ST4MP: A Blueprint of Multiparty Session Typing for Multilingual Programming

Sung-Shik Jongmans[1,2] and José Proença[3]

[1] Open University of the Netherlands
[2] Centrum Wiskunde & Informatica (CWI), NWO-I
[3] CISTER, ISEP, Polytechnic Institute of Porto

Abstract. Multiparty session types (MPST) constitute a method to simplify construction and analysis of distributed systems. The idea is that well-typedness of processes at compile-time (statically) entails deadlock freedom and protocol compliance of their sessions of communications at execution-time (dynamically).
In practice, the premier approach to apply the MPST method in combination with mainstream programming languages has been based on API generation. However, existing MPST tools support only unilingual programming (homogeneity), while many real-world distributed systems are engineered using multilingual programming (heterogeneity).
In this paper, we present a blueprint of ST4MP: a tool to apply the MPST method in multilingual programming, based on API generation.

1 Introduction

Construction and analysis of distributed systems is difficult. Challenges include:

- To implement protocols among roles/participants, by programming *multiparty sessions* of *communicating processes*.
- 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) [17,18] 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 together, collectively, from their shared perspective (e.g., "first, a number from Alice to Bob; second, 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 alone, individually, from its own perspective (e.g., for Bob, "first, he receives a number from Alice; second, he sends a boolean to Carol).

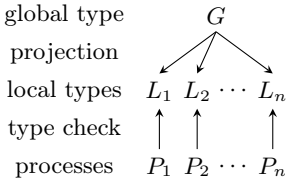


Fig. 1: MPST method

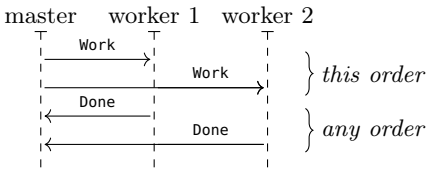


Fig. 2: Master–workers protocol (Example 1)

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 example demonstrates the MPST method.

Example 1. In the master–workers protocol, visualised in Figure 2, first, the *master* ($\mathtt{m}$) tells n *workers* ($\mathtt{w}_1, \ldots, \mathtt{w}_n$) to perform work (`Work`), in that order; second, the workers tell the master that they are done (`Done`), in any order.

The following global type specifies the protocol ($n = 2$):

$$G = \mathtt{m} \to \mathtt{w}_1 : \mathtt{Work} \,.\, \mathtt{m} \to \mathtt{w}_2 : \mathtt{Work} \,.\, (\mathtt{w}_1 \to \mathtt{m} : \mathtt{Done} \,.\, \mathbf{end} \;|||\; \mathtt{w}_2 \to \mathtt{m} : \mathtt{Done} \,.\, \mathbf{end})$$

Global type $p \to q : t.G$ specifies the communication of a value of type t_i through the channel from role p to role q, followed by G. Global type $G_1 \,|||\, G_2$ specifies the free interleaving of G_1 and G_2. Global type **end** specifies termination.

The following local types specify the master ($n = 2$) and a worker (any n):

$$L_{\mathtt{m}} = \mathtt{m}\mathtt{w}_1 ! \mathtt{Work} \,.\, \mathtt{m}\mathtt{w}_2 ! \mathtt{Work} \,.\, (\mathtt{w}_1\mathtt{m} ? \mathtt{Done} \,.\, \mathbf{end} \;|||\; \mathtt{w}_2\mathtt{m} ? \mathtt{Done} \,.\, \mathbf{end})$$
$$L_{\mathtt{w}_i} = \mathtt{m}\mathtt{w}_i ? \mathtt{Work} \,.\, \mathtt{w}_i\mathtt{m} ! \mathtt{Done} \,.\, \mathbf{end}$$

Local types $pq!t.L$ and $pq?t.L$ specify the send and receive of a value of type t through the channel from role p to role q, followed by L. Local types $L_1 \,|||\, L_2$ and **end** are similar to the corresponding forms of global types. In general, a local type L_r for role r is mechanically constructed by projecting a global type G onto r, using a recursive function that traverses G's structure. Roughly: every communication in which r participates as sender is preserved as a send in L_r; every communication in which r participates as receiver is preserved as a receive in L_r; every communication in which r does not participate is omitted from L_r.

The following processes implement the master ($n = 2$) and a worker (any n):

$$P_{\mathtt{m}} = \mathtt{m}\mathtt{w}_1 ! \mathtt{work("grep -o -i foo file.txt | wc -l")} \,.\, \mathtt{m}\mathtt{w}_2 ! \mathtt{work("grep -o -i bar file.txt | wc -l")} \,.\, (\mathtt{w}_1\mathtt{m} ? \_ : \mathtt{Done} \,.\, \mathbf{end} \;|||\; \mathtt{w}_2\mathtt{m} ? \_ : \mathtt{Done} \,.\, \mathbf{end})$$
$$P_{\mathtt{w}_i} = \mathtt{m}\mathtt{w}_i ? \mathtt{x} : \mathtt{Work} \,.\, \mathtt{w}_i\mathtt{m} ! \mathtt{do(x)} \,.\, \mathbf{end}$$

Processes $pq!e.P$ and $pq?x{:}t.P$ implement the send and receive of the value of expression e (evaluated at the sender) through the channel from role p to role q into variable x of type t (stored at the receiver), followed by P. Processes $P_1 \,|||\, P_2$ and **end** are similar to the corresponding forms of global/local types.

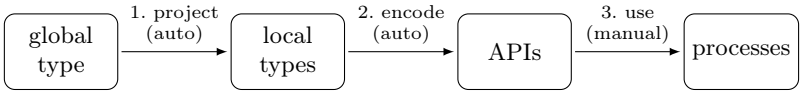


Fig. 3: Workflow of API generation (arrows 1–2 are performed automatically by the tool; arrow 3 is performed manually by the programmer)

$L_{\mathtt{m}}$ and $L_{\mathtt{w}_i}$ are projections of G. Furthermore, assuming that the return types of `work` and `do` are `Work` and `Done`, $P_{\mathtt{m}}$ and $P_{\mathtt{w}_i}$ are well-typed by $L_{\mathtt{m}}$ and $L_{\mathtt{w}_i}$: roughly, the processes and the local types implement (in terms of concrete values of data) and specify (in terms of abstract types of data) the same behaviour. □

In practice, the premier approach to apply the MPST method in combination with mainstream programming languages has been based on *API generation*. The idea, conceived by Hu and Yoshida [19], is based on the insight that local types 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). The corresponding workflow is visualised in Figure 3. API generation has been influential: it is used in the majority of tools that support the MPST method, including Scribble [19,20], its many dialects/extensions [7,23,25,28,30,32,39], νScr [38], `mpstpp` [22], and Pompset [8]. The curious reader can glimpse at Figure 12 for an excerpt of the generated Scala API (following [8,19]) for the master–workers protocol from Example 1.

