Oven: Safe and Live Communication Protocols in Scala, using Synthetic Behavioural Type Analysis

Francisco Ferreira
Royal Holloway, University of London
London, UK

Sung-Shik Jongmans
Open University of the Netherlands
Heerlen, the Netherlands
Centrum Wiskunde & Informatica (CWI), NWO-I
Amsterdam, the Netherlands

ABSTRACT

We present Oven: a toolset to assure safety and liveness of communication protocols among threads in concurrent programs in Scala.

Oven is the first practical toolset that is built on top of new theoretical foundations of synthetic behavioural type analysis, recently developed by us to improve the expressiveness of existing work.

We explain Oven's usage, summarise its design and implementation (main challenge: how to encode the new synthetic behavioural typing rules in Scala's existing type system), and discuss a preliminary evaluation of expressiveness (the results provide first evidence that Oven is an improvement over two state-of-the-art tools).

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**.

KEYWORDS

behavioural types, multiparty session types, choreographies

ACM Reference Format:
Francisco Ferreira and Sung-Shik Jongmans. 2023. Oven: Safe and Live Communication Protocols in Scala, using Synthetic Behavioural Type Analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3597926.3604926

1 INTRODUCTION

Background. To take advantage of multi-core processors, concurrent programming with shared memory—a notoriously difficult enterprise—has become increasingly important. In the wake of this development, programming languages have started to offer core support for high-level *communication primitives*, in the form of message passing through channels (e.g., Go, Rust, Clojure), in addition to lower-level *synchronisation primitives*. The idea is that channels can serve as a programming abstraction for shared memory.

Supposedly, the usage of channels is less prone to concurrency bugs than the usage of locks, semaphores, etc. However, evidence suggests otherwise. For instance, Tu et al. [24] find that "message passing [using channels] does not necessarily make multi-threaded programs less error-prone than shared memory [using locks]".

One of the challenges of message passing is how to assure that the implementation of *communication protocols* is safe and live relative to their specification. Safety means that "bad" communications never happen: if a communication happens in the implementation, then it is allowed to happen by the specification. Liveness means that "good" communications eventually happen.

Multiparty session typing (MPST) [9] is a method to automatically assure safety and liveness of protocol implementations relative to protocol specifications. The idea is to specify protocols as *behavioural types* [2, 12] against which threads are type-checked; the method ensures that static well-typedness at compile-time implies dynamic safety and liveness at execution-time. Over the past 10–15 years, substantial progress has been made, including the development of practical tools for many programming languages (e.g., F# [21], F* [26], Go [6], Java [10, 11], OCaml [13], Rust [17, 18], Scala [7, 22], and TypeScript [20]).

This paper. The problem with MPST is *expressiveness*: many desirable protocols cannot be specified and implemented using MPST, as they are conservatively rejected by the existing behavioural type analysis techniques. To improve expressiveness, the MPST community has been working extensively on making the techniques more liberal (e.g., [3–5, 19, 25]). As part of these research efforts, we recently developed new theoretical foundations of *synthetic* behavioural type analysis [14]. In this companion paper, we present the first practical toolset that is built on top of that: Oven.

In §2, we explain Oven's usage. In §3, we summarise its design and implementation. In §4, we discuss a preliminary evaluation of expressiveness. The envisioned users of Oven are programmers that use message passing through channels as a programming abstraction for shared memory. The current version of Oven has been archived in a virtual machine [15], while the source code is presently hosted at https://github.com/nuscr/oven.

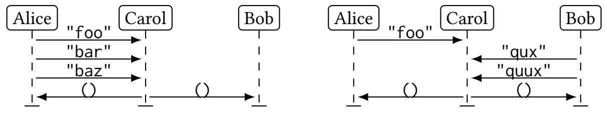

Figure 1: Two possible executions of the logging protocol

2 USAGE OF THE Oven TOOLSET

To explain the usage of Oven, we will develop a simple concurrent program that consists of three threads, say, Alice, Bob, and Carol.

