Multiparty Session Typing, Embedded

Sung-Shik Jongmans[0000-0002-4394-8745]

University of Groningen, the Netherlands

Abstract. Multiparty session typing (MPST) is a method to make concurrent programming simpler. The idea is to use type checking to automatically detect safety and liveness violations of implementations relative to specifications. In practice, the premier approach to combine MPST with mainstream languages—in the absence of native support—is based on *external DSLs* and associated tooling.
In contrast, we study the question of how to support MPST by using *internal DSLs*. Answering this question positively, this paper presents the `mpst.embedded` library: it leverages Scala's lightweight form of dependent typing, called match types, to embed MPST directly into Scala. Our internal-DSL-based approach avoids programming friction and leaky abstractions of the external-DSL-based approach for MPST.

1 Introduction

Background With the advent of multicore processors, multithreaded programming—a notoriously error-prone enterprise—has become increasingly important.

Because of this, mainstream languages have started to offer core support for higher-level *communication* primitives besides lower-level *synchronisation* primitives (e.g., Clojure, Go, Kotlin, Rust). The idea has been to add *message passing* as an abstraction for *shared memory*, as—supposedly—*channels* are easier to use than *locks*. Yet, empirical research shows that "message passing does not necessarily make multithreaded programs less error-prone than shared memory" [33].

One of the core challenges is as follows: given a specification S of the *communication protocols* that an implementation I should fulfil, how to prove that I is *safe* and *live* relative to S? Safety means "bad" communication actions never happen: <u>if</u> a communication action happens in I, <u>then</u> it is allowed to happen by S. Liveness means "good" communication actions eventually happen.

Multiparty session typing (MPST) MPST [16] is a method to automatically prove safety and liveness of communication protocol implementations relative to specifications. The idea is to write specifications as *behavioural types* [1, 19] against which implementations are type-checked. Formally, the central theorem is that well-typedness at compile-time implies safety and liveness at run-time. Over the past 10–15 years, much progress has been made, including the development of many tools to combine MPST with mainstream languages (e.g., F# [29], F* [36], Go [5], Java [17, 18], OCaml [20], Rust [26, 27], Scala [2, 6, 12, 30], and TypeScript [28]). Fig. 1 visualises the idea behind MPST in more detail:

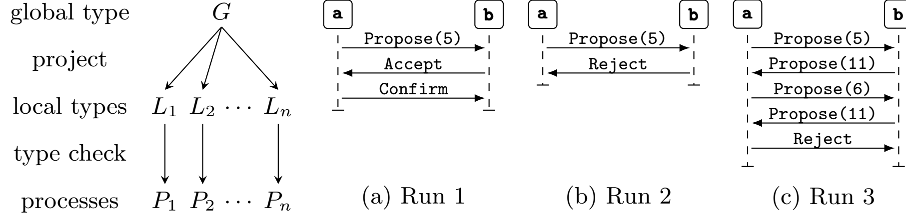

Fig. 1: MPST Fig. 2: A few possible runs of the Negotiation protocol

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.
2. Next, G is decomposed into local types $L_1, \ldots, L_n$ by *projecting* G onto each role. Each local type models the behaviour of one process alone.
3. Last, safety and liveness are verified by *type-checking* each P_i against L_i.

Example 1. The *Negotiation* protocol, originally defined in the MPST literature by Neykova et al. [29], consists of roles *Alice* and *Bob*. Fig. 2 shows three possible runs. First, a proposal is communicated from Alice to Bob. Next, its acceptance, rejection, or a counter-proposal is communicated from Bob to Alice. Next:

- In case of an acceptance, a confirmation is communicated from Alice to Bob.
- In case of a rejection, the protocol ends.
- In case of a counter-proposal, its acceptance, rejection, or another counter-proposal is communicated from Alice to Bob. And so on.

The following recursive **global type** specifies the protocol:

$$G = \mathbf{a}\twoheadrightarrow\mathbf{b}{:}\mathtt{Propose(Int)}.\mu X.\mathbf{b}\twoheadrightarrow\mathbf{a}{:}\begin{cases}\mathtt{Accept}.\mathbf{a}\twoheadrightarrow\mathbf{b}{:}\mathtt{Confirm}.\checkmark\\ \mathtt{Reject}.\checkmark\\ \mathtt{Propose}.\mathbf{a}\twoheadrightarrow\mathbf{b}{:}\begin{cases}\mathtt{Accept}.\mathbf{b}\twoheadrightarrow\mathbf{a}{:}\mathtt{Confirm}.\checkmark\\ \mathtt{Reject}.\checkmark\\ \mathtt{Propose(Int)}.X\end{cases}\end{cases}$$

Global type $p\twoheadrightarrow q{:}\{t_i.G_i\}_{1\leq i\leq n}$ specifies the communication of a value of data type t_i from role p to role q, followed by G_i, for some $1 \leq i \leq n$; we omit braces when $n=1$. Global type $\checkmark$ specifies termination. The following recursive **local type, projected from the global type,** specifies Bob (Alice is similar):

$$L_{\mathbf{b}} = \mathbf{ab}?\mathtt{Propose(Int)}.\mu X.\mathbf{ba}!\begin{cases}\mathtt{Accept}.\mathbf{ab}?\mathtt{Confirm}.\checkmark\\ \mathtt{Reject}.\checkmark\\ \mathtt{Propose}.\mathbf{ab}?\begin{cases}\mathtt{Accept}.\mathbf{ba}!\mathtt{Confirm}.\checkmark\\ \mathtt{Reject}.\checkmark\\ \mathtt{Propose(Int)}.X\end{cases}\end{cases}$$

Local types $pq!\{t_i.L_i\}_{1\leq i\leq n}$ and $pq?\{t_i.L_i\}_{1\leq i\leq n}$ specify the send and receive of a value of data type t_i from role p to role q, followed by L_i, for some $1 \leq i \leq n$; we omit braces when $n=1$. Local type $\checkmark$ specifies termination. The following **process, well-typed by the local type,** implements a version of Bob:

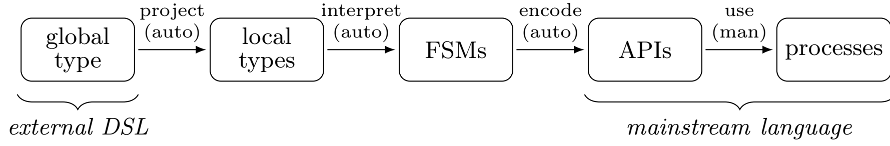

Fig. 3: Workflow of external-DSL-based MPST tools

$$P_{\mathtt{b}} = \mathtt{s}[\![\mathbf{ab}]\!]?_\texttt{:Propose(Int)}\ .\ \mathbf{loop}_{\mathtt{s}}\ \mathtt{s}[\![\mathbf{ab}]\!]!\texttt{propose(11)}\ .\ \mathtt{s}[\![\mathbf{ab}]\!]?\begin{cases}_\texttt{:Accept}\ .\ \mathtt{s}[\![\mathbf{ab}]\!]!\texttt{confirm}\ .\ \mathbf{0}\\ _\texttt{:Reject}\ .\ \mathbf{0}\\ _\texttt{:Propose(Int)}\ .\ \mathbf{recur}_{\mathtt{s}}\end{cases}$$

