The key insight is that every subset of *words that differ only in the order of concurrent actions* (in the regular language defined by a local type) can be represented as a pomset in exponentially less time and space.

**Pomsets, formally (structure).** The formalisation of *pomsets* ("partially ordered multisets") is relatively complicated; we first explain the intuition. Recall that a multiset is a set in which every element can have multiple "instances" (e.g., $\{a, a, b\}$ has two elements but three instances). A pomset is just a multiset endowed with a partial order $\prec$ on the instances (e.g., $a \prec b \prec a$, where the left $a$ and the right $a$ are different instances). Following Pratt and Gischer [13, 36], the idea is to formalise pomsets using labelled digraphs: vertices represent instances, arcs represent the ordering, and vertex labels represent elements (e.g.,

$$\frac{v' \not\prec v \text{ for every } v'}{(V, \prec, \lambda) \xrightarrow{\lambda(v)} (V \backslash \{v\}\,,\ \prec \backslash (\{v\} \times V)\,,\ \lambda \backslash (v \mapsto \lambda(v)))}\ [\textsc{Lpo}] \qquad \frac{X \xrightarrow{\sigma} X'}{[X] \xrightarrow{\sigma} [X']}\ [\textsc{Pom}]$$

$$\frac{\mathcal{X}_i \xrightarrow{\sigma} \mathcal{X}_i' \text{ and } 1 \leq i \leq n}{\{\mathcal{X}_1, \ldots, \mathcal{X}_n\} \xrightarrow{\sigma} \{\mathcal{X}_i'\}}\ [\textsc{SopA}] \qquad \text{or} \qquad \frac{\begin{array}{c}\mathcal{X}_i \xrightarrow{\sigma} \mathcal{X}_i' \text{ for every } 1 \leq i \leq k \\ \mathcal{X}_i \not\xrightarrow{\sigma} \text{ for every } k+1 \leq i \leq n\end{array}}{\{\mathcal{X}_1, \ldots, \mathcal{X}_n\} \xrightarrow{\sigma} \{\mathcal{X}_1', \ldots, \mathcal{X}_k'\}}\ [\textsc{SopB}]$$

**Figure 9** Transitions of lposets, pomsets, and sets of pomsets.

$\{a, a, b\}$, endowed with $a \prec b \prec a$, can be formalised using labelled vertices $1_a, 2_a, 3_b$ and arcs $(1, 3), (3, 2)$); such labelled digraphs are called *lposets* ("labelled partially ordered sets"). To make the formalisation of pomsets insensitive to the choice of vertices (e.g., it should not matter if we use $1_a, 2_a, 3_b$ or $4_a, 5_a, 6_b$ as labelled vertices), pomsets are ultimately defined as isomorphism classes of lposets. Henceforth, we write "events" instead of "vertices".

Let **Lpo** denote the set of all *lposets*, ranged over by $X$. Formally, an lposet is a tuple $(V, \prec, \lambda)$, where $V$ is a set of *events*, $\prec \subseteq V \times V$ is a *precedence relation* (strict partial order), and $\lambda : V \to \mathbf{\Sigma}$ is a *labelling function*, where $\mathbf{\Sigma} = \{pq!t \mid p \neq q\} \cup \{pq?t \mid p \neq q\}$ (§ 3.1). Let $\mathsf{pred}_X(v)$ and $\mathsf{succ}_X(v)$ denote the *immediate predecessors* and *successors* of $v$ in $X$. Formally, pred and succ are the smallest functions induced by the following equations:

$$\mathsf{pred}_{(V,\prec,\lambda)}(v) = \{v' \mid v \prec v' \text{ and } (v', v) \notin \{(v_1, v_3) \mid v_1 \prec v_2 \prec v_3\}\}$$
$$\mathsf{succ}_{(V,\prec,\lambda)}(v) = \{v' \mid v' \prec v \text{ and } (v, v') \notin \{(v_1, v_3) \mid v_1 \prec v_2 \prec v_3\}\}$$

Let $X_1 \parallel X_2$ and $X_1 \cdot X_2$ denote the *parallel composition* and the *sequential composition* of $X_1$ and $X_2$. Formally, $\parallel$ and $\cdot$ are the smallest functions induced by the following equations:

$$(V_1, \prec_1, \lambda_1) \parallel (V_2, \prec_2, \lambda_2) = (V_1 \cup V_2, \prec_1 \cup \prec_2, \lambda_1 \cup \lambda_2)$$
$$(V_1, \prec_1, \lambda_1) \cdot (V_2, \prec_2, \lambda_2) = (V_1 \cup V_2, \prec_1 \cup \prec_2 \cup (V_1 \times V_2), \lambda_1 \cup \lambda_2)$$

Let $X_1 \cong X_2$ denote *isomorphism equivalence* of $X_1$ and $X_2$; informally, $X_1$ and $X_2$ are isomorphic if, and only if, there exists a bijection between their sets of events that preserves their precedence relations and labelling functions.

Let $\mathbf{Pom} = \mathbf{Lpo}/\cong$ denote the set of all *pomsets*, ranged over by $\mathcal{X}$; it is the quotient set of **Lpo** by $\cong$ (i.e., a pomset is an isomorphism class of lposets). We write $[X]$ to denote the isomorphism class of $X$ (i.e., a pomset). A set of pomsets is similar to a formal language (i.e., a set of words), except that words that differ only in the order of concurrent actions are represented collectively as a single pomset instead of individually as multiple words.

**Pomsets, formally (behaviour).** A transition relation can be associated to lposets, pomsets, and sets of pomsets. Figure 9 shows the rules. Rule [Lpo] states that an lposet can reduce with symbol $\lambda(v)$ when $v$ is minimal; $v$ is removed from the reduced lposet. Rule [Pom] states that a pomset can reduce with symbol $\sigma$ when one of its lposets can.

