Multiparty Session Typing in Java, Deductively

Jelle Bouma[1], Stijn de Gouw[1], and Sung-Shik Jongmans[1,2]

[1] Open University of the Netherlands, Heerlen, the Netherlands
[2] Centrum Wiskunde & Informatica (CWI), Amsterdam, the Netherlands

Abstract. Multiparty session typing (MPST) is a method to automatically prove safety and liveness of protocol implementations relative to specifications. We present BGJ: a new tool to apply the MPST method in combination with Java. The checks performed using our tool are purely static (all errors are reported early at compile-time) and resource-efficient (near-zero cost abstractions at run-time), thereby addressing two issues of existing tools. BGJ is built using VerCors, but our approach is general.

1 Introduction

Construction and analysis of distributed systems is hard. One of the challenges is this: given a specification $\mathcal{S}$ of the *roles* and the *protocols* an implementation $\mathcal{I}$ of *processes* and *communication sessions* should fulfil, can we prove that $\mathcal{I}$ is *safe* and *live* relative to $\mathcal{S}$? Safety means "bad" communication actions never happen: if a channel action happens in $\mathcal{I}$, then it is allowed by $\mathcal{S}$. Liveness means "good" communication actions eventually happen (communication deadlock freedom). *Multiparty session typing* (MPST) [14, 15] is a method to automatically prove safety and liveness of protocol implementations. The idea is shown 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 (e.g., "first, a number from Alice to Bob; next, a boolean from Bob to Carol").
2. Next, G is decomposed into local types $L_1, \ldots, L_n$ by *projecting* G onto every role. Each local type models the behaviour of one process alone (e.g., for Bob, "first, he receives from Alice; next, he sends to Carol").
3. Last, absence of communication errors is verified by *type-checking* every process P_i against its local type L_i. MPST theory assures that well-typedness at compile-time implies safety and liveness at run-time.

The following simple example demonstrates global types and local types in *Scribble* notation [28], as used in the Scribble tool [16, 17] for the MPST method.

Example 1. The *Adder* protocol [12] consists of two roles: *Client* (`C`) and *Server* (`S`). Client either asks Server to add two numbers (`Add`-message with two `Int`-payloads) or tells Server goodbye (`Bye`-message). In the former case, Server tells Client the result (`Res`-message). This is repeated until Server is told goodbye.

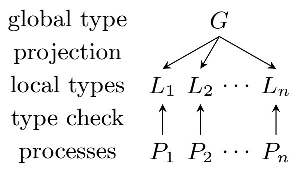

Fig. 1: MPST method

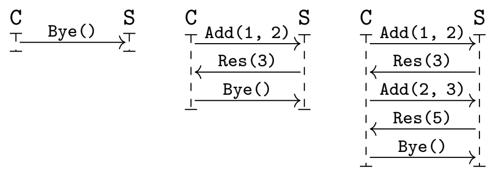

Fig. 2: Example runs of Adder

```
global Adder(role C, role S) {
  choice at C {
    Add(Int, Int) from C to S;
    Res(Int) from S to C;
    do Adder(C, S); // recur
  } or {
    Bye() from C to S; } }
```

Fig. 3: Global type for Adder

```
local Adder(role C, role S) at C {
  choice at C {
    Add(Int, Int) to S; // send
    Res(Int) from S;    // receive
    do Adder(C, S);
  } or {
    Bye() to S; } }     // send
```