Process $x[\![pq]\!]!e$ implements the send of the value of expression e from role p to role q, followed by P, in session x. Process $x[\![pq]\!]?\{x_i{:}t_i\,.\,P_i\}_{1\leq i\leq n}$ implements the receive of a value of data type t_i into variable x_i, followed by P_i, in session x, for some $1\leq i\leq n$; we omit braces when $n=1$. Process $\mathbf{0}$ implements termination. Processes $\mathbf{loop}_x\ P$ and $\mathbf{recur}_x$ implement iteration in session x. □

In practice [34], the premier approach to combine MPST with mainstream languages—in the absence of native support—is based on: (1) external DSLs[1] to write global types; (2) associated tooling to generate corresponding code in mainstream languages, including Scribble [17,18], its extensions [5,8,9,26–30,36], StMungo [25], `mpstpp` [24], νScr [35], Pompset [6], Teatrino [2], and Oven [12].

The key ideas of the external-DSL-based approach were originally conceived by Deniélou, Hu, and Yoshida. It is based on two insights: local types can be interpreted as *finite-state machines* (FSM) [10,11], where states and transitions model sends and receives; FSMs can be encoded as object-oriented *application programming interfaces* (API) [17,18], where classes and methods model states and transitions. Fig. 3 visualises the workflow. First, the programmer writes a global type in a DSL; this is the input of the MPST tool. Next, the MPST tool projects the global type to local types, interprets the local types as FSMs, and encodes the FSMs as APIs in the mainstream language; this is the output of the MPST tool. Last, the programmer uses the APIs to write processes.

Example 2. Fig. 4 shows a global type for Negotiation (cf. G in Exmp. 1), written in the external DSL of Scribble. Statement t `from` p `to` q specifies the communication of a value of data type t from p to q. Statement `choice at` r `{` G_1 `}` `or` $\cdots$ `or` `{` G_k `}` specifies a choice among $G_1,\ldots,G_k$ made by r.

Fig. 5 shows the FSM for Bob, derivable from Fig. 4. Transition labels $pq!t$ and $pq?t$ specify the send and receive of a value of data type t from p to q.

Fig. 6 shows a *callback-based API* for Bob in Scala, derivable from Fig. 5. Trait `S`i in the API corresponds with state i of the FSM; methods of trait `S`i

[1] A *domain-specific language* (DSL) is either *external* or *internal*. External DSLs are stand-alone languages with their own dedicated syntax, while internal DSLs are embedded languages into a *general-purpose language* (GPL) with syntax inherited from that GPL. Both approaches have advantages and disadvantages [13].

```
Propose from A to B; rec X {
  choice at B
      { Accept  from B to A;
        Confirm from A to B; }
   or { Reject  from B to A; }
   or { Propose from B to A;
        choice at A
            { Accept  from A to B;
              Confirm from B to A; }
         or { Reject  from A to B; }
         or { Propose from A to B;
              continue X; } } }
```

Fig. 4: Global type for Negotation (Scribble)

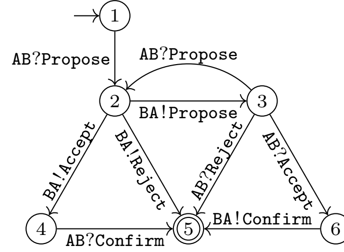

Fig. 5: FSM for Bob

```
trait Loop[S]:
  def loop(f: ((S => S5, S) => S5)): S5

trait S1:
  def recvFromA(f: (Propose, S2) => S5): S5

trait S2 extends Loop[S2]:
  def sendToA(v: Accept,  f: S4 => S5): S5
  def sendToA(v: Reject,  f: S5 => S5): S5
  def sendToA(v: Propose, f: S3 => S5): S5

trait S3 extends Loop[S3]:
  def recvFromA(f1: (Accept,  S6) => S5,
                f2: (Reject,  S5) => S5,
                f3: (Propose, S2) => S5): S5

... // traits S4, S5, and S6
```

Fig. 6: Callback-based API for Bob (Scala)

```
class Propose(val x: Int)
class Accept
class Reject
class Confirm

val v = new Propose(11)

def bob(s1: S1): S5 =
  s1.recvFromA((x, s2) =>
    s2.loop((recur, s2) =>
      s2.sendToA(v, s3 =>
        s3.recvFromA(
          (_, s6) =>
            s6.sendToA(...),
          (_, s5) => s5,
          (_, s2) =>
            recur(s2)))))
```

Fig. 7: Process for Bob

correspond with transitions of state i. Traits `S2` and `S3` also extend trait `Loop` to be able to start callback-based iteration in states 2 and 3 (i.e., these are the only states on a cycle in the FSM) in a type-sound manner. We note that each method and each callback returns a value of type `S5` to ensure that the program can terminate only when the final state has been reached.

To demonstrate the usage of the API, Fig. 7 shows a process for Bob (cf. P in Exmp. 1). The idea is to write a function, `bob`, that consumes an “initial state object” `s1` as input and produces a “final state object” `s5` as output. First, the only communication action that can be performed, is the one for which `s1` has a method (receiving). When that method is called, the actual receive is performed, and the callback is called with the received value `x` and a fresh “successor state object” `s2`. Next, the only communication actions that can be performed, are the ones for which `s2` has a method (sending). And so on. □

This work The external-DSL-based approach is well-established in the MPST literature: it is used in all MPST tools [5, 6, 8, 9, 12, 17, 18, 20, 24–30, 35, 36] that support *classical* MPST as in Fig. 1 (global types and projection; fully automatic; static up-to linearity). However, despite the major impact, it has two weaknesses:

Programming friction: The usage of an external DSL to specify protocols as global types causes programming friction. In general, this is a well-documented issue with external DSLs (e.g., [13]): new syntax needs to be learned; new tools to edit DSL code need to be adopted; extra effort is needed to intermix DSL code with the mainstream language.

Leaky abstractions: As demonstrated in Exmp. 2, APIs generated by MPST tools leak internal details: global types are essentially declarative, whereas the FSMs that seep through the APIs are essentially imperative. This representational gap causes dissonance between the level of abstraction at which global types are produced by the programmer (before API generation), and the level of abstraction at which local types are consumed by that same programmer *in terms of FSMs* (after API generation).

To avoid these weaknesses, we explore a different approach and study the question of how to support classical MPST by using *internal DSLs*. Answering this question positively, we present the `mpst.embedded` library: it leverages Scala's "lightweight form of dependent typing" [3], called *match types*, to embed global/local types directly into Scala. As a result, `mpst.embedded` offers a frictionless interface between global/local types and processes (i.e., no new syntax, editors, or other tools need to be adopted). Moreover, `mpst.embedded` avoids leaky abstractions by not relying on FSMs; global/local types are first-class citizens.