Rule [SopA] states that a set of pomsets can reduce when one of its pomsets can; the reduced pomset is kept, while the others are removed. This formalises the idea of an *early choice*: at the start of an execution, when the first action is performed, a commitment is made to one behaviour. Alternatively, rule [SopB] states that a set of pomsets can reduce with symbol $\sigma$ when it can be split into two disjoint subsets, such that $k$ pomsets can reduce with $\sigma$, while the remaining $n - k$ pomsets cannot; the $k$ reduced pomsets are kept, while the others are removed. This formalises the idea of a *late choice*: during an execution, as

$$
\begin{aligned}
[\![\mathbf{end}]\!]_{\mathrm{SOP}} &= \{[(\emptyset, \emptyset, \emptyset)]\} \\
[\![pq\dagger\{t_i.L_i\}_{1\le i\le n}]\!]_{\mathrm{SOP}} &= \left\{ \left[ \bullet \xrightarrow{pq\dagger t_i} X \right] \,\middle|\, [X] \in [\![L_i]\!]_{\mathrm{SOP}} \text{ and } 1 \le i \le n \right\} \\
&= \{[(\{\bullet\}, \emptyset, \{\bullet \mapsto pq\dagger t_i\}) \cdot X] \mid [X] \in [\![L_i]\!]_{\mathrm{SOP}} \text{ and } 1 \le i \le n\} \\
[\![L_1 \parallel L_2]\!]_{\mathrm{SOP}} &= \{[X_1 \parallel X_2] \mid [X_1] \in [\![L_1]\!]_{\mathrm{SOP}} \text{ and } [X_2] \in [\![L_2]\!]_{\mathrm{SOP}}\} \\
[\![L_1 \cdot L_2]\!]_{\mathrm{SOP}} &= \{[X_1 \cdot X_2] \mid [X_1] \in [\![L_1]\!]_{\mathrm{SOP}} \text{ and } [X_2] \in [\![L_2]\!]_{\mathrm{SOP}}\}
\end{aligned}
$$

**Figure 10** Interpretation of local types as SOPs.

actions are performed, a commitment is gradually made towards a subset of behaviours that are still "eligible". The transition system generated by rule [SopA] is trace equivalent to the transition system generated by rule [SopB]. However, these two transition system are *not* bisimulation equivalent: the former can be simulated by the latter, but not vice versa (i.e., rule [SopB] subsumes rule [SopA]). Shortly, we will argue that late choices (i.e., rule [SopB]) are appropriate in our setting, so we will use rule [SopB] instead of rule [SopA].

**Interpretation.** Let $[\![L]\!]_{\mathrm{SOP}}$ denote the *interpretation* of local type $L$ as a SOP. Formally, $[\![\text{-}]\!]_{\mathrm{SOP}} : \mathbb{L} \to 2^{\mathbf{Pom}}$ is the smallest function induced by the equations in Figure 10. The interpretation of **end** is the SOP that contains only the empty pomset. The interpretation of $pq\dagger\{t_i.L_i\}_{1\le i\le n}$ is the SOP that contains for every $1 \le i \le n$, and for every pomset $X$ in the interpretation of $L_i$, the pomset in which a $pq\dagger t_i$-labelled event precedes all events in $X$; in the visualisation, the arrow represents precedence ($\prec$). The interpretations of $L_1 \parallel L_2$ and $L_1 \cdot L_2$ are the pairwise parallel composition and the pairwise sequential composition.

▶ **Example 13.** The following SOP is the interpretation of $L_{\mathbf{s}}$ in Example 3:

$$
[\![L_{\mathbf{s}}]\!]_{\mathrm{SOP}} = [\![(\mathbf{sb}!\mathtt{Descr} \parallel \mathbf{sb}!\mathtt{Price}) \cdot \mathbf{sb}?\{\mathtt{Acc}, \mathtt{Rej}\}]\!]_{\mathrm{SOP}}
$$

$$
= \left\{ \left[ \begin{array}{ccc} \mathbf{sb}!\mathtt{Descr} & & \\ & \searrow & \mathbf{bs}?\mathtt{Acc} \\ \mathbf{sb}!\mathtt{Price} & \nearrow & \end{array} \right], \left[ \begin{array}{ccc} \mathbf{sb}!\mathtt{Descr} & & \\ & \searrow & \mathbf{bs}?\mathtt{Rej} \\ \mathbf{sb}!\mathtt{Price} & \nearrow & \end{array} \right] \right\}
$$

▶ **Example 14.** The following SOP is the interpretation of $L_{\mathbf{m}}$ in Example 4:

$$
[\![L_{\mathbf{m}}]\!]_{\mathrm{SOP}} = [\![\mathbf{mw}_1!\mathtt{Work} \,.\, \mathbf{mw}_2!\mathtt{Work} \,.\, (\mathbf{w}_1\mathbf{m}?\mathtt{Done} \parallel \mathbf{w}_2\mathbf{m}?\mathtt{Done})]\!]_{\mathrm{SOP}}
$$

$$
= \left\{ \left[ \begin{array}{ccccc} & & & \nearrow & \mathbf{w}_1\mathbf{m}?\mathtt{Done} \\ \mathbf{mw}_1!\mathtt{Work} & \rightarrow & \mathbf{mw}_2!\mathtt{Work} & & \\ & & & \searrow & \mathbf{w}_2\mathbf{m}?\mathtt{Done} \end{array} \right] \right\}
$$

▶ **Example 15.** The following SOP is the interpretation of $L_{\mathbf{m}}$ in Example 12:

$$
[\![L_{\mathbf{m}}]\!]_{\mathrm{SOP}} = [\![\mathbf{mw}_1!\mathtt{Work} \,.\, \mathbf{mw}_2!\mathtt{Work} \,.\, \mathbf{mw}_3!\mathtt{Work} \,.\, (\mathbf{w}_1\mathbf{m}?\mathtt{Done} \parallel \mathbf{w}_2\mathbf{m}?\mathtt{Done} \parallel \mathbf{w}_3\mathbf{m}?\mathtt{Done})]\!]_{\mathrm{SOP}}
$$

$$
= \left\{ \left[ \begin{array}{ccccccc} & & & & & \nearrow & \mathbf{w}_1\mathbf{m}?\mathtt{Done} \\ \mathbf{mw}_1!\mathtt{Work} & \rightarrow & \mathbf{mw}_2!\mathtt{Work} & \rightarrow & \mathbf{mw}_3!\mathtt{Work} & \rightarrow & \mathbf{w}_2\mathbf{m}?\mathtt{Done} \\ & & & & & \searrow & \mathbf{w}_3\mathbf{m}?\mathtt{Done} \end{array} \right] \right\}
$$

(We explain the meaning of the dashed lines later.)

**This SOP avoids complication 1 in Figure 5 (i.e., state explosion):** it has linearly many events in the number of unordered receives (cf. the DFA in Example 12). ⌟

## 4.2 From SOPs to APIs

To encode SOPs as APIs, the key insight is that an input-enabled process $P$ is well-typed by local type $L$ (§ 2) if, and only if, every possible sequence of sends and receives by $P$ precisely *covers* the set of events of a pomset $\mathcal{X}$ in $[\![L]\!]_{\text{SOP}}$ (i.e., every send or receive in the sequence is an event of $\mathcal{X}$, and vice versa) and *respects* the precedence relation (i.e., the sequence is a linearisation of $\mathcal{X}$). That is, the sequence of sends and receives by $P$ must correspond to a sequence of transitions of the SOP, derived using the rules in Figure 9. The "trick", then, is to structure the API in such a way that when the compiler successfully type-checks the API's usage, it has effectively validated coverage and respectfulness. In the rest of this subsection, we show how to achieve this in Scala 3 using match types.