Fig. 4: Local type for Client in Adder

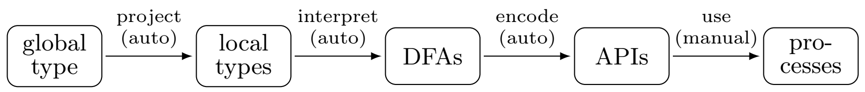

Fig. 5: Workflow of API-generation-based tools for the MPST method

Figure 2 shows three example runs as sequence diagrams. Figure 3 shows the global type. Notation "$m(t_1, \ldots, t_n)$ from p to q" specifies the *communication* of a message of type m with payloads of types $t_1, \ldots, t_n$ from role p to role q. Notation "choice at r { G_1 } or $\cdots$ or { G_k }" specifies a *choice* among branches $G_1, \ldots, G_k$ made by role r. Figure 4 shows the local type for Client. The notation for local types resembles the notation for global types, except that communications are broken up into *sends* ("$m(t_1, \ldots, t_n)$ to q") and *receives* ("from p"). □

A premier approach to apply the MPST method in combination with mainstream programming languages is based on *API generation* (Figure 5); it is used in the majority of MPST tools, including Scribble [16, 17], its extensions [5, 8, 9, 22, 23, 25, 27, 32, 35], StMungo [21], νScr [34], mpstpp [20], and Pompset [6]. The main ideas, first conceived by Deniélou/Hu/Yoshida and pursued in Scribble, follow two insights: **(a)** local types can be interpreted as *deterministic finite automata* (DFA) [10, 11], where every transition models a send/receive action; **(b)** DFAs can be encoded as object-oriented *application programming interfaces* (API) [16, 17], where classes and methods model states and transitions.

Example 2. Figure 6 shows the DFA and a Java API for Client in Adder (Example 1), in the style of Scribble. Transition labels of the form $q!m(t_1, \ldots, t_n)$ and $p?m(t_1, \ldots, t_n)$ in the DFA specify the send to q and the receive from p of a message of type m with payloads of types $t_1, \ldots, t_n$. Classes `State1`, `State2`, and `State3` in the API correspond to states 1, 2, and 3 of the DFA; the methods of class `State`i in the API correspond to the transitions from state i in the DFA.

Figure 7 shows a process for Client, using the Java API. The idea is to write method `client` that consumes an "initial state object" `s1` as input and produces

```java
class UseOnce { // superclass
  boolean b = false;
  void use() { if (b) throw new RuntimeException(); b = true; } }

class State1 extends UseOnce { // subclass
  State2 sendAddToS(int x, int y) { use(); ... }
  State3 sendByeToS()             { use(); ... } }

class State2 extends UseOnce { // subclass
  State1 recvResFromS(int[] buff) { use(); ... } }

class State3 extends UseOnce { }
```

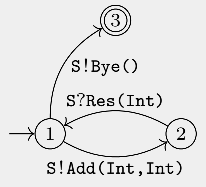

Fig. 6: DFA and Java API for Client in Adder (Scribble-style)

a "final state object" `s3` as output. First, the only communication actions that can be performed, are those for which `s1` has a method. When called, the communication action is performed and a fresh "successor state object" `s2` (line 4) or `s3` (line 8) is returned. Next, the only communication actions that can be performed, are those for which `s2` or `s3` has a method. And so on. By using state objects in this way, a run of method `client` simulates a run of the DFA. □

```java
1 State3 client(State1 s1) {
2   int x = 1; int y = 2;
3   while (x + y < 100) {
4     State2 s2 = s1.sendAddToS(x, y);
5     int[] buff = new int[1];
6     s1 = s2.recvResFromS(buff);
7     x = y; y = buff[0]; }
8   State3 s3 = s1.sendByeToS();
9   return s3; }
```

Fig. 7: Process for Client in Adder

However, existing API-generation-based tools that follow Example 2 in MPST practice, do not fully meet the promise of MPST theory, in two ways:

1. **Mixed static/dynamic checks:** To ensure safety and liveness, every non-final state object must be used *linearly* (exactly one method call). However, the type systems of most mainstream programming languages are too weak to check linear usage statically. Instead, dynamic checks are needed (e.g., method `use` in Figure 6). As a result, MPST practice is weaker than MPST theory: in MPST practice, some errors are reported late at run-time, whereas in MPST theory, all errors are reported early at compile-time.
2. **Resource-inefficient checks:** Every time when a communication action is performed, a fresh state object is created. This costs time (allocation; garbage collection) and space. As a result, MPST practice is costlier at run-time than MPST theory: in MPST practice, API-encodings of DFA-interpretations of local types have a real footprint (proportionate to the number of communication actions), whereas in MPST theory, local types are zero cost abstractions.