In this way, `mpst.embedded` is the first internal-DSL-based MPST tool that supports all key aspects of classical MPST as in Fig. 1 (unlike Imai et al. [20], who do not support n-ary choice and require extra manual work to guide projection). This is a significant contribution, because: (a) internal DSLs have advantages over external DSLs, but (b) it is far from obvious how to build an internal DSL for MPST in a mainstream language without native support for session types.

Technically, to apply classical MPST and offer static guarantees, some form of compile-time computation is needed. This is the role of match types. They are essentially match expressions at the type level, which are evaluated by the Scala compiler as part of its static analysis, and which we use in this work to embed MPST theory. That is, the Scala compiler can check the typing rules of MPST theory by evaluating carefully crafted match types.

First, through extensive examples, we give an overview of the capabilities of `mpst.embedded` (Sect. 2 and Sect. 3). Next, we present technical details (Sect. 4). Last, we conclude this paper with related work and future work (Sect. 5).

2 A Tour of `mpst.embedded`: Basic Features

Global types Fig. 8 (top rows) shows the correspondence between global types in `mpst.embedded` and in MPST theory. In `mpst.embedded`, each global type G is built from classes `Com`, `End`, `Loop`, and `Recur`. The third type parameter of `Com` is an n-ary product type, called "the branches". Type parameter X of `Loop` is bound in type parameter G to the whole `Loop[`X`, `G`]`, to embed a recursive type. Each role p or q, and each recursion variable X, is a Scala *string literal type* (e.g., `"foo"` is a type with one inhabitant, `"foo"`). Each data type t is a Scala type.

mpst.embedded	MPST theory
Global types:	
Com[p, q, ((t_1, G_1), ..., (t_n, G_n))],	$p \twoheadrightarrow q{:}\{t_i.G_i\}_{1 \leq i \leq n}$,
End, Loop[X, G], Recur[X]	$\checkmark$, $\mu X.G$, X
Local types:	
Send[p, q, ((t_1, L_1), ..., (t_n, L_n))],	$pq!\{t_i.L_i\}_{1 \leq i \leq n}$,
Recv[p, q, ((t_1, L_1), ..., (t_n, L_n))],	$pq?\{t_i.L_i\}_{1 \leq i \leq n}$,
End, Loop[X, L], Recur[X]	$\checkmark$, $\mu X.L$, X
Processes	
x.send(q, e, (_) => P),	$x[\![pq]\!]!e$,
x.recv(p, ((x_1: t_1, _) => P_1, ..., (x_n: t_n, _) => P_n)),	$x[\![pq]\!]?\{x_i{:}t_i.P_i\}_{1 \leq i \leq n}$,
x.loop((recur, _) => P), recur(x)	$\mathbf{loop}_x\ P$, $\mathbf{recur}_x$

Fig. 8: Correspondence between `mpst.embedded` (left) and MPST theory (right)

Example 3. Fig. 9 shows a global type for Negotiation (cf. G in Exmp. 1). □

Local types and projection Fig. 8 (middle rows) shows the correspondence between local types in `mpst.embedded` and in MPST theory. We add that local types can be computed from global types fully automatically and statically via type `Proj`: the Scala compiler reduces `Proj[`G, r`]` to the projection of G onto r.

Example 4. Fig. 10 shows a local type for Bob (cf. $L_{\mathtt{b}}$ in Exmp. 1). Alternatively, it can be computed by having the Scala compiler reduce `Proj[S, "B"]`. □

Processes and type checking Fig. 8 (bottom rows) shows the correspondence between processes in `mpst.embedded` and in MPST theory. In `mpst.embedded`, each process is a sequence of calls to methods `send`, `recv`, `loop`, and `recur` of class `Local`. This generic class has two type parameters: one to represent a role (enacted by the process), and another one to represent a local type (with which the process must comply). In turn, instances of `Local` are obtained through calls to method `init` of class `Global`. This generic class has one type parameter to represent a global type (with which all processes must comply). Method `init` consumes a role, initialises the session for it, and produces a `Local` object for it. Calls to `init` are *blocking*: they return only when *all* processes have called `init`.

Intuitively, `Global` and `Local` objects represent executable sessions from the global and local perspective, leveraging the same abstractions as the global and local types by which they are parametrised (no leaky abstractions).

Example 5. Fig. 11 shows processes for Alice and Bob on lines 1–19 and 21–31 (cf. P in Exmp. 1), plus session initiation on lines 33–35. We make three remarks:

- The process for Bob looks similar to Fig. 7. However, Fig. 11 is defined in terms of communication actions in a session (`Local` objects), whereas Fig. 7 is defined in terms of transitions of an FSM (`S`i objects).

```
1  type S =
2    Com["A", "B", ((Propose,
3      Loop["X",
4        Com["B", "A", (
5          (Accept,  Com["A", "B", ((Confirm, End))]),
6          (Reject,  End),
7          (Propose, Com["A", "B", (
8            (Accept,  Com["B", "A", ((Confirm, End))]),
9            (Reject,  End),
10           (Propose, Recur["X"]))]))]]))]
```

Fig. 9: Global type for Negotiation

```
1  type `S@B` = // equivalent to Proj[S, "B"] -- S is defined in Fig. 9
2    Recv["A", "B", ((Propose,
3      Loop["X",
4        Send["B", "A", (
5          (Accept,  Recv["A", "B", ((Confirm, End))]),
6          (Reject,  End),
7          (Propose, Recv["A", "B", (
8            (Accept,  Send["B", "A", ((Confirm, End))]),
9            (Reject,  End),
10           (Propose, Recur["X"]))]))]]))]
```

Fig. 10: Local type for Bob

```
1  def alice(
2      s: Local["A", Proj[S, "A"]] // S is defined in Fig. 9
3      ): Local["A", End] =
4    s.send("B", new Propose(5), s =>
5      s.recv("B", (
6        (_, s) => s.send("B", new Confirm, s => s),
7        (_, s) => s,
8        (v, s) =>
9          if
10           v.x < 11
11         then
12           s.send("B", new Accept, s =>
13             s.recv("B", (_, s) => s))
14         else
15           s.send("B", new Propose(6), s =>
16             s.recv("B", (
17               (_, s) => s.send("B", new Confirm, s => s),
18               (_, s) => s,
19               (_, s) => s.send("B", new Reject, s => s)))))))
20
21 def bob(
22     s: Local["B", `S@B`] // `S@B` is defined in Fig. 10
23     ): Local["B", End] =
24   s.recv("A", (_, s) =>
25     s.loop((recur, s) =>
26       // val error = s // redundant line -- only used in Exmp. 6
27       s.send("A", new Propose(11), s =>
28         s.recv("A", (
29           (_, s) => s.send("A", new Confirm, s => s),
30           (_, s) => s,
31           (_, s) => recur(s))))))
32
33 val s = new Global[S]
34 val _ = new Thread(() => { alice(s.init["A"]); () }).start
35 val _ = new Thread(() => { bob  (s.init["B"]); () }).start
```