Open Problem

So far, MPST tools based on API generation have been developed for a wide range of languages, imperative and functional alike, including (in alphabetic order): F# [30], F⋆ [39], Go [7], Java [19,20,22], OCaml [38], PureScript [23], Rust [25], Scala [8,32], and TypeScript [28]. However, none of the existing tools is capable of generating APIs in multiple languages; they support only *unilingual programming* (homogeneity), often leveraging special features of the host's type system to encode advanced MPST concepts (e.g., type providers in F# to encode MPST-based refinement types [30]). In contrast, many real-world distributed systems are engineered using *multilingual programming* (heterogeneity), where some processes are implemented in one language, but others in another. This open problem has not received due attention.

Figure 4a visualises a naive, seemingly simple/low-effort, solution: reuse an existing MPST tool that can generate APIs in a single language $\mathcal{L}_1$, in combination with language bindings for languages $\mathcal{L}_2, \ldots, \mathcal{L}_n$ using *wrapper libraries* as foreign function interfaces, to write processes in $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$. Whereas executing processes in this way is well-understood, verifying processes in this way—the whole point of using API generation—is not. The key issue is that, to enjoy deadlock freedom and protocol compliance as usual, the language bindings should guarantee that well-typed usage of the wrappers in $\mathcal{L}_2, \ldots, \mathcal{L}_n$ implies

well-typed usage of the APIs in $\mathcal{L}_1$. Figure 4b visualises an alternative solution that avoids this issue: use a new tool that can generate APIs in multiple languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$.

Contribution

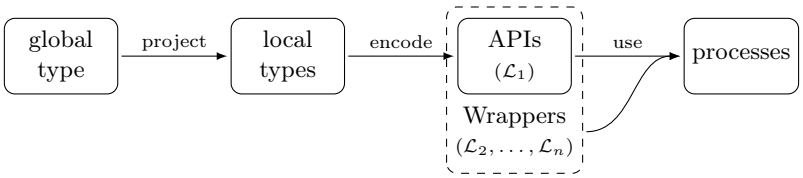
In this paper, we present the *ST4MP* project. ST4MP is an acronym for "Session Types Fo(u)r Multilingual Programming" and pronounced as "stamp". The aim of ST4MP is to develop a tool to apply the MPST method in multilingual programming, based on API generation, according to the workflow in Figure 4b.

Existing work focusses on the discovery/invention of new techniques for a single language (e.g., by leveraging special features of the language's type system in generated APIs). In contrast, ST4MP focusses on the combination/integration of existing techniques for multiple languages. That is, in ST4MP, we prioritise "the (re)engineering of existing techniques" over "the science of new ones".

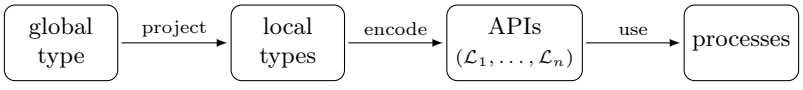
In Section 2, we briefly summarise a basic version of MPST theory; it serves as the foundation of ST4MP. In Section 3, we present a blueprint of the ST4MP tool, including an overview of the ST4MP language. In Section 4, we discuss related work. We emphasise that this paper focusses on the design of ST4MP; the implementation is work-in-progress, part of ongoing efforts, and presented in more detail in a future paper. Moreover, in alignment with the "engineering first, science second"-attitude behind ST4MP, we note that we do not present new techniques in this paper: the blueprint of ST4MP is based on the combination/integration of existing techniques. As a result, this paper can also be read as an introductory article on API generation.

2 MPST Theory in a Nutshell

In this section, we summarise a basic version of MPST theory, based on the more advanced version by Deniélou and Yoshida [10]; given the aim of this



(a) Foreign APIs in languages $\mathcal{L}_2, \ldots, \mathcal{L}_n$



(b) Native APIs in languages $\mathcal{L}_1, \ldots, \mathcal{L}_n$

Fig. 4: Workflows of API generation for multilingual programming

paper, we omit orthogonal and/or more complicated features from this section (e.g., dynamic channel creation, dynamic process creation, and delegation). Our presentation in this section follows the top-down visualisation in Figure 1.

2.1 Global Types

Let $\mathbb{R} = \{\mathtt{alice}, \mathtt{bob}, \mathtt{carol}, \mathtt{m}, \mathtt{w}, \ldots\}$ denote the set of all *roles*, ranged over by p, q, r. Let $\mathbb{T} = \{\mathtt{Unit}, \mathtt{Bool}, \mathtt{Nat}, \mathtt{Work}, \mathtt{Done}, \ldots\}$ 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:

$$G ::= p \twoheadrightarrow q{:}\{t_i.G_i\}_{1\leq i\leq n} \mid G_1 \oplus G_2 \mid \mu X.G \mid X \mid \mathbf{end} \qquad \oplus ::= \;|||\; \mid \cdot$$

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

- 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 channel from role p to role q, 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). We omit the curly brackets when $n = 1$.
- Global type $G_1 \,|||\, G_2$ 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.
- Global types $\mu X.G$ and X specify a **recursive protocol**.
- Global type **end** specifies the **empty protocol**.