In this paper, we present *BGJ*: a new API-generation-based tool to apply the MPST method in combination with Java. The checks performed using BGJ are purely static (all errors are reported early at compile-time) and resource-efficient (near-zero cost abstractions at run-time), thereby addressing the issues above. Instead of building a new static analyser from scratch, we leverage a state-of-the-art deductive verifier for Java, namely *VerCors* [2]. Under active development for years, VerCors has been used in industrial case studies, too [18, 26, 30]. We note that our approach is generic, though, while our current tool is VerCors-specific.

```java
class DFA {
  int state;
  //@ ensures Perm(state, write);
  //@ ensures state == 1;
  DFA() { state = 1; }

  //@ context  Perm(state, write);
  //@ requires state == 1;
  //@ ensures  state == 2;
  void sendAddToS(int x, int y) {
    state = 2; ... }
  //@ context  Perm(state, write);
  //@ requires state == 1;
  //@ ensures  state == 3;
  void sendByeToS() {
    state = 3; ... }

  //@ context  Perm(state, write);
  //@ requires state == 2;
  //@ ensures  state == 1;
  int recvResFromS() {
    state = 1; ... } }
```

Fig. 8: Java API for Client in Adder (BGJ-style)

2 Usage: BGJ in a Nutshell

BGJ follows the same workflow as in Figure 5. We explain the steps below.

Steps 1-3: global types; local types; DFAs. First, the programmer manually writes a global type in Scribble notation (e.g., Figure 3). Next, BGJ automatically projects the global type to local types, and it automatically interprets the local types as DFAs. This is standard and as usual [16, 17].

Step 4: APIs. Next, BGJ automatically encodes the DFAs as APIs. Our approach is to encode a DFA of n states as an API of *a single* class instead of n classes (Figure 6). At run-time, only one instance of this class is created ("near-zero cost abstraction"); this instance allows any number of usages (method calls). To be able to check that these usages are proper, a key novelty of our approach is that BGJ also generates annotations for *method contracts*, Hoare-logic-style.

Example 3. Figure 8 shows the Java API for Client in Adder (Example 1), generated using BGJ (cf. Figure 6). Field `state` of class `DFA` identifies the current state; the methods of class `DFA` correspond to transitions. The annotations ("`//@ ...`") define for each method: a *precondition* ("`requires`"; what must be true before a call?), a *postcondition* ("`ensures`"; what will be true after?), and a *method invariant* ("`context`"; read/write permissions for which fields are needed?). □

Step 5: processes. Last, the programmer manually writes processes using the APIs and automatically verifies proper usage with VerCors (i.e., methods are called only if the preconditions hold). These checks are purely static. If successful, safety relative to the global type and liveness (communication deadlock freedom) are assured; else, a bug is found ("all errors are reported early at compile-time").

```java
//@ context  Perm(a.state, write)
//@ requires a.state == 1;
//@ ensures  a.state == 3;
void client(DFA a) {
  int x = 1; int y = 2;
  //@ loop_invariant a.state == 1;
  while (x + y < 100) {
    a.sendAddToS(x, y);
    x = y; y = a.recvResFromS(); }
  a.sendByeToS(); }
```

Fig. 9: Process for Client in Adder

Example 4. Figure 9 shows a process for Client in Adder (Example 1), using the Java API in Figure 8. It resembles Figure 7, except that method `client` and the loop are annotated with a simple contract and invariant. Using VerCors, we can verify that the methods are called only if the preconditions hold. Conversely, if we duplicate line 8, then VerCors reports an error: consecutively sending two `Add`-messages is forbidden. This can be detected only dynamically in Figure 7 (i.e., a `RuntimeException` would be thrown in `UseOnce` of Figure 6). □