Fig. 11: Processes for Alice and Bob

– The process for Bob exactly mimics the recursive structure of local type `` `S@B` ``. Such mimicry is not a general requirement for well-typed processes, as demonstrated by the process for Alice: instead of exactly mimicking the recursive structure of `Proj[S, "A"]` (which has a similar recursive structure as `S` in Fig. 9), it mimics two *unfoldings* of `Proj[S, "A"]`, followed by termination. That is, lines 5–15 in Fig. 11 comply with the first unfolding, while lines 16–19 comply with the second unfolding, without entering a loop. □

Using `mpst.embedded`, the Scala compiler statically checks for each call α on a `Local` object, parametrised by local type L, whether or not α complies with L. If not, the Scala compiler reports an error. In this way, `mpst.embedded` assures that well-typedness at compile-time implies safety and liveness at run-time, modulo linear usage of `Local` objects (checked dynamically), and modulo non-terminating/exceptional behaviour (unchecked). These **two provisos** are standard for MPST tools. As the type parameters of `Local` objects are erased at compile-time, only generic `Local` objects exist at run-time.

Example 6. The following protocol violations are reported at compile-time:

– In Fig. 11, replace line 29 with one of the following:
```
(_, s) => s.send("A", new Reject, s => s),  // wrong data type
(_, s) => s.send("C", new Confirm, s => s), // wrong receiver
(_, s) => s.recv("A", (_, s) => s),         // wrong communication action
```
– In Fig. 11, uncomment line 26 and replace line 31 with:
```
(_, s) => recur(error)))))                  // wrong recursive type
```

The following protocol violation is reported as an error at run-time:

– In Fig. 11, replace line 31 with:
```
(_, s) => { recur(s); recur(s) })))))       // linearity violation
```

The technical report [22] contains a screenshot of error reporting. □

Besides protocol violations, additionally, basic well-formedness violations of global types are reported as errors at compile-time; they are checked as part of the instantiation of generic class `Global` (e.g., Fig. 11, line 33). For instance, for `Com`[p, q, ((t_1, G_1), ..., (t_n, G_n))], we always require $p \neq q$ and $n \geq 1$.

3 The Tour, Continued: Advanced Features

Full merging To project global types, an auxiliary partial operator to *merge* local types—the projections—is needed. There are two variants [31]: "plain" (basic) and "full" (advanced). Plain merge is *relatively easy* to support, but it works for *few* local types, so many global types cannot be projected. Conversely, full merge works for *many* local types, but it is *relatively hard* to support. For instance, Imai et al. [20] support only manual full merge (i.e., the programmer must write extra protocol-specific code to guide the computation of projections). In contrast, `mpst.embedded` supports automatic full merge via type `Merg`: the Scala compiler reduces `Merg`[L_1, L_2] to the full merge of L_1 and L_2.

```
type S =
  Com["B1", "S", ((String,
    Com["S", "B1", ((Int,
      Com["S", "B2", ((Int,
        Com["B1", "B2", ((Int,
          T))]))]))]))]

type T =
  Com["B2", "B1", (
    (Ok,
      Com["B2", "S", ((Ok,
        Com["B2", "S", ((String,
          Com["S", "B2", ((Date,
            End))]))]))]),
    (Quit,
      Com["B2", "S", ((Quit,
        End))]))]
```

Fig. 12: Global type for Two-Buyer

```
type `S@S` = // equiv. Proj[S, "S"]
  Recv["B1", "S", ((String,
    Send["S", "B1", ((Int,
      Send["S", "B2", ((Int,
        // ignore Int from B1 to B2
          `T@S`))]))]))]))]

type `T@S` = // equiv. Proj[T, "S"]
  Merg[(
    // ignore Ok from B2 to B1
      Recv["B2", "S", ((Ok,
        Recv["B2", "S", ((String,
          Send["S", "B2", ((Date,
            End))]))]))]),
    // ignore Quit from B2 to B1
      Recv["B2", "S", ((Quit,
        End))])]
```

Fig. 13: Local type for Seller

```
def seller(
    s: Local["S", Proj[S, "S"]]
    ): Local["S", End] =
  s.recv("B1", (_, s) =>
    val v = 11
    s.send("B1", v, s =>
      s.send("B2", v, s =>
        s.recv("B2", (
          (_: Ok, s) =>
            s.recv("B2", (_, s) =>
              val q: "B2" = "B2"
              val v = new Date
              s.send(q, v, s => s)),
          (_: Quit, s) => s)))))
```

Fig. 14: Process for Seller

```
// one session between B1, B2, and S
type S = ... // Fig. 12
type T = ... // Fig. 12

// another session between B2 and B3
type U =
  Com["B2", "B3", ((Int,
    Com["B2", "B3", ((Delegatee,
      Com["B3", "B2", (
        (Ok, End),
        (Quit, End))]))]))]

type Delegatee =
  Local["B2", Proj[T, "B2"]]
```

Fig. 15: Global types for Three-Buyer

```
def buyer2( // Three-Buyer version
    s: Local["B2", Proj[S, "B2"]],
    u: Local["B2", Proj[U, "B2"]]
    ): Local["B2", End] =
  s.recv("S", (x, s) =>
    s.recv("B1", (y, s) =>
      u.send("B3", x - y, u =>
        u.send("B3", s, u =>
          u.recv("B3", (
            (_, u) => u, ...))))))
```

Fig. 16: Process for Buyer2

```
def buyer3(
    u: Local["B3", Proj[U, "B3"]]
    ): Local["B3", End] =
  u.recv("B2", (_, u) =>
    u.recv("B2", (s, u) =>
      val v = new Quit
      u.send("B2", v, u =>
        s.send("B1", v, s =>
          s.send("S", v, s => s))
        u)))
```

Fig. 17: Process for Buyer3

Example 7. The *Two-Buyer* protocol, originally defined in the MPST literature by Honda et al. [16], consists of roles *Buyer1*, *Buyer2*, and *Seller*: "[Buyer1 and Buyer2] wish to buy an expensive book from Seller by combining their money. Buyer1 sends the title of the book to Seller, Seller sends to both Buyer1 and Buyer2 its quote, Buyer1 tells Buyer2 how much she can pay, and Buyer2 either accepts the quote or rejects the quote by notifying Seller." We use an extended version defined by Coppo et al. [7], in which Buyer2 notifies not only Seller about acceptance/rejection, but also Buyer1. In the case of acceptance, Buyer2 sends his address to Seller, and Seller sends back the delivery date to Buyer2.