We proceed in three paragraphs: first, to set the stage, we present a basic encoding of SOPs for *choice-free protocols*; next, we extend the basic encoding with advanced support for *concurrent subprotocols;* last, we also extend the basic encoding with advanced support for *choice-based protocols*. We note that "advanced" pertains to the encodings, but not to the features (i.e., choices are a basic feature in theory, but when modelled as SOPs, they require an advanced encoding in practice). Furthermore, we note that the two extensions cannot be used together yet; we explain the challenges to combine them at the end of this section.

Our aim in this subsection is to convey the main ideas and insights of the encoding as clearly as possible. To this end, we focus mostly on examples (instead of presenting the general encoding schemes). We do emphasise upfront, though, that the encodings are general, as also evidenced by our tool and the examples that we distribute with it (§ 5).

**Basic encoding: choice-free protocols.** The idea behind the basic encoding of singleton SOPs is similar to the states-as-type-parameters encoding of DFAs. That is, as in § 3.3, a single generic state class is generated, combined with the usage of match types ⟨$r$⟩`$SendReturn` and ⟨$r$⟩`$RecvArgument` for the return and argument values of methods `send` and `recv`:

```
class ⟨r⟩$State[X](x: X, net: Network) extends UseOnce:
  def send[Q, E](q: Q, e: E): ⟨r⟩$SendReturn[X, Q, E] = { use; ... }
  def recv(fff: ⟨r⟩$RecvArgument[X]): ⟨r⟩$Final = { use; ... }
```

The main difference is the way in which the type parameter of ⟨$r$⟩`$State` is instantiated: whereas `N` in § 3.3 was instantiated with numeric literal types to identify states in a DFA, `X` in this section is instantiated with tuples of *boolean literal types* (e.g., in Scala, symbol "`true`" denotes both value true and a type with value true as its only inhabitant) to represent events in the current pomset. For instance, if $\{1, 2, 3\}$ is the set of events, then its representation as a tuple is `(v1, v2, v3)`, where each of `v1`, `v2`, and `v3` is either type `true` or type `false`. Intuitively, if an event is represented as `true`, then it is still *enabled* (i.e., it has not happened yet); if it is represented as `false`, then it is *disabled* (i.e., it has happened already). We note that "state" in this section should be understood as "the current pomsets in the SOP".

Match type ⟨$r$⟩`$SendReturn` has three type parameters, called `X`, `Q`, and `E`, which represent the current pomset $\mathcal{X}$, the receiver $q$, and the type $t$ of the value to send. If the pomset has a send transition $\mathcal{X} \xrightarrow{rq!t} \mathcal{X}'$ (derived using the rules in Figure 9 through a new auxiliary match type ⟨$r$⟩`$Pom$Send`), then the compiler succeeds to reduce ⟨$r$⟩`$SendReturn[`⟨$\mathcal{X}$⟩`,`⟨$q$⟩`,`⟨$t$⟩`]` to ⟨$r$⟩`$State[`⟨$\mathcal{X}'$⟩`]`; this is the type of the fresh successor state object after sending. Match type ⟨$r$⟩`$RecvArgument` has one type parameter, called `X`, which represents the current pomset $\mathcal{X}$. If the pomset has receive transitions $\mathcal{X} \xrightarrow{p_1 r?t_1} \mathcal{X}'_1, \ldots, \mathcal{X} \xrightarrow{p_n r?t_n} \mathcal{X}'_n$ (derived using the rules in Figure 9 through a new auxiliary match type ⟨$r$⟩`$Pom$Recv`), then the compiler succeeds to reduce ⟨$r$⟩`$RecvArgument[`⟨$\mathcal{X}$⟩`]` to a tuple of function types each of which is of the form `(`⟨$p_i$⟩`,`⟨$t_i$⟩`,`⟨$r$⟩`$State[`⟨$\mathcal{X}'$⟩`]) => `⟨$r$⟩`$Final`; these are the types of the continuations after receiving. The following examples demonstrate these match types.

▶ **Example 16.** The following API (excerpt) is the encoding of $[\![L_{\mathtt{m}}]\!]_{\mathrm{SOP}}$ in Example 14. We present it in two steps. First, we define auxiliary match types `M$Pom$Send` and `M$Pom$Recv` to derive transitions of the single pomset in $[\![L_{\mathtt{m}}]\!]_{\mathrm{SOP}}$ using rules [Lpo] and [Pom] in Figure 9. Second, we use these auxiliary match types to define `M$SendReturn` and `M$RecvArgument`.

1. The first listing shows the two auxiliary match types:

```
type M$Pom$Send[X, Q, E] = (X, Q, E) match
  case ((true, v2, v3, v4), W1, Work)    => (false, v2, v3, v4)
  case ((false, true, v3, v4), W2, Work) => (false, false, v3, v4)
  case Any                               => Error

type M$Pom$Recv[X, P, E] = (X, P, E) match
  case ((v1, false, true, v4), W1, Done) => (v1, false, false, v4)
  case ((v1, false, v3, true), W2, Done) => (v1, false, v3, false)
  case Any                               => Error
```

   Every case in match type `M$Pom$Send` encodes a send transition of the pomset represented by `X`, with `Q` as the receiver and `E` as the type of the value to send, derived using rules [Lpo] and [Pom] in Figure 9. The first case states that if `X` matches `(true, v2, v3, v4)`, where `v2`, `v3`, and `v4` are local type variables that are bound by matching (i.e., the first event in the pomset has not yet happened and is still enabled, hence `true`; the other events are irrelevant), and if `Q` and `E` match `W1` and `Work`, then the pomset can make a transition to `(false, v2, v3, v4)` when work is sent to worker 1 (i.e., the first event has happened and is disabled, hence `false`; the other events are unaffected). We note that it is more convenient to "set" events to `false` instead of removing them as in rule [Lpo]. The second case is similar, except that it imposes a precedence constraint: to match `X`, its first element must be `false`. That is, for the second event to happen, the first event must have already happened (i.e., the second event must have become minimal to satisfy the premise of rule [Lpo]). The third case states that if the first two cases do not apply, then a send of a value of type `E` to receiver `Q` cannot happen in `X`. Here, `Error` is a special type that we use to explicitly represent "failed reduction"; it is dealt with in the next step.
   Match type `M$Pom$Recv` is similar, but for receives instead of sends.