Example 2. The following global type specifies the master–workers protocol (Example 1) with two extensions (worker i can tell worker $i+1$ to perform some or none of the work on its behalf; the protocol is repeated):

$$G = \mu X\,.\,((\mathtt{m} \twoheadrightarrow [\mathtt{w}_1, \mathtt{w}_2]{:}\mathtt{Work}\,.\,\mathtt{w}_1 \twoheadrightarrow \mathtt{w}_2{:}\{\mathtt{Work}\,.\,\mathtt{w}_2 \twoheadrightarrow \mathtt{w}_1{:}\mathtt{Done}\,.\,G'\,,\,\mathtt{None}\,.\,G'\}) \cdot X)$$
$$G' = \mathtt{w}_1 \twoheadrightarrow \mathtt{m}{:}\mathtt{Done}\,.\,\mathbf{end} \;|||\; \mathtt{w}_2 \twoheadrightarrow \mathtt{m}{:}\mathtt{Done}\,.\,\mathbf{end}$$

We write "$\mathtt{m} \twoheadrightarrow [\mathtt{w}_1, \mathtt{w}_2]{:}\mathtt{Work}$" as a shorthand for "$\mathtt{m} \twoheadrightarrow \mathtt{w}_1{:}\mathtt{Work}\,.\,\mathtt{m} \twoheadrightarrow \mathtt{w}_2{:}\mathtt{Work}$". □

2.2 Local Types and Projection

Let $\dagger \in \{!, ?\}$. Let $\mathbb{L}$ denote the set of all *local types*, ranged over by L:

$$L ::= \underbrace{pq!\{t_i.L_i\}_{1\leq i\leq n}}_{\text{send}} \mid \underbrace{pq?\{t_i.L_i\}_{1\leq i\leq n}}_{\text{receive}} \mid L_1 \oplus L_2 \mid \mu X.L \mid X \mid \mathbf{end}$$

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

$$(G_1 \oplus G_2) \upharpoonright r = \begin{cases} \textbf{end} & \textbf{if: } G_1 \upharpoonright r = \textbf{end} = G_2 \upharpoonright r \\ G_2 \upharpoonright r & \textbf{if: } G_1 \upharpoonright r = \textbf{end} \neq G_2 \upharpoonright r \\ G_1 \upharpoonright r & \textbf{if: } G_1 \upharpoonright r \neq \textbf{end} = G_2 \upharpoonright r \\ (G_1 \upharpoonright r) \oplus (G_2 \upharpoonright r) & \textbf{if: } G_1 \upharpoonright r \neq \textbf{end} \neq G_2 \upharpoonright r \end{cases}$$

$$\mu X.G \upharpoonright r = \mu X.(G \upharpoonright r) \qquad X \upharpoonright r = X \qquad \textbf{end} \upharpoonright r = \textbf{end}$$

Fig. 5: Projection of global types

These forms of local types are similar to the corresponding forms of global types.

Let $G \upharpoonright r$ denote the *projection* of G onto r. Formally, $\upharpoonright$ is the smallest function induced by the equations in Figure 5. The projection of $p \twoheadrightarrow q{:}\{t_i.G_i\}_{1\leq i\leq n}$ 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" [33].

Example 3. The following local types specify the master and the workers in the extended master–workers protocol (Example 2):

$$L'_{\mathtt{w}_1} = \mathtt{mw}_1?\mathtt{Work}\,.\,\mathtt{w}_1\mathtt{w}_2!\{\mathtt{Work}\,.\,\mathtt{w}_2\mathtt{w}_1?\mathtt{Done}\,.\,\mathtt{w}_1\mathtt{m}!\mathtt{Done}\,.\,\textbf{end}\,,\,\mathtt{None}\,.\,\mathtt{w}_1\mathtt{m}!\mathtt{Done}\,.\,\textbf{end}\}$$

$$L'_{\mathtt{w}_2} = \mathtt{mw}_2?\mathtt{Work}\,.\,\mathtt{w}_1\mathtt{w}_2?\{\mathtt{Work}\,.\,\mathtt{w}_2\mathtt{w}_1!\mathtt{Done}\,.\,\mathtt{w}_2\mathtt{m}!\mathtt{Done}\,.\,\textbf{end}\,,\,\mathtt{None}\,.\,\mathtt{w}_2\mathtt{m}!\mathtt{Done}\,.\,\textbf{end}\}$$

$$L_{\mathtt{m}} = \mu X\,.\,(L^{\text{Ex 1}}_{\mathtt{m}} \cdot X) \qquad L_{\mathtt{w}_1} = \mu X\,.\,(L'_{\mathtt{w}_1} \cdot X) \qquad L_{\mathtt{w}_2} = \mu X\,.\,(L'_{\mathtt{w}_2} \cdot X) \qquad \square$$

2.3 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 $\mathbb{E} = \mathbb{V} \cup \mathbb{X} \cup \cdots$ denote the set of all *expressions*, ranged over by e. Let $\mathbb{P}$ denote the set of all *processes*, ranged over by P:

$$P ::= \textbf{if } e\ P_1\ P_2 \mid pq!e.P \mid pq?\{x_i{:}t_i.P_i\}_{1\leq i\leq n} \mid P_1 \oplus P_2 \mid \mu X.P \mid X \mid \textbf{end}$$

Informally, these forms of processes have the following meaning:

- Process **if** $e\ P_1\ P_2$ implements the **conditional choice** between P_1 and P_2.
- Process $pq!e.P$ implements the **send** of the value of expression e through the channel from role p to role q, followed by P. Asynchronous sends can be combined with conditional choices to implement *internal choices*.

$$\frac{\Gamma \vdash e : \texttt{Bool} \qquad \Gamma \vdash P_1 : L \qquad \Gamma \vdash P_2 : L}{\Gamma \vdash \textbf{if}\ e\ P_1\ P_2 : L}\ [\textsc{If}] \qquad \frac{\Gamma \vdash P : L}{\Gamma \vdash \mu X.P : \mu X.L}\ [\textsc{Mu}]$$

$$\frac{\Gamma \vdash e : t_i \qquad \Gamma \vdash P : L_i \qquad 1 \leq i \leq n}{\Gamma \vdash pq!e.P : pq!\{t_i.L_i\}_{1\leq i\leq n}}\ [\textsc{Send}] \qquad \frac{}{\Gamma \vdash X : X}\ [\textsc{Var}]$$

$$\frac{\Gamma, x_i : t_i \vdash P_i : L_i \text{ for every } 1 \leq i \leq n}{\Gamma \vdash pq?\{x_i{:}t_i.P_i\}_{1\leq i\leq n} : pq?\{t_i.L_i\}_{1\leq i\leq n}}\ [\textsc{Recv}] \qquad \frac{}{\Gamma \vdash \textbf{end} : \textbf{end}}\ [\textsc{End}]$$

$$\frac{\Gamma \vdash P_1 : L_1 \qquad \Gamma \vdash P_2 : L_2}{\Gamma \vdash P_1 \,|||\, P_2 : L_1 \,|||\, L_2}\ [\textsc{Par}] \qquad \frac{\Gamma \vdash P_1 : L_1 \qquad \Gamma \vdash P_2 : L_2}{\Gamma \vdash P_1 \cdot P_2 : L_1 \cdot L_2}\ [\textsc{Seq}]$$

Fig. 6: Well-typedness of processes

- Process $pq?\{x_i{:}t_i.P_i\}_{1\leq i\leq n}$ implements the **receive** of a value into variable x_i of type t_i through the 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*. 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 $P_1 \,|||\, P_2$ implements the **parallel composition** of P_1 and P_2. We note that $P_1 \,|||\, P_2$ 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.
- Processes $\mu X.G$ and X implement a **recursive role**.
- Process **end** implements the **empty role**.