Fig. 12 and Fig. 13 show a global type for Two-Buyer and a local type for Seller (split into `S`/`S@S` and `T`/`T@S` for presentational reasons.) First, we note that the communication from Buyer1 to Buyer2 on line 5 in the global type has no counterpart on line 5 in the local type; as Seller does not participate in the communication, it is simply skipped in the projection. Second, we note that the communication from Buyer2 to Buyer1 on line 9 in the global type is ignored, too, but the projections of the two branches do need to be combined into one. This is achieved by having the Scala compiler reduce

```
Merg[Recv["B2", "S", ((Ok, ...))], Recv["B2", "S", ((Quit, ...))]]
```

to `Recv["B2", "S", ((Ok, ...), (Quit, ...))]`.

Fig. 14 shows a process for Seller. It demonstrates that merging is a type-level concept, hidden from the programmer: the Scala compiler reduces `Merg[...]` and type-checks the code against the result transparently. □

Delegation Sessions are *higher-order*: `Local` object `s` for a first session can be *delegated* between processes via `Local` object `u` for a second session, by sending `s` via `u`. In the presence of delegation, within each session, well-typedness at compile-time continues to imply safety and liveness at run-time (modulo the "two provisos"; page 8). However, between sessions, liveness is not assured; supporting this would require substantial extra technical machinery [7], so none of the existing MPST tools support it.

Example 8. The *Three-Buyer* protocol, originally defined in the MPST literature by Coppo et al. [7], consists of roles *Buyer1*, *Buyer2*, *Buyer3*, and *Seller*. It resembles the Two-Buyer protocol, except that Buyer2 can ask Buyer3 to enact his role on his behalf—unbeknownst to Buyer1 and Seller—through delegation.

Fig. 15 shows global types for Three-Buyer. Global type `S` specifies the first sub-protocol among Buyer1, Buyer2, and Seller; it is identical to the global type for Two-Buyer. Global type `U` specifies the second sub-protocol between Buyer2 and Buyer3. Notably, line 8 specifies the delegation from Buyer2 to Buyer3.

Fig. 16 shows a process for Buyer2: on lines 1–4, `Local` objects for two sessions are consumed as inputs (to engage in two sub-protocols); on lines 5–6, the first session is used; on lines 7–10, the second session is used; on line 8, the remainder of the first session is delegated via the second session. Similarly, Fig. 17 shows a process for Buyer3: on lines 1–3, a `Local` object for the first session is consumed as input; on line 5, a `Local` object for the second session is received.

The process for Seller is exactly the same in Three-Buyer as in Two-Buyer (Fig. 14). In particular, Seller does not know that it communicates with Buyer3 instead of Buyer2. Thus, delegation is hidden from each role not involved. □

Generic global types By embedding global/local types as Scala types, Scala's built-in mechanism of type parametrisation is readily available. This allows the programmer to write *generic global types* with type parameters for roles (common in external-DSL-based MPST tools) and sub-protocols (novel of `mpst.embedded`).

```scala
1 type T[P <: Role, Q <: Role] =
2   Com[P, Q, ((Propose,
3     Loop["X",
4       U[Q, P,
5         U[P, Q, Recur["X"]]]])))]
```

```scala
6 type U[P <: Role, Q <: Role, G <: GType] =
7   Com[P, Q, (
8     (Accept,  Com[Q, P, ((Confirm, End))]),
9     (Reject,  End),
10    (Propose, G))]
```

Fig. 18: Generic global types for Negotiation

$$p \twoheadrightarrow q{:}\{t_i.G_i\}_{i\in I} \upharpoonright p = pq!\{t_i.G_i \upharpoonright r\}_{i\in I} \qquad \checkmark \upharpoonright r = \checkmark$$

$$p \twoheadrightarrow q{:}\{t_i.G_i\}_{i\in I} \upharpoonright q = pq?\{t_i.G_i \upharpoonright r\}_{i\in I} \qquad \mu X.G \upharpoonright r = \mu X.(G \upharpoonright r)$$

$$p \twoheadrightarrow q{:}\{t_i.G_i\}_{i\in I} \upharpoonright r = \bigsqcap\{G_i \upharpoonright r\}_{i\in I} \quad \text{if } r \notin \{p, q\} \qquad X \upharpoonright r = X$$

Fig. 19: Projection in MPST theory

Example 9. To alleviate the repetitive feel of Fig. 9, Fig. 18 shows generic global types that leverage type parameters. Type `U` generically specifies the communication of an acceptance, rejection, or counter-proposal from `P` to `Q` (type parameters for roles), followed by `G` (type parameter for a sub-protocol) in case of a counter-proposal; it can be instantiated twice to replace lines 5–7 and lines 8–10 in Fig. 9. This is done in type `T`, which generically specifies a role-parametric version of the whole `S` in Fig. 9. Thus, `T["A", "B"]` is equivalent to `S` in Fig. 9. □

Consistency `mpst.embedded` also supports explicit *consistency checking* of sets of local types. Details can be found in the technical report [22], as they are rather technical/subtle. We do evaluate consistency checking times in Sect. 4.3, though.

4 Technical Details

As `mpst.embedded` closely follows MPST theory, and as it uses unique parts of the Scala type system, first, we summarise a few essential preliminaries (Sect. 4.1). Next, we describe our embedding of MPST into Scala (Sect. 4.2).

This section focusses on the basic features of Sect. 2. It allows us to keep the necessary background on MPST theory simple and succinct, while still being able to explain the general ideas of the embedding into the Scala type system in sufficient depth. The advanced features of Sect. 3 are based on more complex theoretical concepts, but their embedding follows similar general ideas.

4.1 Preliminaries

MPST theory We summarise the theory behind classical MPST (Fig. 1):

Global types, local types, and processes: The syntax was defined and explained in Fig. 8 (right column) and Exmp. 1.

Projection: Let $G \upharpoonright r$ denote the projection of G onto r; it is defined in Fig. 19. The projection of a communication yields a send if r is the sender, a receive if r is the receiver, or the full merge—denoted by $\sqcap$—of the projected branches otherwise (i.e., r does not participate in the communication).

$$\frac{\Gamma \vdash e : t_j \text{ and } \Gamma, x : L_j \vdash P, \text{ for some } j \in I}{\Gamma, x : pq!\{t_i . L_i\}_{i \in I} \vdash x[\![pq]\!]!e.P} \text{[Send]} \qquad \frac{\text{only } \checkmark \text{ local types in } \Gamma}{\Gamma \vdash \mathbf{0}} \text{[Term]}$$

$$\frac{\Gamma, x : L_i, x_i : t_i \vdash P_i, \text{ for each } i \in I}{\Gamma, x : pq?\{t_i . L_i\}_{i \in I}) \vdash x[\![pq]\!]?\{x_i{:}t_i . P_i\}_{i \in I}} \text{[Recv]} \qquad \frac{\Gamma, x : L[\mu X . L / X] \vdash P}{\Gamma, x : \mu X . L \vdash P} \text{[Unfold]}$$

Fig. 20: Type checking in MPST theory (excerpt)

Type checking: Let $\Gamma \vdash P$ denote well-typedness of P in typing environment Γ; it is defined in Fig. 20. Rule [Send] states that a send from p to q in x is well-typed when the local type of x specifies a send, e is well-typed by t_j, and P is well-typed after setting the local type of x to L_j in the typing environment, for some j. Rule [Recv] states that a receive from p to q in x is well-typed when the local type of x specifies a receive, and P_i is well-typed after setting the local type of x to L_i in the typing environment, for each i. Thus, there is asymmetry: for sending, only one send specified must be implemented, but for receiving, each receive specified must be implemented.