2. The second listing shows ⟨*r*⟩`$SendReturn` and ⟨*r*⟩`$RecvArgument`:

```
type M$SendReturn[X, Q, E] =          /* |-then-| */
  IfThenElse[IsError[M$Pom$Send[X, Q, E]], Unit, M$State[M$Pom$Send[X, Q, E]]]
        /* |---if----------------------|      |---else--------------------| */

type M$RecvArgument[X] =
  Simplify[(IfThenElse[IsError[M$Pom$Recv[X, W1, Done]], Unit,
                       (W1, Done, M$State[M$Pom$Recv[X, W1, Done]]) => M$Final],
            IfThenElse[IsError[M$Pom$Recv[X, W2, Done]], Unit,
                       (W2, Done, M$State[M$Pom$Recv[X, W2, Done]]) => M$Final])]
```

   To reduce match type `M$SendReturn[X, Q, E]`, the compiler checks if `M$Pom$Send[X, Q, E]` reduces to `Error`, using "utility match types" `IfThenElse` and `IsError`. If it does (i.e., a send of `E` to `Q` in `X` cannot happen), then the type of the fresh successor state object is `Unit` (i.e., to indicate that something is wrong). In contrast, if it does not reduce to `Error` (i.e., the send can happen), then the type of the fresh successor state is proper.
   To reduce match type `M$RecvArgument[X]`, the compiler uses a similar approach to determine for every possible receive (characterised in terms of the sender and the type of the value) if it can happen or not in `X`. The result is a tuple that consists of either a proper continuation function or `Unit` for every possible receive; the `Unit`s are subsequently removed using utility match type `Simplify`.

We note that exactly the same function that implements the master in Example 9 is also well-typed using the API in this example. For details, see: `https://scastie.scala-lang.org/3PyXcwBKS2argRL9oPlspg`. ⌟

**Advanced encoding: concurrent subprotocols.** In this paragraph, we extend the basic encoding with an advanced feature that significantly subsumes DFA-based API generation: the ability to *fork* and *join* pomsets. Intuitively, when a fork is performed, the pomset is "broken" into "shards", each of which can evolve independently of the others; when a join is performed, shards are "glued" back together. In terms of typing rules, intuitively, forking corresponds to rule [Par] in Figure 7, while joining corresponds to rule [Seq]. In the DFA encodings, rule [Par] has no corresponding representation, as the parallel structure is lost in translation. In contrast, in the SOP encodings, the parallel structure is still there and thus can be exploited. This extension is crucial to effectively support concurrent subprotocols (i.e., avoid branch explosion; Figure 5), as will be demonstrated shortly in Example 18.

A first essential ingredient that we need, is an analysis procedure to statically identify shards. To explain it, suppose that $[X] = [(V, \prec, \lambda)]$ is a pomset. The idea is to partition $V$ into two kinds of subsets, each of which forms a shard:

**Join shards:** If $v \in V$, and if $|\mathsf{pred}_X(v)| > 1$, then $\{v\}$ forms a *join shard*.

**Sequential shards:** If $v \preceq v_1 \prec \cdots \prec v_n \preceq v'$, and if $|\mathsf{succ}_X(v_i) \cup \mathsf{pred}_X(v_i)| \leq 2$, for every $1 \leq i \leq n-1$ (i.e., $v_1, \ldots, v_n$ is a chain), and if either $|\mathsf{succ}_X(v)| > 1$ or $|\mathsf{pred}_X(v_1)| \neq 1$ (i.e., either $v_1$ is preceded by "fork event" $v$, or it is an "initial event" itself), and if either $|\mathsf{pred}_X(v') > 1|$ or $|\mathsf{succ}_X(v_n)| \neq 1$ (i.e., $v_n$ is succeeded by "join event" $v'$, or it is a "final event" of "fork event" itself), then $\{v_1, \ldots, v_n\}$ forms a *sequential shard*. That is, a sequential shard is a longest chain of events, optionally preceded by a fork event (if $v_1$ is not initial), and optionally succeeded by a join event (if $v_n$ is not final or fork).

For instance, the dashed lines in Example 15 visualise four shards. We note that identification of shards is computationally easy: it can be done in polynomial time (in the size of the pomset), using standard graph traversal techniques.

Using this concept of shards, forking and joining generally works as follows:

- If an event $e$ has multiple immediate successors, and if every immediate predecessor has already happened, and if every immediate successor has not yet happened, then the "old pomset" can be forked into "new sub-pomsets" by breaking it into shards: for every immediate successor of $e$, there is a new sub-pomset that consists of all sequential shards that are reachable from $e$ without passing through a join shard; the join shards and all shards beyond are temporarily disabled. That is, each new sub-pomset can evolve independently within the boundaries of its shards, but to go further, a join is needed first. We also note that sub-pomsets can be recursively forked.
- If an event $e$ has multiple immediate predecessors, and if every immediate predecessor has already happened, then the sub-pomsets can be joined by gluing their shards, re-enabling the temporarily disabled join shards and all shards beyond.

To incorporate these concepts, we extend the basic encoding as follows:

- Class $\langle r \rangle$`$State` is extended with a method `fork`. At execution time, when `fork` is called, a fork is performed and fresh successor state objects are returned; they can be used independently of each other (i.e., concurrent subprotocols). At compile-time, to check that this method call is actually allowed in the current pomset, the compiler tries to reduce a match type: if it succeeds, the call is allowed; if it fails, it is not. More precisely, the idea is to use a match type $\langle r \rangle$`$ForkReturn` for the *return value*; it has one type parameter, called `X`, which represents the current pomset $\mathcal{X}$. If $\mathcal{X}$ can be forked into

sub-pomsets $\mathcal{X}_1, \ldots, \mathcal{X}_n$, then the compiler succeeds to reduce ⟨r⟩`$ForkReturn[`⟨$\mathcal{X}$⟩`]` to a tuple `(`⟨r⟩`$State[`⟨$\mathcal{X}_1$⟩`],...,`⟨r⟩`$State[`⟨$\mathcal{X}_n$⟩`])`; these are the types of the fresh successor state objects after forking. In contrast, if $\mathcal{X}$ cannot be forked, then the compiler fails to reduce and yields an error. Thus, `fork` looks as follows:

```
def fork(): M$ForkReturn[X] = { use; ... }
```

We note that forking a pomset also counts as "usage": after calling `fork`, the state object should not be used again to send or receive.