Let $\Gamma \vdash e : t$ denote *well-typedness* of expression e by data type t in environment Γ. Let $\Gamma \vdash P : L$ denote *well-typedness* of process P by local type L in environment Γ. Formally, $\vdash$ is the smallest relation induced by the rules in Figure 6. 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, but produce only one specified output.

Theorem 1 (Deniélou and 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.*

Example 4. The following processes implement the master and the workers in the extended master–workers protocol (Example 2):

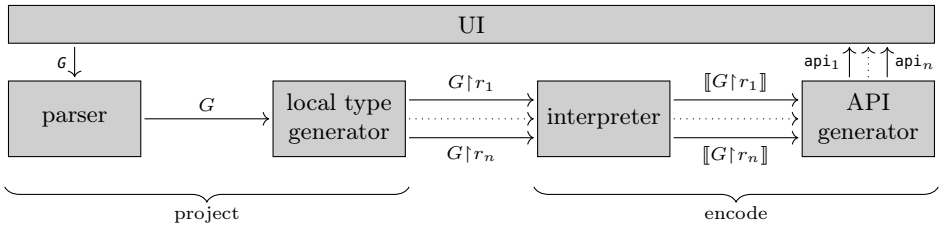


Fig. 7: Architecture of the ST4MP tool

$$P'_{\mathtt{w}_1} = \mathtt{mw}_1?\mathtt{x{:}Work}\,.\,\textbf{if}\ \mathtt{delegate\_work(x)}\ \begin{cases}\mathtt{w}_1\mathtt{w}_2!\mathtt{x}\,.\,\mathtt{w}_2\mathtt{w}_1?\mathtt{y{:}Done}\,.\,\mathtt{w}_1\mathtt{m}!\mathtt{y}\,.\,\textbf{end}\\ \mathtt{w}_1\mathtt{w}_2!\mathtt{none()}\,.\,\mathtt{w}_1\mathtt{m}!\mathtt{do(x)}\,.\,\textbf{end}\end{cases}$$

$$P'_{\mathtt{w}_2} = \mathtt{mw}_1?\mathtt{x{:}Work}\,.\,\mathtt{w}_1\mathtt{w}_2?\begin{cases}\mathtt{y{:}Work}\,.\,\mathtt{w}_2\mathtt{w}_1!\mathtt{do(y)}\,.\,\mathtt{w}_2\mathtt{m}!\mathtt{do(x)}\,.\,\textbf{end}\\ \mathtt{\_{:}None}\,.\,\mathtt{w}_2\mathtt{m}!\mathtt{do(x)}\,.\,\textbf{end}\end{cases}$$

$$P_{\mathtt{m}} = \mu X\,.\,(P_{\mathtt{m}}^{\mathrm{Ex}\,1}\cdot X) \qquad P_{\mathtt{w}_1} = \mu X\,.\,(P'_{\mathtt{w}_1}\cdot X) \qquad P_{\mathtt{w}_2} = \mu X\,.\,(P'_{\mathtt{w}_2}\cdot X) \quad \square$$

3 Blueprint of ST4MP

In this section, we present the blueprint of the *ST4MP tool* (henceforth simply called "ST4MP"). ST4MP is a tool to apply the MPST method in multilingual programming, based on API generation, according to the workflow in Figure 4b. This section focusses on the design; the implementation is work-in-progress, part of ongoing efforts, and presented in more detail in a future paper.

Figure 7 visualises the architecture of ST4MP: boxes represent components; arrows represent data flow between them. The parser and the local type generator correspond to arrow "project" in Figure 4b, while the interpreter and the API generator correspond with arrow "encode". Next, we discuss the purpose of every component in more detail, along with the main design decisions.

3.1 UI

As ST4MP aims to support multilingual programming, the purpose of the UI is to offer a language-independent and cross-platform user interface.

The main design decision related to the UI has been to make it accessible and usable through any contemporary browser, instead of through a separate plug-in for a particular IDE/editor (to avoid the situation in which programmers are forced to install software only to be able to use ST4MP). As a result, while ST4MP is implemented in Scala, it is compiled to HTML-JavaScript-CSS. A live snapshot is available at `https://arca.di.uminho.pt/st4mp/`, which is based on our previous Pompset tool [8].

3.2 Parser

The purpose of the parser is to consume a global type in concrete syntax `G` as input, written in the *ST4MP language* (henceforth simply called "ST4MP"), and produce a global type in abstract syntax G as output (Section 2.1). The parser also checks additional well-formedness requirements.

The main design decisions related to the parser have been:

- Use a language-independent notation for data types based on *JSON* ("JavaScript Object Notation") [6]. JSON is a domain-specific language for typed values; it has been widely adopted as a "text syntax that facilitates structured data interchange between all programming languages" [6]. We use JSON also to auto-serialise (i.e., convert binary data in one language into a common textual exchange format) and auto-deserialise (i.e., convert back to binary data, possibly in another language) inside generated APIs, as clarified shortly.
- Use a language-independent notation for global types based on *Featherweight Scribble* [31]. Featherweight Scribble is a domain-specific language for global types; it was chosen because it is a core fragment of Scribble [19,20] (and its extensions [7,23,25,28,30,32,39]), which has been the premier language to apply the MPST method in combination with mainstream programming languages.