Central theorem: Static well-typedness implies dynamic safety and liveness.

Match types in Scala The main feature of the Scala type system that we take advantage of in `mpst.embedded` is *match types*. We explain it with an example. Suppose that we need to write a function to convert `Int`s and `Boolean`s:

```
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 }
```

However, return type `IntOrBoolean` is not precise enough. For instance, the Scala compiler fails to prove that `convert(5) && false` is safe, as it cannot infer that `convert(5)` is `Boolean`. What is missing, is a relation between the actual type of `x` (e.g., `Int`) and the return type (e.g., `Boolean`). Match types define such relations.

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

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

   Thus, we introduce a type parameter `T` (subtype of `IntOrBoolean`) and declare `x` to be `T`. Also, we declare the return value to be of match type `Convert[T]`.
2. Next, the idea is to define `Convert[T]` in such a way that the relation between the actual type of `x` and the return type can be inferred, as follows:

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

   The Scala compiler *reduces* every occurrence of `Convert[T]` to `Int` or `Boolean`, depending on the instantiation of `T` (e.g., `Convert[Int]` is reduced to `Boolean`).
3. Last, for instance, the Scala compiler correctly succeeds/fails to type-check `convert(5) && false` (safe) and `convert(5) && 6` (unsafe).

Thus, match types are a “lightweight form of dependent typing” [3], to perform “type-level programming”. In the remainder, we use the following built-ins:

```
1 type Proj[G, R] = G match
2   case End           => End
3   case Com[R, q, b] => Send[R, q, Map[b, [E] =>> (Head[E], Proj[Last[E], R])]]
4   case Com[p, R, b] => Recv[p, R, Map[b, [E] =>> (Head[E], Proj[Last[E], R])]]
5   case Com[_, _, b] => MergAll[Map[b, [E] =>> Proj[Last[E], R]]
6   case Loop[x, g]    => Loop[x, Proj[g, R]]
7   case Recur[x]      => Recur[x]
```

Fig. 21: Projection in `mpst.embedded`

- `Head[(T1, ..., Tn)]` and `Last[(T1, ..., Tn)]` reduce to `T1` and `Tn`.
- `Map[(T1, ..., Tn), F]` reduces to `(F[T1], ..., F[Tn])`. We note that `F` can be a *type lambda* of the form `[X] => ... /* do something with X */`.

4.2 Embedding MPST into Scala

Global types, local types, processes As explained in Sect. 2, and as shown in Fig. 8, global types and local types are implemented as classes, while processes are implemented as methods of class `Local`. The communication infrastructure for processes is based on concurrent queues. However, a transport layer for distributed processes is also possible (orthogonal concern).

Projection Fig. 21 shows match type `Proj`. It is used to have the Scala compiler fully automatically and statically compute local types (e.g., line 2 in Fig. 11).

Match type `Proj` has two type parameters: a global type `G` and a role `R` (cf. $G \upharpoonright r$). To reduce `Proj[G, R]`, the Scala compiler matches `G` to a global type constructor, and it produces a local type **exactly as defined in Fig. 19**. By convention, lower case letters in patterns are *type variables*; they are bound to types as part of the matching algorithm. For instance, on lines 3–5 in Fig. 21, `b` is bound to a product type of the form `((T1, G1), ..., (Tn, Gn))`, where each `Ti` is a data type, and each `Gi` is a global type. When `b` is passed to `Map` on lines 3–4, it is converted into `((T1, Proj[G1, R]), ..., (Tn, Proj[Gn, R]))`. Alternatively, when `b` is passed to `Map` on line 5, it is converted into `(Proj[G1, R], ..., Proj[Gn, R])`, which is subsequently passed to `MergAll`; this is a helper match type that reduces to the full merge of all local types in the product type.

Type checking Fig. 22 shows an excerpt of class `Local` related to type checking. The idea is to have the Scala compiler reduce match types `SendCallback` and `RecvCallbacks` to fully automatically and statically compute the *expected types* of the callback arguments of methods `send` and `recv`, given a local type L. The reduction succeeds, and the actual callback argument is well-typed by the expected type, *if, and only if,* the communication action is well-typed by L **exactly as defined in Fig. 20**. Otherwise, the Scala compiler reports an error. In this way, `mpst.embedded` implements the same MPST typing rules as in Fig. 20 in terms of Scala match type reduction, and it provides the same assurances (modulo the "two provisos"; page 8):

```
class Local[R, L] private (val r: R, val net: Network) extends UseOnce:
  def send[Q, D](q: Q, d: D, f: SendCallback[Q, D, L]): Local[R, End] = ...
  def recv[P](p: P, fs: RecvCallbacks[P, L]): Local[R, End] = ...

  type SendCallback[Q, D, L] = L match
    case Send[R, q, b] => q match
      case Q => (Local[R, App[b, D]] => Local[R, End])
    case Loop[x, l] => SendCallback[Q, D, Substitute[l, L, x]]

  type RecvCallbacks[P, L] = L match
    case Recv[p, R, b] => p match
      case P => Map[b, [E] =>> (Head[E], Local[R, Last[E]]) => Local[R, End]]
    case Loop[x, l] => RecvCallbacks[P, Substitute[l, L, x]]

  ... // function loop and type LoopCallback
```

Fig. 22: Type checking in `mpst.embedded` (excerpt)

- if, at compile-time, each process is well-typed by its projection,
- then, at run-time, the session of all processes is safe and live,
- modulo linear usage of `Local` objects (checked dynamically),
- modulo non-terminating/exceptional behaviour (unchecked).

We now explain `send` and `recv`. Regarding `send`, Fig. 20 states that a send is well-typed if the local type specifies it directly (rule [Send]) or indirectly (rule [Unfold]). These cases correspond precisely to the two cases in `SendCallback`:

- Lines 6–7 state that a send is well-typed if the sender, receiver, and data type match the send of the local type `L`, and if the callback is a function that consumes a `Local` object, parametrised by the selected branch of `L`, namely `App[b, D]`. We note that `App[((T1, L1), ..., (Tn, Ln)),Ti]` reduces to `Li`.
- Line 8 states that a send is also well-typed when it is well-typed by the unfolding of the local type. We note that `Substitute[L1, L2, X]` reduces to a version of `L1` in which each occurrence of `Recur[X]` is replaced with `L2`.