3 Implementation

BGJ is implemented in Java. It reuses the front-end of Scribble for global types, local types, and DFAs in steps 1-3 and, thus, supports the same features (including input branching). The encoder of DFAs as APIs in step 4 is new. It generates two versions of every API: concrete (e.g., Figure 8) and abstract (e.g., Figure 8 without "`...`"). The concrete API is for running a process. The abstract API, which omits all verification-irrelevant details, is for verifying a process.[3] At run-time, TCP is used to transport messages between processes.

Besides the APIs, BGJ also generates "skeletons" of process code. These skeletons represent the basic control flow (adapted from the DFAs) with `send...` and `recv...` method calls in the right places (guaranteed to pass verification). The skeletons can subsequently be filled in with the actual computations.

4 Preliminary Evaluation

We obtained first practical experience with BGJ to study its two improvements. Regarding "all errors are reported early at compile-time", we investigated how much time the verification step of VerCors takes for eight example protocols in Scribble's repository [13]. Figure 10 shows the results, averaged over thirty runs, using generated skeletons as process code. A preliminary conclusion is that the extra time can be low enough (worth the effort[4]) for our approach to be feasible.

Regarding "near-zero cost abstractions at run-time", we investigated run-time overhead of a Scribble-based process (e.g., Figure 6) vs. a BGJ-based process (e.g., Figure 8) for Client in Adder. We factored out code common to both versions (e.g., actual transport of messages over the wire), to be able to specifically measure the impact of the *differences* (methodology of Castro et al. [5]). Averaged over thirty runs, the Scribble-based process and the BGJ-based process

[3] The generated annotations are compatible with VerCors 1.0 and above; VerCors can be used as-is. A limitation of our approach is that VerCors supports only a subset of Java. This affects the set of Java features supported for processes.

[4] Usage of BGJ requires two kinds of effort. First, a method in hand-written process code needs to be annotated if the body uses a generated API. All the other code—typically the vast majority of the program (e.g., business logic, database access)—can be tagged to be skipped by VerCors. The few annotations to be added, are only about the state of the DFA at the beginning/ending of a method (pre/postconditions), or at the beginning of each iteration (loop invariants). This is similar to the effort of manually tracking state types when using the existing Scribble. Second, the validity of the annotations need to be checked by VerCors. This is fully automated.

protocol	#roles	time	$\frac{\text{time}}{\text{\#roles}}$
Adder	2	15.1	7.6
Booking	3	24.3	8.1
BuyerBrokerSupplier	4	30.4	7.6
Fibonacci	2	14.9	7.5
HTTP	2	40.0	20.0
Negotiate	2	17.2	8.6
SMTP	2	24.7	12.4
TwoBuyer	3	22.8	7.6

Fig. 10: Time of VerCors (in seconds)

completed 2^{31} (`Integer.MAX_VALUE`) iterations in 5221ms and 974ms, respectively. Our preliminary conclusion is that our approach is indeed more resource-efficient.

5 Conclusion

Related work. The combination of the MPST method and deductive verification is largely unexplored territory. The only other work, by López et al. [24], uses deductive verifier VCC [7] to statically check safety and liveness of C+MPI protocol implementations relative to MPST-based specifications. Their approach is very different from ours, though, as it is not based on API generation.

The approach of encoding DFAs of n states as APIs of a single class was recently studied by Cledou et al. [6], by leveraging advanced features of the type system of Scala 3. Their approach does not address the issues in Section 1, though, whereas our approach does. Previous attempts to address the issue of "mixed static/dynamic checks" either target a programming language with a stronger type system (Rust) [8, 9, 22, 23], or adopt callback-style APIs in the specific context of event-based programming [34, 35]. In contrast, our approach does not rely on (the strength of) the type system of the targeted programming language, and it supports traditional procedural/object-oriented programming.