Let ℓ range over *identifiers* in ST4MP (i.e., strings consisting of alphanumericals). Let `T` and `G` range over *data types* and *global types* in ST4MP:

$$\texttt{T} ::= \ell\texttt{(}\ell_1\texttt{:T}_1\texttt{,}\ldots\texttt{,}\ell_n\texttt{:T}_n\texttt{)} \mid \texttt{[T]} \mid \texttt{Number} \mid \texttt{String} \mid \texttt{Boolean}$$

$$\texttt{G} ::= \texttt{T from } p \texttt{ to } q\texttt{;} \mid \texttt{choice at } p \texttt{ \{ G}_1 \texttt{ \} or} \cdots \texttt{or \{ G}_n \texttt{ \}} \mid$$
$$\texttt{par \{ G}_1 \texttt{ \} and} \cdots \texttt{and \{ G}_n \texttt{ \}} \mid \texttt{G}_1\ \texttt{G}_2 \mid \texttt{rec } \ell \texttt{ \{ G \}} \mid \texttt{continue } \ell\texttt{;}$$

Informally, these forms have the following meaning:

- Data type $\ell\texttt{(}\ell_1\texttt{:T}_1\texttt{,}\ldots\texttt{,}\ell_n\texttt{:T}_n\texttt{)}$ specifies an *object*, where ℓ identifies the class of the object, while $\ell_1\texttt{:T}_1\texttt{,}\ldots\texttt{,}\ell_n\texttt{:T}_n$ identify n typed attributes of the object; parentheses can be omitted when $n=0$. Data type `[T]` specifies an *array*. The remaining data types are self-explanatory.
- Global types in ST4MP follow closely our formalisation of global types in Section 2.1; e.g., `T from` p `to` q`;` G denotes $p \twoheadrightarrow q\texttt{:}\{\texttt{T}.G\}$. As additional well-formedness conditions, we require: **(1)** every branch of `choice at` p start with a communication in which p is the sender and the receiver is the same; **(2)** if both $\ell\texttt{(}\ell_1\texttt{:T}_1\texttt{,}\ldots\texttt{,}\ell_n\texttt{:T}_n\texttt{)}$ and $\ell\texttt{(}\ell_{n+1}\texttt{:T}_{n+1}\texttt{,}\ldots\texttt{,}\ell_{n+m}\texttt{:T}_{n+m}\texttt{)}$ are in a global type, then $\ell_1\texttt{:T}_1\texttt{,}\ldots\texttt{,}\ell_n\texttt{:T}_n = \ell_{n+1}\texttt{:T}_{n+1}\texttt{,}\ldots\texttt{,}\ell_{n+m}\texttt{:T}_{n+m}$ (i.e., if ℓ identifies a class, then it must do so unambiguously).

Example 5. The ST4MP code in Figure 8 specifies the extended master–workers protocol (Example 2). □

```
rec Loop {
  Work(cmd:String) from Master to Worker1;
  Work(cmd:String) from Master to Worker2;
  choice at Worker1 {
    Work(cmd:String) from Worker1 to Worker2;
    Done(res:Number) from Worker2 to Worker1;
  } or {
    None() from Worker1 to Worker2;
  }
  par {
    Done(res:Number) from Worker1 to Master;
  } and {
    Done(res:Number) from Worker2 to Master;
  }
  continue Loop; }
```

Fig. 8: Extended master–workers protocol in ST4MP (Example 5)

3.3 Local type generator

The purpose of the local type generator is to consume a global type G as input and produce local types $G \upharpoonright r_1, \ldots, G \upharpoonright r_n$ as output, by projecting G onto every role $r_1, \ldots, r_n$ that occurs in it (Section 2.2).

3.4 Interpreter

The purpose of the interpreter is to consume local types $G \upharpoonright r_1, \ldots, G \upharpoonright r_n$ as input and produce transition-based models of their operational behaviour $[\![G \upharpoonright r_1]\!], \ldots, [\![G \upharpoonright r_n]\!]$ as output; APIs can subsequently be generated for such models.

The main design decision related to the interpreter has been to support two kinds of transition-based models—*automata* and *pomsets*—based on existing interpretation functions on local types [11, 13]. To illustrate the idea, the automaton interpretation of local types is summarised in Figure 9, including an example; the pomset interpretation is more complicated and explained in detail elsewhere [8]. The reason to support two different models is that they have different advantages and, as such, can serve different purposes:

- The advantage of the automaton interpretation of local types is that it is "total": it is defined for all non-recursive local types and for all tail-recursive local types. Another advantage is that the requirements to generate APIs for automata are relatively low (i.e., no advanced type system features needed and can be implemented, e.g., in Java [19]). The disadvantage is that the automaton interpretation suffers from state explosion in the presence of parallel composition.
- The advantage of the pomset interpretation of local types is that it does not suffer from state explosion (i.e., pomsets offer a more concise representation of concurrency than automata). The disadvantage is that the pomset

Let $\boldsymbol{\Sigma} = \{pq!t \mid p \neq q\} \cup \{pq?t \mid p \neq q\}$ denote the set of all *type-level actions* ("the alphabet"), ranged over by σ. Let $\mathbb{A}$ denote the set of all *automata* over $\boldsymbol{\Sigma}$, ranged over by A. Formally, an automaton 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 $\delta : S \times \boldsymbol{\Sigma} \rightharpoonup S$ denotes a *transition function*.

Let $[\![L]\!]_\mathrm{aut}$ denote the *interpretation* of local type L as an automaton. Formally, $[\![\text{-}]\!]_\mathrm{aut}$ is the smallest function induced by the following equations:

$$[\![\mathbf{end}]\!]_\mathrm{aut} = \longrightarrow\textcircled{\scriptsize}$$

$$[\![pq\dagger\{t_i.L_i\}_{1\leq i\leq n}]\!]_\mathrm{aut} = \longrightarrow\bigcirc \xrightarrow{pq\dagger t_1} A_1 \xrightarrow{\sigma_{11}} \textcircled{\scriptsize}, \quad \bigcirc \xrightarrow{pq\dagger t_n} A_n \xrightarrow{\sigma_{nk_n}} \textcircled{\scriptsize}, \quad A_1 \xrightarrow{\sigma_{n1}} \textcircled{\scriptsize}, \quad A_n \xrightarrow{\sigma_{1k_1}} \textcircled{\scriptsize}$$

$$\text{s.t., for every } 1 \leq i \leq n,\ [\![L_i]\!]_\mathrm{aut} = \longrightarrow A_i \xrightarrow{\sigma_{ik_i}} \textcircled{\scriptsize}, \quad A_i \xrightarrow{\sigma_{i1}} \textcircled{\scriptsize}$$

$$[\![L_1 \mid\mid\mid L_2]\!]_\mathrm{aut} = (S_1 \times S_2, (s_{1\mathrm{i}}, s_{2\mathrm{i}}), (s_{1\mathrm{f}}, s_{2\mathrm{f}}), \delta)$$

$$\text{s.t. } \delta((s_1, s_2), \sigma) = \begin{cases} (s_1', s_2) & \text{if } \delta_1(s_1, \sigma) = s_1' \\ (s_1, s_2') & \text{if } \delta_2(s_2, \sigma) = s_2' \end{cases}$$