- Class ⟨r⟩`$State` is also extended with a static method `join` for every "join event" in the pomset, using overloading. At execution time, when `join` is called, multiple final state objects for sub-pomsets are collected (passed as arguments) and a single fresh successor state object is returned. At compile-time, to check that this method call is actually allowed, the compiler checks if the types of the actual parameters match the types of the formal parameters of one of the overloaded `join` methods.
- To conveniently distinguish sub-pomsets, the representation `X` of the current pomset (i.e., tuple of boolean literal types) is extended with an extra element, namely a *fork identifier* (i.e., numeric literal type): if the current pomset was previously forked off (i.e., `X` actually represents a sub-pomset), then the fork identifier is a numeric literal type that identifies the immediate successor of the "fork event"; else, the fork identifier is `0`.
- To enable method `recv` in class ⟨r⟩`$State` to return different final states for different sub-pomsets, the type of its return value is refined to depend on fork identifiers. The idea is to use an auxiliary match type ⟨r⟩`$Pom$Final` to derive final states: it has one type parameter, called `X`, which represents the current sub-pomset. Based on the fork identifier in `X`, the compiler reduces ⟨r⟩`$Pom$Final[X]` to a tuple in which all events of the enabled shards are represented as `false` (i.e., they have happened). Thus, `recv` looks as follows:

```
def recv(fff: ⟨r⟩$RecvArgument[X]): ⟨r⟩$State[⟨r⟩$Pom$Final[X]] = { use; ... }
```

The following examples demonstrate these match types.

▶ **Example 17.** The following API (excerpt) is the encoding of $[\![L_{\mathtt{m}}]\!]_{\mathrm{SOP}}$ in Example 14. We present it in three steps. First, we define auxiliary match types `M$Pom$Send` and `M$Pom$Recv` (similar to Example 16) and auxiliary match type `M$Pom$Final` (new). Second, we use these auxiliary match types to define `M$SendReturn` and `M$RecvArgument` (similar to Example 16) and `M$Fork$Return` (new). Third, we define static method `join`.

1. The first listing shows the three auxiliary match types:

```
type M$Pom$Send[X, Q, E] = (X, Q, E) match
  case ((n, true, v2, v3, v4), W1, Work)    => (n, false, v2, v3, v4)
  case ((n, false, true, v3, v4), W2, Work) => (n, false, false, v3, v4)
  case Any                                  => Error

type M$Pom$Recv[X, P, E] = ...

type M$Pom$Final[X] = X match
  case (0, v1, v2, v3, v4) => (0, false, false, false, false)
  case (3, v1, v2, v3, v4) => (3, v1, v2, false, v4)
  case (4, v1, v2, v3, v4) => (4, v1, v2, v3, false)
```

Match types `M$Pom$Send` and `M$Pom$Recv` are the same as in Example 16, except that tuple `X` has an extra element `n`, namely a numeric fork identifier. Match type `M$Pom$Final` is new: it is used to infer when a sub-pomset has fully evolved (i.e., within the boundaries of its shards, but not beyond). For instance, the second case states that the sub-pomset identified by `3` is final when the event identified by `3` has happened.

2. The second listing shows `M$SendReturn`, `M$RecvArgument`, and `M$Fork$Return`:

```
type M$SendReturn[X, Q, E] = ...

type M$RecvArgument[X] =
  Simplify[(IfThenElse[IsError[M$Pom$Recv[X, W1, Done]], Unit,
                       (W1, Done, M$State[M$Pom$Recv[X, W1, Done]]) =>
                         M$State[M$Pom$Final[X]]],
            ...)]

type M$ForkReturn[X] = X match
  case (0, v1, v2, true, true) => (
    M$State[(3, v1, v2, true, false)], M$State[(4, v1, v2, false, true)])
```

Match types `M$SendReturn` and `M$RecvArgument` are the same as in Example 16, except that the return types of the continuations as computed by `M$RecvArgument` depend on the fork identifier in `X`, using match type `M$Pom$Final`. Match type `M$ForkReturn` is new: it is used to infer which old pomsets can be forked into which tuples of new sub-pomsets. In this example, there is only one case. It states that if the old pomset is unforked, and if the third and fourth events have not yet happened, then it can be forked into two new sub-pomsets: one for the third event and one for the fourth event.

In general, match type $\langle r \rangle$`$ForkReturn` has as many cases as there are "fork events" in the pomset. That is, we allow a pomset to be forked only right after a fork event, before any event of the immediate successors has happened (if we would allow it also "later", then we would need to generate exponentially many cases).

3. The third listing shows `join`:

```
object M$State: // static methods of class M$State
  def join(
    s1: M$State[(3, false, false, false, false)],
    s2: M$State[(4, false, false, false, false)]
  ): M$State[(0, false, false, false, false)] = ...
```

Method `join` is needed in this example to provide a final state object with a unique type; it is implicitly present in every pomset with more than one maximal element.

In general, $\langle r \rangle$`$State` has as many `join` methods as there are "join events" in the pomset.

We note that exactly the same function that implements the master in Example 9 is also well-typed using the API in this example. Thus, the fork–join extension of the basic encoding is backwards-compatible. To additionally demonstrate forking and joining, the following well-typed function implements the master as well:

```
def master(s: M$Initial): M$Final =
  val (s1, s2) = s.send(W1, new Work).send(W2, new Work).fork()
  val f1 = Future { s1.recv((_, _, s) => { println("#1"); s }) }
  val f2 = Future { s2.recv((_, _, s) => { println("#2"); s }) }
  Await.result(for { t1 <- f1; t2 <- f2 } yield M$State.join(t1, t2), ...)
```

In the first line of the body, the two sends are sequentially performed as before; after that, the pomset is forked into two pomsets, divided over successor state objects `s1` and `s2`. On the second and third line, the two receives are performed concurrently using two *futures* (built-in Scala mechanism for asynchronous programming, using a default thread pool). On the fourth line, the results of the futures are awaited. Any change in the order of the actions (sends, receives, fork, join) results in a compile-time error. For details, see: `https://scastie.scala-lang.org/RoIs430cTsS3w9wh8GC74w`. ⌟

▶ **Example 18.** The following well-typed function implements the master (with three workers), using an API that encodes the SOP in Example 15, including the fork–join extension.

```
def master(s: M$Initial): M$Final =
  val (s1, s2, s3) = s.send(W1, new Work).send(W2, new Work).send(W3, new Work).fork()
  val f1 = Future { s1.recv((_, _, s) => { println("#1"); s }) }
  val f2 = Future { s2.recv((_, _, s) => { println("#2"); s }) }
  val f3 = Future { s3.recv((_, _, s) => { println("#3"); s }) }
  Await.result(
    for { t1 <- f1; t2 <- f2; t3 <- f3 } yield M$State.join(t1, t2, t3), ...)
```

**This implementation avoids complication 2 in Figure 5 (i.e., branch explosion):** it has linearly many branches in the number of unordered receives (cf. the implementation in Example 12, which has exponentially many branches). For details, see: `https://scastie.scala-lang.org/MUEFr4ZvSEyFepAJWAKh3g`. ⌟

**Advanced encoding: choice-based protocols.** In the previous paragraphs, we presented the basic encoding of singleton SOPs and the fork–join extension. In this last paragraph, we present another extension of the basic encoding to support non-singleton SOPs.