Regarding `recv`, similarly, Fig. 20 states that a receive is well-typed if the local type specifies the receive directly (rule [Recv]) or indirectly (rule [Unfold]). Due to the asymmetry between sends and receives, `SendCallback` (singular) reduces to a single function type, while `RecvCallbacks` (plural) reduces to a product of function types, computed using `Map`. Besides that, they follow the same ideas.

4.3 Evaluation and Discussion

Compile-time performance To validate the practical feasibility of using `mpst.embedded`, we systematically measured the type checking times during *non-incremental* compilation of all examples in Sect. 2 and Sect. 3, as well as twelve additional examples from the MPST literature [7,30,31] and the Scribble repository [14].[2] This is a representative set of protocols, previously developed by other

[2] Run-time performance (e.g., latency/throughput) depends on the transport mechanism for message passing, which is orthogonal to the contributions of this paper.

Table 1: Type checking times in milliseconds, reported as $\mu \pm \sigma$, where μ is the average (of 31 measurements), and where σ is the standard deviation

protocol	**type checking** without consistency	**type checking** with consistency	**difference**†
Negotiation (Exmp. 5)	$1,399 \pm 118$ ms	$1,254 \pm 36$ ms	-145 ± 125 ms
Negotiation (Exmp. 9)	$1,299 \pm 85$ ms	$1,240 \pm 22$ ms	-60 ± 88 ms
Two-Buyer (Exmp. 7)	$1,399 \pm 31$ ms	$1,658 \pm 50$ ms	258 ± 59 ms
Three-Buyer (Exmp. 8)	$1,489 \pm 57$ ms	$1,728 \pm 50$ ms	239 ± 76 ms
Three-Buyer [7]	$1,341 \pm 57$ ms	$1,622 \pm 71$ ms	280 ± 78 ms
OAuth2 Fragment [31]	713 ± 24 ms	*inconsistent*	*inconsistent*
Rec. Two-Buyers [31]	775 ± 24 ms	*inconsistent*	*inconsistent*
Rec. Map/Reduce [31]	$1,016 \pm 45$ ms	*inconsistent*	*inconsistent*
MP Workers [31]	891 ± 27 ms	*inconsistent*	*inconsistent*
Game [30]	$1,095 \pm 35$ ms	$1,338 \pm 29$ ms	243 ± 46 ms
Adder [14]	763 ± 21 ms	781 ± 19 ms	18 ± 28 ms
Booking [14]	$1,099 \pm 35$ ms	*inconsistent*	*inconsistent*
Fibonacci [14]	759 ± 22 ms	779 ± 17 ms	20 ± 28 ms
HTTP [14]	$1,703 \pm 41$ ms	$1,838 \pm 77$ ms	134 ± 88 ms
Loan Application [14]	879 ± 48 ms	1132 ± 29 ms	253 ± 56 ms
SMTP [14]	$1,726 \pm 70$ ms	2079 ± 128 ms	353 ± 146 ms

† The difference between type checking times *without* consistency $\mu_1 \pm \sigma_1$ and *with* consistency $\mu_2 \pm \sigma_2$ are reported as $\mu \pm \sigma$, where $\mu = \mu_2 - \mu_1$ and $\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$

researchers (including the protocols in our examples in Sect. 2 and Sect. 3), of various sizes, that exercise all aspects of classical MPST theory.[3]

To measure only the protocol-related type checking times, the processes contained almost no computation code; just communication actions in compliance with the protocol. The measurements were obtained using an Intel i7-8569U processor (4 physical/4 virtual cores at 2.8 GHz) and 16 GB of memory, running macOS 14.0, OpenJDK 18.0.2, and Scala 3.3.1. We ran the measurements with consistency checking disabled and enabled, to be able to study the difference.

Table 1 shows the results, averaged over 31 runs per protocol. We make two main observations. First, without consistency checks, the type checking times seem sufficiently low for the usage of `mpst.embedded` to be practically feasible: less than two seconds for the biggest protocol in our benchmark set (SMTP). Moreover, our measurements were obtained using non-incremental compilation and, as such, constitute an upper bound on the expected type checking delays when using incremental compilation. Anecdotally, in our development environment (Visual Studio Code 1.87 with the Metals 1.30 extension for Scala pro-

[3] That is, the theory as originally defined by Honda et al. [16], but presented in the more recent style of, e.g., Scalas–Yoshida [31], including the full merge operator.

gramming), when using incremental compilation, the type checking delays were significantly lower (<100 ms) than those in Table 1, and not disruptive at all.

Second, with consistency checks (i.e., five examples violate consistency; this was expected), the results show that some overhead is added, but it does not make the usage of `mpst.embedded` infeasible (<500 ms). Also, when using incremental compilation, the type checking delays continued to not get in the way.

Experience The implementation of the benchmark set turned out to be, in its own right, a validation activity to experience whether or not the type checker catches all mistakes in practice. This is because, until a protocol implementation is finished, it does not comply with the specification yet. Thus, all until the end, the type checker reports errors to point out missing pieces. This guidance by the type checker effectively prevented us from making unintended programming mistakes, especially when writing the implementations of HTTP and SMTP (which are the more complicated protocols in our benchmark set). It would be interesting to try to reproduce these anecdotal findings in a larger user study.

Expressiveness Our benchmark set shows that `mpst.embedded` is feature-complete relative to classical MPST theory,[3] with full merging (e.g., the OAuth2 fragment requires full merge), as intended. Moreover, while the ability to write *generic* global types does not add expressive power in the formal sense, it enables better *reuse* of global types and serves as an abstraction/composition mechanism: it allows large protocols to be split into separate smaller sub-protocols—specified as generically as possible to maximise the opportunity for reuse—which can then be "invoked" from each other with concrete arguments. Such generic sub-protocols can also be packaged into libraries and shared between projects.

5 Conclusion

Related work Closest to our approach in this paper is the work by Imai et al. [20]. They developed an internal DSL in OCaml to specify protocols and verify processes based on MPST. However, their tool does *not* support all key aspects of classical MPST as in Fig. 1: it supports only binary choices instead of n-ary choices (e.g., Exmp. 1, which has ternary choices, is not supported), and it is not fully automatic (i.e., Imai et al. require the programmer to manually write extra protocol-specific code to project global types). In contrast, `mpst.embedded` supports n-ary choices and is fully automatic.

Another related tool is the Discourje library [15], which offers an MPST-based internal DSL in Clojure. However, Discourje does all verification dynamically, whereas `mpst.embedded` performs all verification statically up-to linearity.

There are four existing tools to combine MPST with Scala: *Scribble-Scala* [30], *Pompset* [6], *Teatrino* [2], and *Oven* [12]. Table 2 summarises the differences:

– *DSLs to specify protocols as global types:* Scribble-Scala, Pompset, and Teatrino are based on the external DSL of Scribble, while Oven is based on an external DSL for regular expressions.