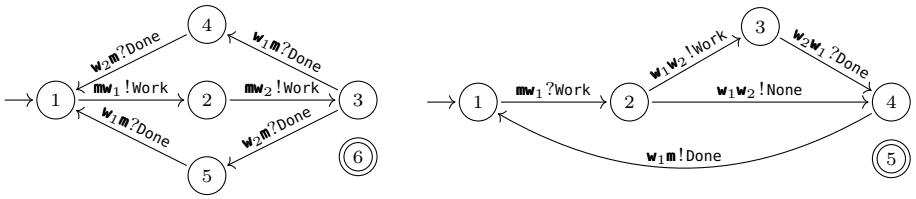
$$[\![L_1 \cdot L_2]\!]_\mathrm{aut} = (S_1 \cup S_2, s_{1\mathrm{i}}, s_{2\mathrm{f}}, \hat{\delta}_1 \cup \delta_2)$$

$$\text{s.t. } \hat{\delta}_1(s_1, \sigma) = \begin{cases} s_1' & \text{if } \delta_1(s_1, \sigma) = s_1' \neq s_{1\mathrm{f}} \\ s_{2\mathrm{i}} & \text{if } \delta_1(s_1, \sigma) = s_1' = s_{1\mathrm{f}} \end{cases}$$

$$[\![\mu X.(L_1 \cdot X)]\!]_\mathrm{aut} = (S_1, s_{1\mathrm{i}}, s_{1\mathrm{f}}, \hat{\delta}_1) \quad \text{s.t. } \hat{\delta}_1(s_1, \sigma) = \begin{cases} s_1' & \text{if } \delta_1(s_1, \sigma) = s_1' \neq s_{1\mathrm{f}} \\ s_{1\mathrm{i}} & \text{if } \delta_1(s_1, \sigma) = s_1' = s_{1\mathrm{f}} \end{cases}$$

The interpretation of **end** is the automaton that accepts the empty language. The interpretation of $pq\dagger\{t_i.L_i\}_{1\leq i\leq n}$ is the automaton 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 $L_1 \mid\mid\mid L_2$ and $L_1 \cdot L_2$ are the automata that accept the shuffle and the concatenation of the languages accepted by the interpretations of L_1 and L_2.

Example 6. The following automata are the interpretations of $L_\mathtt{m}$ (left) and $L_{\mathtt{w}_1}$ (right) in the extended master-workers protocol (Example 3):



We note that state 6 (left) and state 5 (right) are unreachable. □

Fig. 9: Interpreting local types as automata [11]

interpretation is "partial": it is defined only for non-recursive local types and top-level tail-recursive local types (i.e., of the form $\mu X.(L \cdot X)$, where L is non-recursive) that are, moreover, choice-free. Another disadvantage is that the requirements to generate APIs for pomsets are relatively high (i.e., advanced type system features are needed; e.g., match types in Scala [1,8]).

Example 7. $L_{\mathtt{m}}$ in Example 3 is both top-level tail-recursive and choice-free, so it can be interpreted not only as an automaton (Example 6 in Figure 9) but also as a pomset (Cledou et al. [8]). In this way, state explosion can be avoided (i.e., the automaton has $O(2^n)$ states in general, where n is the number of workers) to reduce both the time to generate an API and the space to store it. □

3.5 API generator

The purpose of the API generator is to consume operational models $[\![G \upharpoonright r_1]\!], \ldots, [\![G \upharpoonright r_n]\!]$ as input and produce APIs $\mathtt{api}_1, \ldots, \mathtt{api}_n$ as output.

The underlying principle is that a process P_i is well-typed by local type $G \upharpoonright r_i$ (Section 2) if, and only if, every possible sequence of sends and receives by P_i can be simulated by a sequence of transitions of $[\![G \upharpoonright r_i]\!]$. The "trick" is to structure the API in such a way that when the compiler successfully type-checks the API's usage, it has effectively computed a sequence of transitions of $[\![G \upharpoonright r_i]\!]$ for every possible sequence of sends and receives by P_i. The main technique to achieve this is to represent every state of $[\![G \upharpoonright r_i]\!]$ as an object (broadly construed), and every transition as a method (broadly construed), such that every call of method `t` on object `s` is well-typed if, and only if, the transition represented by `t` is allowed in the state represented by `s`.

From the programmer's perspective, to use the API, a function `f` needs to be defined that consumes an "initial state object" `s0` as input and produces a "final state object" as output. Inside of `f`, initially, the only actions that can be performed, are those for which a well-typed method call on `s0` exists. When such a method is called, an action is performed and a fresh "successor state object" `s1` is returned. Subsequently, the only actions that can be performed, are those for which a well-typed method call on `s1` exists. When such a method is called, another action is performed, and another fresh "successor successor state object" `s2` is returned. This goes on until the final state object is returned (if any).

When a method call is *not* well-typed, it means that: **(1)** the transition is not allowed in the state; **(2)** hence, the local type does not specify the action; **(3)** hence, the action is not allowed in the protocol. 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 in many languages [7,19,22,28,30,32,38]: checking the first assumption requires a form of substructural types, while checking the second assumption is generally

	JSON	Scala
object	ℓ(`$\ell_1$: $T_1$, ..., $\ell_n$: $T_n$`)	`class` ℓ(`$\ell_1$: $T_1$, ..., $\ell_n$: $T_n$`) (generated)
array	`[T]`	`Array[T]`
number	`Number`	`Int	Double` (union type)
string	`String`	`String`
boolean	`Boolean`	`Boolean`

Fig. 10: Mapping between types in JSON and Scala

undecidable. However, the first assumption can be dynamically monitored using lightweight checks at execution-time.[4]

The main design decisions related to the API generator have been to:

- Support three languages initially: Java, Scala, and Rust. We selected Java because it is one of the most-used languages today.[5] We selected Scala and Rust because their type systems have special features (match types in Scala and ownership types in Rust) that can be leveraged in generated APIs.
  We note that the combination of Java and Scala also presents original research opportunities: as both languages are executed by the JVM, it is interesting to investigate if special API generation techniques can be developed to leverage the common runtime environment (e.g., to reduce communication latency relative to TCP connections).

- Use TCP and JSON as language-independent mechanisms to transport and represent data. That is, ST4MP will include code in every API to create/configure the underlying TCP connections, plus code to (de)serialise data to JSON values. To illustrate the idea, for Scala, the JSON mapping is summarised in Figure 10.