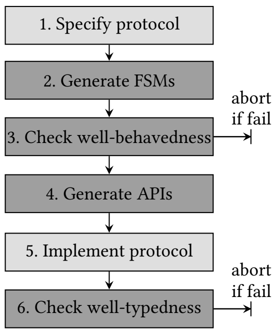

Figure 2: Usage of `Oven`

```
 1 global Logging(role A,
 2                role B,
 3                role C) {
 4   fin {
 5     choice {
 6       String from A to C;
 7     } or {
 8       String from B to C;
 9   } }
10   par {
11     Unit from C to A;
12   } and {
13     Unit from C to B;
14 } }
```

Figure 3: Specification of the logging protocol

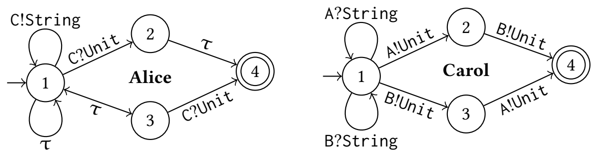

Figure 4: FSMs for Alice and Carol in the logging protocol

The threads engage in a *logging protocol*: Alice and Bob independently tell Carol to log messages (by sending strings), until Carol tells Alice and Bob that the log is full (by sending units). Figure 1 visualises two possible executions as sequence diagrams.

Figure 2 visualises the main steps of using `Oven`; light grey boxes indicate manual work by the programmer; dark gray boxes indicate automatic work by `Oven`. We explain and exemplify the steps.

Step 1: Specify protocol. First, the programmer specifies the protocol in the `Oven` language; it is an extension of the *Scribble* language [8], which is a DSL for protocols that is used in most MPST tools.

Figure 3 shows the `Oven` code for the logging protocol. Lines 1-3 defines the name of the protocol (`Logging`) and the names of the *roles* that the threads enact (`A`, `B`, `C`). Lines 4–14 define the body of the protocol: lines 4–9 define that, iteratively (`fin`), a string is communicated either from Alice, or from Bob, to Carol (`choice`); lines 10–14 define that a unit is communicated from Carol to both Alice and Bob in unspecified order (`par`). Intuitively, the specification in Figure 3 is a textual version of the sequence diagrams in Figure 1.

Step 2: Generate finite state machines. Next, `Oven` consumes the protocol specification as input and produces thread-specific *finite state machines* (FSM) as output. Each FSM models the communication behaviour of a single thread from that thread's perspective.

Figure 4 shows the FSMs for Alice and Carol in the logging protocol; Bob's FSM is similar to Alice's. Each state models a control point in the protocol; each transition models an action by a thread (!/?) or by the environment of a thread (τ). For instance, in Alice's FSM, the `C!String`-transition models the send of a string to Carol by Alice, while in Carol's FSM, the `A?String`-transition models the corresponding receive. The τ-transitions in Alice's FSM model actions by Bob and Carol that are unobservable to Alice. Intuitively, each FSM in Figure 4 is a formal version of a lifeline in Figure 1.

```
 1 class S1:
 2   def loop(f: (S1, S1 => S4) => S4): S4 = ...
 3   def select[L1 <: (Null | (Unit, S2 => S4)),
 4              L2 <: (Null | (Unit, S3 => S4))]
 5             (sendUnitToA: L1 = null,
 6              sendUnitToB: L2 = null,
 7              recvStringFromA: (String, S1) => S4,
 8              recvStringFromB: (String, S1) => S4)
 9             (implicit ev: (L1 =:= (Unit, S2 => S4))
10                         | (L2 =:= (Unit, S3 => S4)))
11             : S4 = ...
12 class S2:
13   def select[L1 <: (Null | (Unit, S4 => S4))]
14             (sendUnitToB: L1 = null)
15             (implicit ev: (L1 =:= (Unit, S4 => S4)))
16             : S4 = ...
17 class S3:
18   def select[L1 <: (Null | (Unit, S4 => S4))]
19             (sendUnitToA: L1 = null)
20             (implicit ev: (L1 =:= (Unit, S4 => S4)))
21             : S4 = ...
22 class S4 // empty
```

Figure 5: API for the FSM for Carol in the logging protocol

Step 3: Check well-behavedness. Next, `Oven` consumes each FSM as input and produces an accept/reject verdict as output. If the FSM is accepted, it is *well-behaved* [14]: this is a sufficient condition to assure that well-typedness implies safety and liveness (step 6). If the FSM is rejected, `Oven` produces an error message to indicate why; in this case, the protocol as specified is not supported by `Oven`.

Informally, an FSM is well-behaved when these conditions hold:

(1) *external actions are neutral:* the same transitions are possible both before and after τ-transitions;
(2) *sending is causal:* a !-transition is possible initially, or after a non-τ-transition, or both before and after a τ-transition;
(3) *receiving is deterministic:* ?-transitions from the same source state to different target states have different labels.

The FSMs in Figure 4 are well-behaved. To exemplify condition 1, in the Alice's FSM, a `C?Unit`-transition is possible both before a τ-transition in state 1 and after a τ-transition in state 3. To exemplify condition 2, in Carol's FSM, each !-transition is possible initially or after a non-τ-transition. To exemplify condition 3, vacuously, there are no ?-transitions from the same source to different targets.