Table 2: Comparison of MPST tools for Scala

	DSL	**projection**	**interpretation**	**encoding**
Scribble-Scala [30]	external	syntactic	FSMs	`lchannels`
Pompset [6]	external	syntactic	pomsets	vanilla Scala
Teatrino [2]	external	syntactic	–	Effpi
Oven [12]	external	semantic	FSMs	vanilla Scala
`mpst.embedded`	internal	syntactic	–	vanilla Scala

- *Projection of global types:* Scribble-Scala, Pompset, and Teatrino apply the classical *structural* projection operator (defined in terms of the syntax of global types; Sect. 4.2), while Oven applies a non-classical *behavioural* projection operator (defined in terms of the operational semantics of global types). The latter has additional expressive power to support the usage of regular expressions as global types [23].
- *Interpretation of local types:* Different from Fig. 3, Pompset uses *partially-ordered multisets* instead of FSMs as an intermediate operational model, while Teatrino directly encodes local types as APIs in Scala.
- *Encoding as APIs:* The APIs generated by Scribble-Scala and Teatrino are built on top of the existing libraries `lchannels` and Effpi (discussed in more detail below), while Pompset and Oven do not rely on such existing libraries.

Besides these existing tools to combine multiparty session typing with Scala (including global types and projection), there also exist libraries to combine binary session typing with Scala (excluding global types and projection), namely `lchannels` [30] and *Effpi* [32]. Conceptually, as `mpst.embedded` targets multiparty instead of binary, it is not really comparable to `lchannels` and Effpi. Technically, moreover, `lchannels` and Effpi do not use match types.

Future work Many extensions of MPST theory have been proposed. We are keen to explore which of them can be incorporated in `mpst.embedded` using match types. For instance, an important feature that we believe is compatible with `mpst.embedded` and match types is *parameterised MPST with indexed roles* as developed by Castro et al. [5]. Another feature that seems representable using match types, is *MPST with refinements* along the lines of Zhou et al. [36]. In contrast, a feature that seems prohibitively difficult to incorporate, is *timed MPST* [4]: match types seem unsuitable to statically offer real-time guarantees.

Data Availability Statement

The artifact is available on Zenodo [21]. It contains: (1) `mpst.embedded`; (2) the examples in the paper; (3) reproduction instructions for our evaluation.

References

1. Ancona et al., D.: Behavioral types in programming languages. Foundations and Trends in Programming Languages **3**(2-3), 95–230 (2016)
2. Barwell, A.D., Hou, P., Yoshida, N., Zhou, F.: Designing asynchronous multiparty protocols with crash-stop failures. In: ECOOP. LIPIcs, vol. 263, pp. 1:1–1:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023)
3. Blanvillain, O., Brachthäuser, J.I., Kjaer, M., Odersky, M.: Type-level programming with match types. Proc. ACM Program. Lang. **6**(POPL), 1–24 (2022)
4. Bocchi, L., Yang, W., Yoshida, N.: Timed multiparty session types. In: CONCUR. LNCS, vol. 8704, pp. 419–434. Springer (2014)
5. Castro, 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. PACMPL **3**(POPL), 29:1–29:30 (2019)
6. Cledou, G., Edixhoven, L., Jongmans, 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)
7. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. Math. Struct. Comput. Sci. **26**(2), 238–302 (2016)
8. Cutner, Z., Yoshida, N.: Safe session-based asynchronous coordination in rust. In: COORDINATION. Lecture Notes in Computer Science, vol. 12717, pp. 80–89. Springer (2021)
9. Cutner, Z., Yoshida, N., Vassor, M.: Deadlock-free asynchronous message reordering in rust with multiparty session types. In: PPoPP. pp. 246–261. ACM (2022)
10. Deniélou, P., Yoshida, N.: Multiparty session types meet communicating automata. In: ESOP. LNCS, vol. 7211, pp. 194–213. Springer (2012)
11. Deniélou, P., Yoshida, N.: Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In: ICALP (2). LNCS, vol. 7966, pp. 174–186. Springer (2013)
12. Ferreira, F., Jongmans, S.: Oven: Safe and live communication protocols in scala, using synthetic behavioural type analysis. In: ISSTA. pp. 1511–1514. ACM (2023)
13. Fowler, M.: Domain-Specific Languages. The Addison-Wesley signature series, Addison-Wesley (2011)
14. GitHub, Inc: scribble-java/scribble-demos/scrib at ccb0e48d69c6e3088e746138099c3183ca1ac79b · scribble/scribble-java, accessed January 2025, `https://github.com/scribble/scribble-java/tree/ccb0e48d69c6e3088e746138099c3183ca1ac79b/scribble-demos/scrib`
15. Hamers, R., Jongmans, S.: Discourje: Runtime verification of communication protocols in clojure. In: TACAS (1). LNCS, vol. 12078, pp. 266–284. Springer (2020)
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)
17. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. LNCS, vol. 9633, pp. 401–418. Springer (2016)
18. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. LNCS, vol. 10202, pp. 116–133. Springer (2017)
19. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016)

20. 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)
21. Jongmans, S.: Multiparty session typing, embedded (artifact) (2025), `https://doi.org/10.5281/zenodo.14201951`
22. Jongmans, S.: Multiparty session typing, embedded (technical report) (2025), `https://doi.org/10.48550/arXiv.2501.17741`
23. Jongmans, S., Ferreira, F.: Synthetic behavioural typing: Sound, regular multiparty sessions via implicit local types. In: ECOOP. LIPIcs, vol. 263, pp. 42:1–42:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023)
24. Jongmans, S., Yoshida, N.: Exploring type-level bisimilarity towards more expressive multiparty session types. In: ESOP. LNCS, vol. 12075, pp. 251–279. Springer (2020)
25. 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)
26. Lagaillardie, N., Neykova, R., Yoshida, N.: Implementing multiparty session types in rust. In: COORDINATION. LNCS, vol. 12134, pp. 127–136. Springer (2020)
27. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay safe under panic: Affine rust programming with multiparty session types. In: ECOOP. LIPIcs, vol. 222, pp. 4:1–4:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
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., 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)
30. 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 fuer Informatik (2017)
31. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. Proc. ACM Program. Lang. **3**(POPL), 30:1–30:29 (2019)
32. Scalas, A., Yoshida, N., Benussi, E.: Verifying message-passing programs with dependent behavioural types. In: PLDI. pp. 502–516. ACM (2019)
33. Tu, T., Liu, X., Song, L., Zhang, Y.: Understanding real-world concurrency bugs in go. In: ASPLOS. pp. 865–878. ACM (2019)
34. Yoshida, N.: Programming language implementations with multiparty session types. In: Active Object Languages: Current Research Trends, Lecture Notes in Computer Science, vol. 14360, pp. 147–165. Springer (2024)
35. Yoshida, N., Zhou, F., Ferreira, F.: Communicating finite state machines and an extensible toolchain for multiparty session types. In: FCT. LNCS, vol. 12867, pp. 18–35. Springer (2021)
36. 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)