Intuitively, a non-singleton SOP $\{\mathcal{X}_1, \ldots, \mathcal{X}_n\}$ represents a *choice* among $n$ possible local behaviours. The trouble is that both at compile-time and initially at execution-time, it is still unknown which of the $n$ pomsets will actually be chosen (e.g., the seller in the seller-buyer protocol initially does not know yet if the buyer will accept or reject the offer). In particular, this observation entails that we cannot "compositionally" apply the basic encoding to each of the $n$ pomsets and require a process to choose one of them in the beginning; generally, there is no way in which the process can make such an early choice upfront. That is, rule [SopA] in Figure 9, which formalises the idea of early choices, is too inflexible.

Instead, a process needs to keep its options open for as long as possible. To achieve this, in accordance with rule [SopB] in Figure 9, which formalises the idea of late choices, the plan is to incrementally refine the set of "eligible pomsets" (to become the chosen one), by accumulating knowledge during the execution of the process. That is (cf. rule [SopB]), initially, all pomsets are eligible; subsequently, every time a send or receive happens, all eligible pomsets *that do not allow the action to happen*, become ineligible. In this way, when the process terminates, coverage and respectfulness are satisfied if, and only if, all events of at least one remaining eligible pomset have happened. It is straightforward to perform such an incremental eligibility analysis dynamically; the challenge is to "trick" the compiler into doing it statically. To achieve this, we extend the basic encoding as follows:

- Class ⟨$r$⟩`$State` is extended with additional type parameters: instead of just `X`, which represents the one pomsets in a singleton SOP, it has `X1,...,Xn`, which represent the $n$ pomsets in a non-singleton SOP.
- Similarly, match types ⟨$r$⟩`$SendReturn` and ⟨$r$⟩`$RecvArgument` are extended with additional type parameters `X1,...,Xn`. Furthermore, the definitions of these match types are extended to derive send and receive transitions of the non-singleton SOP represented by `X1,...,Xn` using rule [SopB] in Figure 9. We note that, essentially, the basic encoding of singleton SOPs is a special case of the advanced encoding of non-singleton SOPs.

The following example demonstrates these match types.

▶ **Example 19.** The following API (excerpt) is the encoding of $[\![L_{\mathbf{s}}]\!]_{\mathrm{SOP}}$ in Example 13. We present it in two steps. First, we define auxiliary match types `S$Pom1$Send` and `S$Pom1$Recv` to derive transitions of the "left" pomset in $[\![L_{\mathbf{s}}]\!]_{\mathrm{SOP}}$, and auxiliary match types `S$Pom2$Send` and `S$Pom2$Recv` to derive transitions of the "right" pomset in $[\![L_{\mathbf{s}}]\!]_{\mathrm{SOP}}$, using rules [Lpo] and [Pom] in Figure 9 (similar to Example 16). Second, we use these auxiliary match types to define `S$SendReturn` and `S$RecvArgument`.

1. The first listing shows the four auxiliary match types:

```
type S$Pom1$Send[X, Q, E] = (X, Q, E) match
  case ((true, v2, v3), B, Descr) => (false, v2, v3)
  case ((v1, true, v3), B, Price) => (v1, false, v3)
  case Any                        => Error

type S$Pom1$Recv[X, Q, E] = (X, P, E) match
  case ((false, false, true), B, Acc) => (false, false, false)
  case Any                            => Error

type S$Pom2$Send[X, Q, E] = (X, Q, E) match
  case ((true, v2, v3), B, Descr) => (false, v2, v3)
  case ((v1, true, v3), B, Price) => (v1, false, v3)
  case Any                        => Error

type S$Pom2$Recv[X, P, E] = (X, P, E) match
  case ((false, false, true), B, Rej) => (false, false, false)
  case Any                            => Error
```

   Conceptually, there is nothing new here relative to the basic encoding, except that we should be more precise about the meaning of `Error`: it stands for "ineligible".

2. The second listing shows `S$SendReturn` and `S$RecvArgument`:

```
type S$SendReturn[X1, X2, Q, E] =
  IfThenElse[
    And[IsError[S$Pom1$Send[X1, Q, E]], IsError[S$Pom2$Send[X2, Q, E]]], Unit,
    S$State[S$Pom1$Send[X1, Q, E], S$Pom2$Send[X2, Q, E]]]

type S$RecvArgument[X1, X2] =
  Simplify[(
    IfThenElse[
      And[IsError[S$Pom1$Recv[X1, B, Acc]], IsError[S$Pom2$Recv[X2, B, Acc]]], Unit,
      (B, Acc, S$State[S$Pom1$Recv[X1, B, Acc], S$Pom2$Recv[X2, B, Acc]]) =>
        S$Final],
    ...)]
```

   To reduce match type `S$SendReturn[X1, X2, Q, E]`, where `X1` and `X2` represent the two pomsets in the SOP, the compiler checks if `M$Pom1$Send[X1, Q, E]` and `M$Pom2$Send[X2, Q, E]` reduce to `Error`. If they do (i.e., a send of `E` to `Q` can happen neither in `X1` nor in `X2`), then the type of the fresh successor state object is `Unit` (i.e., all pomsets have become ineligible). In contrast, if they do not both reduce to `Error` (i.e., the send can happen in `X1`, or in `X2`, or in both), then the type of the fresh successor state is proper; it is formed by evolving both `X1` and `X2`. There are three cases:

   - If `X1` and `X2` are bound to a non-`Error` type, and they evolve to non-`Error` types, then the corresponding type parameters of the successor remain bound to non-`Error` types. That is, both pomsets remain eligible.
   - If `X1` (resp. `X2`) is bound to a non-`Error` type, but it evolves to `Error`, then the corresponding type parameter of the successor becomes bound to `Error` as well. That is, the first pomset (resp. second pomset) becomes ineligible. We note that it is more convenient to "set" pomsets to `Error` instead of removing them as in rule [SopB].
   - If `X1` (resp. `X2`) is bound to `Error`, then it "evolves" again to `Error` (see previous listing), so the corresponding type parameter of the successor remains bound to `Error` as well. That is, the first pomset (resp. second pomset) remains ineligible.