Step 4: Generate APIs. Next, `Oven` consumes each FSM as input, determinises it (modulo τ-transitions), and produces a thread-specific API in Scala. Roughly, every state in the FSM is encoded as a class in the API; every set of transitions out of the same state is encoded as a method. The idea is that if the usage of the API is well-typed at compile-time, then the corresponding thread will faithfully simulate a run of the corresponding FSM at execution-time.

Figure 5 shows (an excerpt of) the API generated for Carol's FSM in the logging protocol. Classes `S1`, `S2`, `S3`, and `S4` encode states 1, 2, 3, and 4 in Carol's FSM in Figure 4; each method `select` of class `S`*i* encodes the transitions out of state *i*. Furthermore, method `loop` of class `S1` allows the programmer to implement loops of state 1.

Each method `select` has one or more *named arguments* (lines 5–8, line 14, and line 19), some of which also have a *default value* of `null`. If `select` is called, then the instantiation of named arguments indicates which transitions the thread is willing to make. For instance, if `S1.select` is called and `sendUnitToA`, `recvStringFromA`, and `recvStringFromB` are instantiated, then Carol is willing to send

```
1  def carolGood(s: S1): S4 =
2    s.loop((s, recur) => s.select(
3      sendUnitToA = ((), s => s.select(
4        sendUnitToB = ((), s => s))),
5      sendUnitToB = ((), s => s.select(
6        sendUnitToA = ((), s => s))),
7      recvStringFromA = (_, s) => recur(s),
8      recvStringFromB = (_, s) => recur(s)))
9
10 def carolBad1(s: S1): S4 =
11   s.loop((s, recur) => s.select(
12     recvStringFromA = (_, s) => recur(s),
13     recvStringFromB = (_, s) => recur(s)))
14
15 def carolBad2(s: S1): S4 =
16   s.loop((s, recur) => s.select(
17     sendUnitToA = ((), s => s.select(
18       sendUnitToB = ((), s => s))),
19     sendUnitToB = ((), s => s.select(
20       sendUnitToA = ((), s => s))),
21     recvStringFromB = (_, s) => recur(s)))
22
23 def carolBad3(s: S1): S4 =
24   s.loop((s, recur) => s.select(
25     sendUnitToA = ((), s => recur(s)),
26     sendUnitToB = ((), s => s.select(
27       sendUnitToA = ((), s => s))),
28     recvStringFromA = (_, s) => recur(s),
29     recvStringFromB = (_, s) => recur(s)))
30
31 def carolBad4(s: S1): S4 =
32   s.loop((s, recur) => s.select(
33     sendUnitToA = ((), s => s.select(
34       sendUnitToB = ((), s => s))),
35     sendUnitToB = ((), s => s.select(
36       sendUnitToA = ((), s => s))),
37     recvStringFromA = (_, s) => s,
38     recvStringFromB = (_, s) => recur(s)))
```

Figure 6: Implementations of Carol in the logging protocol

to Alice, receive from Alice, or receive from Bob, *but not send to Bob*; the body of `S1.select` non-deterministically chooses one of these alternatives, depending also on the environment (e.g., a receive can happen only if the corresponding send has happened).

Named arguments that encode !-transitions are pairs that consist of a value to send and a callback function (input: a successor state object; output: a final state object). Symmetrically, named arguments that encode ?-transitions are callback functions (input: a received value and a successor state object; output: a final state object).

Step 5: Implement protocol. Next, the programmer implements the protocol in Scala, using the generated APIs.

Figure 6 shows Scala code for five versions of Carol in the logging protocol, using the API from Figure 5. Each version is implemented as a function that consumes an initial state object (of type `S1`) and produces a final state object (of type `S4`). To actually spawn Carol, a thread can be created that calls one of the five functions.

In function `carolGood`, all named arguments are instantiated when `select` is called (lines 3, 5, 7, and 8). Thus, in this version, Carol is willing to make any transition out of state 1 in Carol's FSM in Figure 4. Conversely, in function `carolBad1`, only some named arguments are instantiated (lines 12 and 13). Thus, in this version, Carol is willing to make only any ?-transition.