- Use existing techniques to generate APIs for transition-based models of local types, namely those conceived by Hu and Yoshida [19] for automata (in Java, Scala, and Rust) and Cledou et al. [8] for pomsets (in Scala). To illustrate the idea, for automata in Scala, the former technique is summarised in Figure 11, including an example illustrated in Figure 12; the latter technique is more complicated and explained elsewhere [8].

Example 9. In the extended master-workers protocol, the master can be implemented using the Scala API based on pomsets (to avoid state explosion); the workers can be implemented in Java, Scala, and/or Rust based on automata. □

[4] Thus, typically, the application of the MPST method using API generation is not absolutely zero-cost in terms of overhead: while the vast majority of the work is done statically, a bit of work is done dynamically. Previous experiments indicate that this overhead is negligible in practice, though (e.g., [7]).

[5] `https://www.tiobe.com/tiobe-index/`

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

- Every state $s \in S$ is represented as class $\langle r \rangle\$\langle s \rangle$ 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 represented as a method of class $\langle r \rangle\$\langle s \rangle$ to perform action σ and return an instance of class $\langle r \rangle\$\langle \delta(s, \sigma) \rangle$.

If s has only !-transitions of the form $\delta(s, rq_1!t_1), \ldots, \delta(s, rq_n!t_n)$, then:

```
class ⟨r⟩$⟨s⟩(net: Network):
  def send(q: ⟨q1⟩, e: ⟨t1⟩): ⟨r⟩$⟨δ(s, pq1!t1)⟩ = ...
  ...
  def send(q: ⟨qn⟩, e: ⟨tn⟩): ⟨r⟩$⟨δ(s, pqn!tn)⟩ = ...
```

Parameter `net` of class $\langle r \rangle\$\langle s \rangle$ encapsulates the underlying communication infrastructure; 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 successor state object. These methods mimic $pq!e.P$ (Section 2.3).
If s has only ?-transitions of the form $\delta(s, p_1r?t_1), \ldots, \delta(s, p_nr?t_n)$, then:

```
class ⟨r⟩$⟨s⟩(net: Network):
  def recv(f1: (⟨p1⟩, ⟨t1⟩, ⟨r⟩$⟨δ(s, p1q?t1)⟩)) => ⟨r⟩$⟨sf⟩,
           ...,
           fn: (⟨pn⟩, ⟨tn⟩, ⟨r⟩$⟨δ(s, pnq?tn)⟩)) => ⟨r⟩$⟨sf⟩) = ...
```

Parameter `fi` of method `recv` is the i-th *continuation*; it is called with the identifier of the sender, the value to receive, and a successor state object after the "real receive". This method mimics $pq?\{x_i{:}t_i.P_i\}_{1 \leq i \leq n}$ (Section 2.3).
If s has both !-transitions and ?-transitions, an error is reported.

Example 8. The APIs in Figure 12 are generated for $[\![L_\mathtt{m}]\!]_\mathrm{aut}$ and $[\![L_{\mathtt{w}_1}]\!]_\mathrm{aut}$ in the extended master–workers protocol (Example 6 in Figure 9). We note that classes `M$6` and `W1$5` represent unreachable final states.
Furthermore, functions `m` and `w1` in Figure 12, which use the generated APIs, are Scala versions of $P_\mathtt{m}$ and $P_{\mathtt{w}_1}$ (Example 4). 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 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. □

Fig. 11: Generating Scala APIs for automata

4 Related Work

The idea to interpret local types as automata 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 compatibility*: it is used to provide a sound and complete characterisation between global types and *systems* (i.e., parallel compositions of automata that communicate through asynchronous channels). Multiparty compatibility was further studied and generalised in sub-

```
// generated API (excerpt)
class M$1(net: Network):
  def send(q: W1, e: Work): M$2 = ...
class M$2(net: Network):
  def send(q: W2, e: Work): M$3 = ...
class M$3(net: Network):
  def recv(f1: (W1, Done, M$4) => M$6,
           f2: (W2, Done, M$5) => M$6): M$6 = ...
class M$4(net: Network):
  def recv(f2: (W2, Done, M$1) => M$6): M$6 = ...
class M$5(net: Network):
  def recv(f1: (W1, Done, M$1) => M$6): M$6 = ...
class M$6(net: Network)
type M$Initial = M$1
type M$Final   = M$6

// process
def m(s: M$Initial): M$Final = s.
  send(W1, new Work("grep -o -i foo file.txt | wc -l")).
  send(W2, new Work("grep -o -i bar file.txt | wc -l"))).
  recv((_, x, s) => s.recv(
         (_, y, s) => { println(x.res + y.res); m(s) }),
       (_, x, s) => s.recv(
         (_, y, s) => { println(x.res + y.res); m(s) }))
```

(a) Master

```
// generated API (excerpt)
class W1$1(net: Network):
  def recv(f: (M, Work, W1$2) => W1$5): W1$5 = ...
class W1$2(net: Network):
  def send(q: W2, e: Work): W1$3 = ...
  def send(q: W2, e: None): W1$4 = ...
class W1$3(net: Network):
  def recv(f: (W2, Done, W1$4) => W1$5): W1$5 = ...
class W1$4(net: Network):
  def send(q: M, e: Done): W1$1 = ...
class W1$5(net: Network):
type W1$Initial = W1$1
type W1$Final   = W1$5

// process
def w1(s: W1$Initial): W1$Final = s.
  recv((_, x, s) => if delegateWork(x) then
    s.send(W2, x).recv((_, y, s) => w1(s.send(M, y))) else
    w1(s.send(W2, new None()).send(M, doWork(x)))
```

(b) Worker 1

Fig. 12: Generated Scala APIs and processes for the extended master–workers protocol (Example 8 in Figure 11)

sequent work, to cover timed behaviour [4], more flexible choice [26], and non-synchronisability [27].

The idea to represent automata as APIs was conceived by Hu and Yoshida [19, 20], for Java. The approach has subsequently been used in combination with numerous other programming languages as well, including F# [30], F⋆ [39], Go [7], OCaml [38], PureScript [23], Rust [25], Scala [32], and TypeScript [28]. In many of these works, special features 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⋆ to generate APIs that support MPST-based refinement [30,39], while King et al. and Lagaillardie et al. use indexed monads in PureScript and ownership types in Rust to support static linearity [23,25].

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. [21] (for OCaml), the work of Harvey et al., Kouzapas et al., and Voinea et al. [16,24,37] (for Java, using a typestate extension), and the work of Scalas et al. [34,35] (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 [2,3,9,15,29,30]. 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.