Closest to BGJ is StMungo [21]: the approaches of both tools are similar, but the underlying static analysis techniques differ. BGJ leverages method contracts and deductive verification, while StMungo is based on *typestate* [33]. A key advantage of using deductive verification is that it immediately opens the door to reasoning about functional correctness (next paragraph).

Future work. There are two next steps. First, now that we have the infrastructure to combine the MPST method and deductive verification, we are keen to explore their further integration to reason about *functional correctness* of distributed systems. VerCors is based on concurrent separation logic [4, 29], so key capabilities to reason about concurrency are already in place. This is connected to work in which separation logic is used to control I/O operations (e.g., Penninckx et al. [31]). Second, while the usage of deductive verification is central to BGJ, our approach does not crucially depend on VerCors: we chose it because it is a fully automated, well-supported deductive verifier for Java, but other tools (e.g., KeY [1], VeriFast [19]) offer opportunities worth investigating, too.

Data Availability Statement

The artifact is available on Zenodo [3]. It contains: (a) our tool and its dependencies; (b) material to replicate the example in Section 2; (c) material to replicate the experiments in Section 4.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016)
2. Blom, S., Huisman, M.: The vercors tool for verification of concurrent programs. In: FM. Lecture Notes in Computer Science, vol. 8442, pp. 127–131. Springer (2014)
3. Bouma, J., de Gouw, S., Jongmans, S.: Multiparty session typing in java, deductively (artifact) (2023). `https://doi.org/10.5281/zenodo.7559175`
4. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1-3), 227–270 (2007)
5. 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)
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. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009)
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. Lecture Notes in Computer Science, 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). Lecture Notes in Computer Science, vol. 7966, pp. 174–186. Springer (2013)
12. GitHub, Inc: scribble-java/adder.scr at 02dbf9abd9993b17c809aa610311452ec4c763bc · scribble/scribble-java, accessed 22 January 2023, `https://github.com/scribble/scribble-java/blob/02dbf9abd9993b17c809aa610311452ec4c763bc/scribble-demos/scrib/tutorial/src/tutorial/adder/Adder.scr`
13. GitHub, Inc: scribble-java/scribble-demos/scrib at ccb0e48d69c6e3088e746138099c3183ca1ac79b · scribble/scribble-java, accessed 22 January 2023, `https://github.com/scribble/scribble-java/tree/ccb0e48d69c6e3088e746138099c3183ca1ac79b/scribble-demos/scrib`
14. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. pp. 273–284. ACM (2008)

15. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016)
16. 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)
17. 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)
18. Huisman, M., Monti, R.E.: On the industrial application of critical software verification with vercors. In: ISoLA (3). Lecture Notes in Computer Science, vol. 12478, pp. 273–292. Springer (2020)
19. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and java. In: NASA Formal Methods. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011)
20. 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)
21. 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)
22. 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)
23. 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)
24. López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-based verification of message-passing parallel programs. In: OOPSLA. pp. 280–298. ACM (2015)
25. 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)
26. Monti, R.E., Rubbens, R., Huisman, M.: On deductive verification of an industrial concurrent software component with vercors. In: ISoLA (1). Lecture Notes in Computer Science, vol. 13701, pp. 517–534. Springer (2022)
27. 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)
28. 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)
29. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. **375**(1-3), 271–307 (2007)
30. Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: IFM. Lecture Notes in Computer Science, vol. 11918, pp. 418–436. Springer (2019)
31. Penninckx, W., Jacobs, B., Piessens, F.: Sound, modular and compositional verification of the input/output behavior of programs. In: ESOP. Lecture Notes in Computer Science, vol. 9032, pp. 158–182. Springer (2015)
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. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. **12**(1), 157–171 (1986)
34. 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)
35. 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)