Step 6: Check well-typedness. Last, the Scala compiler consumes the protocol implementation as input and produces an accept/reject verdict as output. If the protocol implementation is accepted, it is *well-typed*: combined with the well-behavedness of the FSMs (step 3), well-typedness implies safety and liveness, modulo non-linear usage of API objects (checked dynamically) and non-terminating/exceptional behaviour (beyond the scope); these caveats are standard when encoding session typing in a mainstream programming language [6, 10, 16, 20–22]. If the protocol implementation is rejected, the Scala compiler produces an error message; in this case, the protocol as implemented is not supported.

Informally, a protocol implementation is well-typed when these conditions hold for each call of method `select`:

(1) *partially output-enabled:* if there are ≥1 named arguments for !-transitions, then at least one of them is instantiated;
(2) *totally input-enabled:* each named argument for a ?-transition is instantiated.

Thus, a thread must implement each receive as specified in its FSM (to be able to process any input provided by its environment), but it may implement only some sends. Condition 1 is checked using *type parameters* with *union types* (e.g., lines 3–4 in Figure 5), and using *implicit parameters* with *type constraints* (e.g., lines 9–10 in Figure 5): the former allows to express that instantiation of named arguments is optional, by providing a default value of `null` (e.g., lines 5–6); the latter allows to express that, nevertheless, at least one named argument with a default value needs to be instantiated.

Function `carolGood` in Figure 6 is well-typed (i.e., it satisfies the two conditions). In contrast, the remaining functions in Figure 6 are ill-typed: `carolBad1` violates condition 1; `carolBad2` violates condition 2; `carolBad3` calls `recur` with a state object of the wrong type (`S2` instead of `S1`; line 25), indicating an attempt to re-enter the loop in the wrong state of the FSM; `carolBad4` returns a state object of the wrong type (`S1` instead of `S4`; line 37), indicating an attempt to exit the loop in the wrong state of the FSM.

3 DESIGN AND IMPLEMENTATION

The `Oven` toolset consists of two separate tools:

- `OvenMPST` is a programming language-generic front-end for steps 1–3 in Figure 2. It is written in OCaml, but via OCaml-to-Javascript compilation, it runs with a GUI in the browser, including visualisation of FSMs (useful to diagnose well-behavedness errors).
  To support step 1, `OvenMPST` applies standard parsing techniques from concrete syntax to abstract syntax. To support step 2, `OvenMPST` runs an interpreter on the abstract syntax to build the state space. The abstract syntax and the interpreter of `OvenMPST` essentially implement the formal grammar and transition rules (operational semantics) of a core calculus of the `Oven` language [14]. To support step 3, `OvenMPST` checks the three well-behavedness conditions; it implements the formal definitions of these conditions, including an algorithm to equate states modulo *weak bisimilarity* [1].
- `OvenCG` is a Scala-specific back-end for step 4. It is written in Java and runs with a CLI in the terminal. `OvenCG` encodes FSMs as classes in Scala, as exemplified in §2. Internally, a lightweight run-time library uses channels to transport messages between threads via shared memory. To encode the advanced typing rules from new theoretical foundations [14], we crucially leverage advanced features of Scala's type

Table 1: Preliminary evaluation

Protocol	1	2	3	mpstk [23]	mpstpp [16]	Oven
Logging [§2]	✓	✗	✓	–	lin.	lin.
Summation [14]	✓	✗	✗	exp.	–	lin.
Recursive Two Buyer [14]	✓	✗	✗	exp.	–	lin.
Binomial (data types) [14]	✗	✗	✗	exp.	lin.	lin.
Binomial (threads) [14]	✗	✗	✓	–	lin.	lin.
Acq.-Use-Rel. [14]	✓	✓	✓	–	–	lin.
Unfair Acq.-Use-Rel. [14]	✓	✓	✓	–	–	lin.
Example 9 [25]	✓	✗	✗	exp.	–	lin.
Example 13 [25]	✗	✗	✓	–	lin.	lin.
Example 15 [25]	✗	✗	✓	–	–	lin.

Communication patterns:
1: Different threads participate in different branches
2: A receiver choose a sender of the first communication
3: ≥2 threads synchronously choose a sender and receiver of a next communication

system (the combination of type parameters, union types, implicit parameters, type constraints).

The advantage of the separation between OvenMPST and OvenCG is that back-ends for other implementation languages can be developed independent of the front-end. The interface between OvenMPST and OvenCG is the existing *DOT* language to represent graphs.