The idea to represent local types as pomsets was conceived by Guanciale and Tuosto [13], in a continuation of earlier work on pomset-based semantics of global types [36]. A key contribution of Guanciale and Tuosto is a sound procedure to determine if a pomset interpretation of a global type is *realisable* as a collection of pomset interpretations of the global type's projections. The PomCho tool [14] supports analysis (including counterexample generation), visualisation, and projection of pomsets. However, PomCho cannot generate APIs.

5 Conclusion

Multiparty session types (MPST) constitute a method to simplify construction and analysis of distributed systems. In practice, the premier approach to apply the MPST method in combination with mainstream programming languages has been based on API generation. However, existing tools support only unilingual programming (homogeneity), while many real-world distributed systems are engineered using multilingual programming (heterogeneity). In this paper, we presented a blueprint of ST4MP: a tool to apply the MPST method in multilingual programming, based on API generation.

Acknowledgements *Sung-Shik Jongmans*: Netherlands Organisation of Scientific Research (NWO): 016.Veni.192.103.
José Proença: This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the

CISTER Unit (UIDP/UIDB/04234/2020) and the IBEX project (PTDC/CCI-COM/4280/2021); also by national funds through FCT and European funds through EU ECSEL JU, within project VALU3S (ECSEL/0016/2019 - JU grant nr. 876852) – The JU receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Czech Republic, Germany, Ireland, Italy, Portugal, Spain, Sweden, Turkey. Disclaimer: This document reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains.

References

1. Blanvillain, O., Brachthäuser, J.I., Kjaer, M., Odersky, M.: Type-level programming with match types. Proc. ACM Program. Lang. **6**(POPL), 1–24 (2022)
2. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. Theor. Comput. Sci. **669**, 33–58 (2017)
3. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: CONCUR. Lecture Notes in Computer Science, vol. 6269, pp. 162–176. Springer (2010)
4. Bocchi, L., Lange, J., Yoshida, N.: Meeting deadlines together. In: CONCUR. LIPIcs, vol. 42, pp. 283–296. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015)
5. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
6. Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Mar 2014). https://doi.org/10.17487/RFC7159, https://www.rfc-editor.org/info/rfc7159
7. Castro-Perez, D., Hu, R., Jongmans, S., Ng, N., Yoshida, N.: 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. Cledou, G., Edixhoven, L., Jongmans, S.S., Proença, J.: API generation for multiparty session types, revisited and revised using Scala 3. In: ECOOP. LIPIcs, vol. 222, pp. 27:1–27:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
9. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. Formal Methods Syst. Des. **46**(3), 197–225 (2015)
10. Deniélou, P., Yoshida, N.: Dynamic multirole session types. In: POPL. pp. 435–446. ACM (2011)
11. Deniélou, P., Yoshida, N.: Multiparty session types meet communicating automata. In: ESOP. Lecture Notes in Computer Science, vol. 7211, pp. 194–213. Springer (2012)
12. Deniélou, P., Yoshida, N.: Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In: ICALP (2). Lecture Notes in Computer Science, vol. 7966, pp. 174–186. Springer (2013)
13. Guanciale, R., Tuosto, E.: Realisability of pomsets. J. Log. Algebraic Methods Program. **108**, 69–89 (2019)
14. Guanciale, R., Tuosto, E.: Pomcho: A tool chain for choreographic design. Sci. Comput. Program. **202**, 102535 (2021)

15. Hamers, R., Jongmans, S.: Discourje: Runtime verification of communication protocols in clojure. In: TACAS (1). Lecture Notes in Computer Science, vol. 12078, pp. 266–284. Springer (2020)
16. Harvey, P., Fowler, S., Dardha, O., Gay, S.J.: Multiparty session types for safe runtime adaptation in an actor language. In: ECOOP. LIPIcs, vol. 194, pp. 10:1–10:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)
18. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016)
19. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. Lecture Notes in Computer Science, vol. 9633, pp. 401–418. Springer (2016)
20. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. Lecture Notes in Computer Science, vol. 10202, pp. 116–133. Springer (2017)
21. Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: ECOOP. LIPIcs, vol. 166, pp. 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
22. Jongmans, S., Yoshida, N.: Exploring type-level bisimilarity towards more expressive multiparty session types. In: ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 251–279. Springer (2020)
23. King, J., Ng, N., Yoshida, N.: Multiparty session type-safe web development with static linearity. In: PLACES@ETAPS. EPTCS, vol. 291, pp. 35–46 (2019)
24. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. Sci. Comput. Program. **155**, 52–75 (2018)
25. Lagaillardie, N., Neykova, R., Yoshida, N.: Implementing multiparty session types in Rust. In: COORDINATION. Lecture Notes in Computer Science, vol. 12134, pp. 127–136. Springer (2020)
26. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: POPL. pp. 221–232. ACM (2015)
27. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: CAV (1). Lecture Notes in Computer Science, vol. 11561, pp. 97–117. Springer (2019)
28. Miu, A., Ferreira, F., Yoshida, N., Zhou, F.: Communication-safe web programming in typescript with routed multiparty session types. In: CC. pp. 94–106. ACM (2021)
29. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Aspects Comput. **29**(5), 877–910 (2017)
30. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in F#. In: CC. pp. 128–138. ACM (2018)
31. Neykova, R., Yoshida, N.: Featherweight scribble. In: Models, Languages, and Tools for Concurrent and Distributed Programming. Lecture Notes in Computer Science, vol. 11665, pp. 236–259. Springer (2019)
32. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
33. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (2019)
34. Scalas, A., Yoshida, N., Benussi, E.: Effpi: verified message-passing programs in dotty. In: SCALA@ECOOP. pp. 27–31. ACM (2019)

35. Scalas, A., Yoshida, N., Benussi, E.: Verifying message-passing programs with dependent behavioural types. In: PLDI. pp. 502–516. ACM (2019)
36. Tuosto, E., Guanciale, R.: Semantics of global view of choreographies. J. Log. Algebraic Methods Program. **95**, 17–40 (2018)
37. Voinea, A.L., Dardha, O., Gay, S.J.: Typechecking Java protocols with [St]Mungo. In: FORTE. Lecture Notes in Computer Science, vol. 12136, pp. 208–224. Springer (2020)
38. Yoshida, N., Zhou, F., Ferreira, F.: Communicating finite state machines and an extensible toolchain for multiparty session types. In: FCT. Lecture Notes in Computer Science, vol. 12867, pp. 18–35. Springer (2021)
39. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. **4**(OOPSLA), 148:1–148:30 (2020)