   To reduce match type `M$RecvArgument[X1, X2]`, the compiler uses a similar approach to determine for every possible receive if it can happen or not in `X1` and `X2`.

We note that exactly the same function that implements the master in Example 8 is also well-typed using the API in this example. For details, see: `https://scastie.scala-lang.org/WN5ZmEMcRh2ecUMaCgvjPA`. ⌟

It remains a largely open question how to use both the fork–join extension of the previous paragraph, and the choice extension of this paragraph, together. The only situation in which we know how to do it, is when a non-singleton SOP has evolved in such a way that only one of its pomsets has remained eligible; in that case, it has effectively become a singleton SOP, to which the fork–join extension readily applies. The main challenge to also support forking and joining of SOPs with multiple eligible pomsets is that we need to devise a mechanism to control non-forked pomsets. For instance, suppose that we have a SOP $\{[X_1], [X_2]\}$ (both eligible), and suppose that $[X_1]$ can be forked into $[X_1^\dagger]$ and $[X_1^\ddagger]$. By naively performing the fork, we get two fresh successor state objects: one for $\{[X_1^\dagger], [X_2]\}$ and one for $\{[X_1^\ddagger], [X_2]\}$. However, without an additional control mechanism, the events in $[X_2]$ are now allowed to be executed *twice*. Solving this non-trivial problem is part of future work.

## 5 Tool Support: Pompset

We developed a tool, called Pompset (portmanteau of `pom_set` and `__mps_t`), to automatically generate SOP-based APIs, according to the workflow in Figure 4. More precisely, Pompset consumes a global type as input and produces a set of APIs as output as follows:

1. **From global type to local types:** First, the global type is parsed to a "global AST". Next, for every role that occurs in the global AST, the global AST is projected to a "local AST" in accordance with § 2.

2. **From local types as SOPs:** Every local AST is interpreted to a "SOP data structure" in accordance with § 4.1.

3. **From SOPs to APIs:** Every SOP data structure is encoded as an API in accordance with § 4.2, including the ability to fork–join SOPs. (For practical/engineering reasons, though, the generated code is not completely identical, but it follows the same ideas and insights and works morally the same as in the examples.)

   Technically, the encoding is carried out by filling generic templates with specific data from the SOP under consideration. We refer to our artefact (published in DARTS), for details of the templates and the filling process; it includes source code and build instructions. Furthermore, it includes additional examples to demonstrate the generality of the implemented encoding scheme.

   We note that generated APIs also consist of functions to spawn processes and transparently set up the underlying communication infrastructure (i.e., transport abstraction). The latter is based on shared-memory channels, but it could work equally well with TCP channels; it just requires additional engineering effort.

The guarantees that APIs generated by Pompset provide at compile-time, are as usual (§ 3.2): deadlock freedom and protocol compliance, modulo non-linear usage of state objects (checked at execution-time), and modulo uncontrollable sources of non-terminating/exceptional behaviour. In addition to API generation as presented so far, Pompset also offers *enhanced error messages* and *additional pomset support* to improve both the usability of the generated APIs and the usefulness of Pompset; we describe these features in [8, § A].

Pompset is written in Scala, open source, and it has a browser-based graphical user interface; we provide a screenshot in [8, § B].

# 6 Conclusion

In this paper (pearl), we revisited and revised the API generation approach to support the MPST method in practice. Regarding the "revisitation", using Scala 3, we presented two versions of the existing DFA-based API generation: states-as-classes (existing) and states-as-type-parameters (new, by leveraging match types in Scala 3). Regarding the "revision", we presented a new SOP-based API generation (again, by leveraging match types in Scala 3). Through this fresh perspective, we showed how to effectively support concurrent subprotocols for the first time in MPST practice. The SOP-based version is incorporated in a new tool.

Regarding choices, DFA-based and SOP-based API generation are equally expressive. However, the DFA approach supports loops, which the SOP approach currently does not. In contrast, the SOP approach supports forking/joining of subprotocols, which the DFA approach does not. Thus, for now, a trade-off needs to be made when choosing which encoding to use, but our vision is that the SOP approach has the potential to subsume the DFA approach (when the type system of the host language supports a kind of match types).

## 6.1 Related Work

**Local types as DFAs.** The idea to interpret local types as DFAs was conceived by Deniélou and Yoshida [11,12], within the framework of *communicating finite state machines* (CFSM) [5]. A central notion in this work is *multiparty compability*: it is used to provide a sound and complete characterisation between global types and *systems* (i.e., parallel compositions of DFAs that communicate through asynchronous channels). Multiparty compatibility was further studied and generalised in subsequent work, to cover timed behaviour [4], more flexible choice [28], and non-synchronisability [29].

**DFAs as APIs.** The idea to encode DFAs as APIs was conceived by Hu and Yoshida [20,21], for Java. The approach has subsequently been used in combination with numerous other programming languages as well, including F# [33], F$^\star$ [46], Go [7], OCaml [45], PureScript [25], Rust [27], Scala [37], and TypeScript [31]. In many of these works, distinguished capabilities of the type system of "the host" are leveraged to offer additional compile-time guarantees and/or support MPST extensions. For instance, Neykova et al. and Zhou et al. use type providers in F# and refinement types in F$^\star$ to generate APIs that support MPST-based refinement [33,46], while King et al. and Lagaillardie et al. use indexed monads in PureScript and ownership types in Rust to support static linearity [25,27].

Alternative approaches (i.e., not based on API generation) to apply the MPST method in combination with mainstream programming languages include the work of Imai et al. [22] (for OCaml), the work of Harvey et al., Kouzapas et al., and Voinea et al. [17,26,44] (for Java, using a typestate extension), and the work of Scalas et al. [40,41] (for Scala, using an external model checker). Furthermore, there exist approaches to apply the MPST method that rely on *monitoring* and/or *assertion checking* at execution-time [2,3,9,16,32,33]. The motivation is that in practice, some distributed components of a system might not be amenable to static type-checking (e.g., the source code is unavailable), but they can be dynamically monitored for compliance.