4 PRELIMINARY EVALUATION AND FUTURE

The aim of developing Oven was to provide evidence that our new theoretical foundations of synthetic behavioural type analysis [14]: (1) can form the basis of a practical toolset; (2) increase expressiveness relative to existing MPST tools. Point 1 is evidenced by Oven's existence. To substantiate point 2, we applied Oven to a variety of protocols (selected to cover a variety of combinations of features), summarised in Table 1; Examples 9/13/15 constitute basic multiparty buyer–seller scenarios. Columns "1", "2", and "3" indicate communication patterns required in the protocols. Columns "mpstk [23]", "mpstpp [16]", and "Oven" indicate non-support ("–"), support using exponential-time analysis in the number of threads ("exp."), or support using linear-time analysis ("lin.").

Oven supports all protocols under study using linear-time analysis. In contrast, mpstk supports fewer protocols, and it uses exponential-time analysis to support the remaining ones. Similarly, mpstpp supports fewer protocols, but unlike mpstk, it uses linear-time analysis to support the remaining ones. We conclude that Oven indeed increases expressiveness relative to mpstk and mpstpp.

In future work, we aim to extend the evaluation by studying a larger set of protocols and conduct bigger real(istic) case studies. Although Oven targets shared-memory concurrent programming, to evaluate expressiveness, a very interesting source of case studies is *distributed algorithms*. For instance, we believe that Oven might be the first MPST tool that supports the Paxos protocol to solve consensus, if we assume a statically fixed number of threads and synchronous communication; extensions to a dynamically flexible number of threads and asynchronous communication are also very interesting, but they first require significant theoretical advances. Another direction for future work is to extend Oven with additional back-ends for other programming languages, beyond Scala. Due to its powerful type system, Rust would be an interesting candidate.

ACKNOWLEDGMENTS

Funded by the Netherlands Organisation of Scientific Research (NWO): 016.Veni.192.103.

REFERENCES

[1] Luca Aceto, Anna Ingólfsdóttir, and Jirí Srba. 2012. The algorithmics of bisimilarity. In *Advanced Topics in Bisimulation and Coinduction*. Cambridge tracts in theoretical computer science, Vol. 52.
[2] Davide Ancona et al. 2016. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages* 3, 2-3 (2016).
[3] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2019. Reversible sessions with flexible choices. *Acta Informatica* 56, 7-8 (2019).
[4] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. 2022. Asynchronous Sessions with Input Races. *CoRR* abs/2203.12876 (2022).
[5] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Ross Horne. 2020. Global types with internal delegation. *Theor. Comput. Sci.* 807 (2020).
[6] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *PACMPL* 3, POPL (2019).
[7] Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. 2022. API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3. In *ECOOP (LIPIcs, Vol. 222)*.
[8] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. 2011. Scribbling Interactions with a Formal Foundation. In *ICDCIT (LNCS, Vol. 6536)*.
[9] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*.
[10] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *FASE (LNCS, Vol. 9633)*.
[11] Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE (LNCS, Vol. 10202)*.
[12] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016).
[13] Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. 2020. Multiparty Session Programming With Global Protocol Combinators. In *ECOOP (LIPIcs, Vol. 166)*.
[14] Sung-Shik Jongmans and Francisco Ferreira. 2023. Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types. In *ECOOP (LIPIcs, Vol. 263)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:29.
[15] Sung-Shik Jongmans and Francisco Ferreira. 2023. Synthetic Behavioural Typing: Sound, Regular Multiparty Sessions via Implicit Local Types (Artifact). *Dagstuhl Artifacts Ser.* 9, 2 (2023).
[16] Sung-Shik Jongmans and N. Yoshida. 2020. Exploring Type-Level Bisimilarity towards More Expressive Multiparty Session Types. In *ESOP (LNCS, Vol. 12075)*.
[17] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2020. Implementing Multiparty Session Types in Rust. In *COORDINATION (LNCS, Vol. 12134)*.
[18] Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. 2022. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In *ECOOP (LIPIcs, Vol. 222)*.
[19] Rupak Majumdar, Madhavan Mukund, Felix Stutz, and Damien Zufferey. 2021. Generalising Projection in Asynchronous Multiparty Session Types. In *CONCUR (LIPIcs, Vol. 203)*.
[20] Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. 2021. Communication-safe web programming in TypeScript with routed multiparty session types. In *CC*.
[21] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A session type provider: compile-time API generation of distributed protocols with refinements in F#. In *CC*.
[22] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP (LIPIcs, Vol. 74)*.
[23] Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* 3, POPL (2019).
[24] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *ASPLOS*.
[25] Rob van Glabbeek, Peter Höfner, and Ross Horne. 2021. Assuming Just Enough Fairness to make Session Types Complete for Lock-freedom. In *LICS*.
[26] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically verified refinements for multiparty protocols. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).

Received 2023-05-18; accepted 2023-06-08