**Local types as pomsets.** The idea to encode local types as pomsets was conceived by Guanciale and Tuosto [14], in a continuation of earlier work on pomset-based semantics of global types [43]. A key contribution of Guanciale and Tuosto is a sound and complete procedure to determine if a SOP-interpretation of a global type is *realisable* as a collection

of SOP-interpretations of the global type's projections; most procedures in the MPST literature are only sound. The PomCho tool [15] supports analysis (including counterexample generation), visualisation, and projection of pomsets. The crucial difference with our tool is that PomCho cannot generate APIs.

## 6.2 Future Work

We intend to demonstrate the potential of SOP-based API generation with this paper, and we believe that it can become the starting point for many improvements to the current design. Concretely, we are pursuing two pieces of future work.

First, as explained at the end of § 4.2, fork–join support and choice support can be used together only to a limited extent. We are currently approaching this open problem from two angles: on the practical side, we try to devise a mechanism to control non-forked pomsets, without changing the underlying foundations; meanwhile, on the theoretical side, we are studying an alternative pomset-based version of MPST theory that should make choices simpler to support (inspired by *branching automata* [30] and *pomset automata* [24]).

Second, our current version of SOP-based API generation does not support loops. This open problem is foundational: in the same way that Kleene star gives rise to infinite regular languages of finite words, a looping construct in the grammar of global/local types would give rise to infinite sets of finite pomsets. In theory, this is fine; in practice, it is not (i.e., generated APIs would need to be infinite as well). Solving this problem is another reason for us to study an alternative pomset-based version of MPST theory, in which loops can be represented finitely. We expect the key ideas and insights of SOP-based API generation in this paper to remain applicable, though.

## References


1 Olivier Blanvillain, Jonathan Immanuel Brachthäuser, Maxime Kjaer, and Martin Odersky. Type-level programming with match types. *Proc. ACM Program. Lang.*, 6(POPL):1–24, 2022.

2 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, 669:33–58, 2017.

3 Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2010.

4 Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *CONCUR*, volume 42 of *LIPIcs*, pages 283–296. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

5 Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

6 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012.

7 David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: statically-typed endpoint apis for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL):29:1–29:30, 2019.

8 Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. Api generation for multiparty session types, revisited and revised using scala 3 (full version). Technical Report OUNL-CS-2022-03, Open University of the Netherlands, 2022.

9 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods Syst. Des.*, 46(3):197–225, 2015.

10 Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *POPL*, pages 435–446. ACM, 2011.

11 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012.

12 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In *ICALP (2)*, volume 7966 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2013.

13 Jay L. Gischer. The equational theory of pomsets. *Theor. Comput. Sci.*, 61:199–224, 1988.

14 Roberto Guanciale and Emilio Tuosto. Realisability of pomsets. *J. Log. Algebraic Methods Program.*, 108:69–89, 2019.

15 Roberto Guanciale and Emilio Tuosto. Pomcho: A tool chain for choreographic design. *Sci. Comput. Program.*, 202:102535, 2021.

16 Ruben Hamers and Sung-Shik Jongmans. Discourje: Runtime verification of communication protocols in clojure. In *TACAS (1)*, volume 12078 of *Lecture Notes in Computer Science*, pages 266–284. Springer, 2020.

17 Paul Harvey, Simon Fowler, Ornela Dardha, and Simon J. Gay. Multiparty session types for safe runtime adaptation in an actor language. In *ECOOP*, volume 194 of *LIPIcs*, pages 10:1–10:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.

19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.

20 Raymond Hu and Nobuko Yoshida. Hybrid session verification through endpoint API generation. In *FASE*, volume 9633 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 2016.

21 Raymond Hu and Nobuko Yoshida. Explicit connection actions in multiparty session types. In *FASE*, volume 10202 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2017.

22 Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. Multiparty session programming with global protocol combinators. In *ECOOP*, volume 166 of *LIPIcs*, pages 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

23 Sung-Shik Jongmans and Nobuko Yoshida. Exploring type-level bisimilarity towards more expressive multiparty session types. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 251–279. Springer, 2020.

24 Tobias Kappé, Paul Brunet, Bas Luttik, Alexandra Silva, and Fabio Zanasi. On series-parallel pomset languages: Rationality, context-freeness and automata. *J. Log. Algebraic Methods Program.*, 103:130–153, 2019.

25 Jonathan King, Nicholas Ng, and Nobuko Yoshida. Multiparty session type-safe web development with static linearity. In *PLACES@ETAPS*, volume 291 of *EPTCS*, pages 35–46, 2019.

26 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo: A session type toolchain for java. *Sci. Comput. Program.*, 155:52–75, 2018.

27 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Implementing multiparty session types in rust. In *COORDINATION*, volume 12134 of *Lecture Notes in Computer Science*, pages 127–136. Springer, 2020.

28 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL*, pages 221–232. ACM, 2015.

29 Julien Lange and Nobuko Yoshida. Verifying asynchronous interactions via communicating session automata. In *CAV (1)*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 2019.

30 Kamal Lodaya and Pascal Weil. Series-parallel languages and the bounded-width property. *Theor. Comput. Sci.*, 237(1-2):347–380, 2000.

31 Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. Communication-safe web programming in typescript with routed multiparty session types. In *CC*, pages 94–106. ACM, 2021.

32 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects Comput.*, 29(5):877–910, 2017.

33 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in f#. In *CC*, pages 128–138. ACM, 2018.

34 Rumyana Neykova and Nobuko Yoshida. Let it recover: multiparty protocol-induced recovery. In *CC*, pages 98–108. ACM, 2017.

35 Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017.

36 Vaughan R. Pratt. Modeling concurrency with partial orders. *Int. J. Parallel Program.*, 15(1):33–71, 1986.

37 Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. A linear decomposition of multiparty sessions for safe distributed programming. In *ECOOP*, volume 74 of *LIPIcs*, pages 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

38 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *ECOOP*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

39 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019.

40 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Effpi: verified message-passing programs in dotty. In *SCALA@ECOOP*, pages 27–31. ACM, 2019.

41 Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In *PLDI*, pages 502–516. ACM, 2019.

42 Jesse A. Tov and Riccardo Pucella. Stateful contracts for affine types. In *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 550–569. Springer, 2010.

43 Emilio Tuosto and Roberto Guanciale. Semantics of global view of choreographies. *J. Log. Algebraic Methods Program.*, 95:17–40, 2018.

44 A. Laura Voinea, Ornela Dardha, and Simon J. Gay. Typechecking java protocols with [st]mungo. In *FORTE*, volume 12136 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2020.

45 Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In *FCT*, volume 12867 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2021.

46 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA):148:1–148:30, 2020.