`Kmclib`: Automated Inference and Verification of Session Types from OCaml Programs

Keigo Imai[1] ✉, Julien Lange[2], and Rumyana Neykova[3]

[1] Gifu University, Gifu, Japan, keigoi@gifu-u.ac.jp
[2] Royal Holloway, University of London, UK, julien.lange@rhul.ac.uk
[3] Brunel University London, UK, rumyana.neykova@brunel.ac.uk

Abstract. Theories and tools based on multiparty session types offer correctness guarantees for concurrent programs that communicate using message-passing. These guarantees usually come at the cost of an intrinsically top-down approach, which requires the communication behaviour of the entire program to be specified as a global type.
This paper introduces `kmclib`: an OCaml library that supports the development of *correct* message-passing programs without having to write any types. The library utilises the meta-programming facilities of OCaml to automatically infer the session types of concurrent programs and verify their compatibility (k-MC [15]). Well-typed programs, written with `kmclib`, do not lead to communication errors and cannot get stuck.

Keywords: Multiparty Session Types · Concurrent Programming · OCaml

1 Introduction

Multiparty session types (MPST) [5] are a popular type-driven technique to ensure the correctness of concurrent programs that communicate using message-passing. The key benefit of MPST is to guarantee statically that the components of a program have compatible behaviours, and thus no components can get permanently stuck. Many implementations of MPST in different programming languages have been proposed in the last decade [2,4,6,10,12,16–18,20,23], however, all suffer from a notable shortcoming: they require programmers to adopt a top-down approach that does not fit well in modern development practices. When changes are frequent and continual (e.g., continuous delivery), re-designing the program and its specification at every change is not feasible.

Most MPST theories and tools advocate an intrinsically top-down approach. They require programmers to specify the communication (often in the form of a global type) of their programs before they can be type-checked. In practice, type-checking programs against session types is very difficult. To circumvent the problem, most implementations of MPST rely on *external* toolings that generate code from a global type, see e.g., all works based on the Scribble toolchain [22].

In this paper, we present an OCaml library, called `kmclib` [8,9], which supports the development of programs that enjoy all the benefits of MPST while avoiding their main drawbacks. The `kmclib` library guarantees that threads in

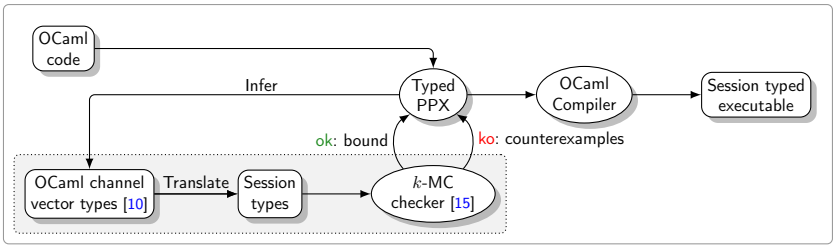

Fig. 1: Workflow of the `kmclib` library (the PPX plugin is the shaded box).

well-typed programs will not get stuck. The library also enables *bottom-up development*: programmers write message-passing programs in a natural way, without having to write session types. Our library is built on top of *Multicore OCaml* [21] that offers highly scalable and efficient concurrent programming, but does not provide any static guarantees wrt. concurrency.

Figure 1 gives an overview of `kmclib`. Its implementation combines the power of the *type-aware* macro system of OCaml (Typed PPX) with two recent advances in the session types area: an encoding of MPST in OCaml (channel vector types [10]) and a session type compatibility checker (k-MC checker [15]). To our knowledge, this is the first implementation of type inference for MPST and the first integration of compatibility checking in a programming language.

The `kmclib` library [8,9] offers several advantages compared to earlier MPST implementations. **(1)** It is *flexible*: programmers can implement communication patterns (e.g., fire-and-forget patterns [15]) that are not expressible in the synchrony-oriented syntax of global types. **(2)** It is *lightweight* as it piggybacks on OCaml's type system to check and infer session types, hence lifting the burden of writing session types off the programmers. **(3)** It is *user-friendly* thanks to its integration in Visual Studio Code, where compatibility violations are mapped to precise locations in the code. **(4)** It is *well-integrated* into the natural edit-compile-run cycle. Although compatibility is checked by an external tool, this step is embedded as a compilation step and thus hidden from the user.

2 Safe Concurrent Programming in Multicore OCaml

We give an overview of the features and usage of `kmclib` using the program in Figure 2 (top) which calculates Fibonacci numbers. The program consists of three concurrent *threads* (`user`, `master`, and `worker`) that interact using point-to-point message-passing. Initially, the `user` thread sends a request to the `master` to start the calculation, then waits for the `master` to return a work-in-progress message, or the final result. After receiving the result, the `user` sends back a stop message. Upon receiving a new request, the `master` splits the initial computation in two and sends two tasks to a `worker`. For each task that the `worker` receives, it replies with a result. The `master` and `worker` threads are recursive and terminate only upon receiving a stop message.

```
let KMC (uch,mch,wch) = [%kmc.gen (u,m,w)]

let user () =
  let uch = send uch#m#compute 42 in
  let rec loop uch : unit =
    match receive uch#m with
    | `wip(res, uch) ->
      printf "in progress: %d\n" res;
      loop uch
    | `result(res, uch) ->
      printf "result: %d\n" res;
      send uch#m#stop ()
  in loop uch

let worker () =
  let rec loop wch : unit =
    match receive wch#m with
    | `task(num, wch) ->
      loop (send wch#m#result (fib num))
    | `stop((), wch) -> wch
  in loop wch
```

```
let master () =
  let rec loop (mch : [%kmc.check u]) : unit =
    match receive mch#u with
    | `compute(x, mch) ->
      let mch = send mch#w#task (x - 2) in
      let mch = send mch#w#task (x - 1) in
      let `result(r1, mch) = receive mch#w in
      let mch = send mch#u#wip r1 in
      let `result(r2, mch) = receive mch#w in
      loop (send mch#u#result (r1 + r2))
    | `stop((), mch) ->
      send mch#w#stop ()
  in loop mch

let () =
  let ut = Thread.create user () in
  let mt = Thread.create master () in
  let wt = Thread.create worker () in
  List.iter Thread.join [ut;mt;wt]
```

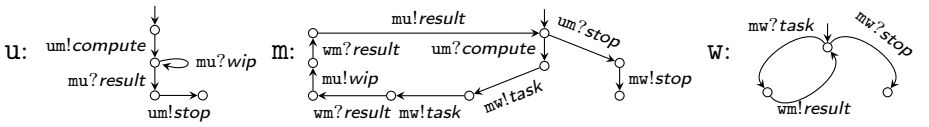

Fig. 2: Example of `kmclib` program (top) and *inferred* session types (bottom).

Figure 2 (bottom) gives a session type for each thread, i.e., the behaviour of each thread wrt. communication. For clarity we represent session types as a communicating finite state machines (CFSM [1]), where ! (resp. ?) denotes sending (resp. receiving). For example, `um`!*compute* means that the user is sending to the master a message *compute*, while `um`?*compute* says that the master receives *compute* from the user. Our library infers these CFSM representations from the OCaml code, in Figure 2 (top), and verifies *statically* that the three threads are *compatible*, hence no thread can get stuck due to communication errors. If compatibility cannot be guaranteed, the compiler reports the kind of violations (i.e., *progress* or *eventual reception* error) and their locations in the code. Figure 3 shows how such semantic errors are reported visually in Visual Studio Code.

Albeit simple, the common communication pattern used in Figure 2 cannot be expressed as a global type, and thus cannot be implemented in previous MPST implementations. Concretely, global types cannot express the intrinsically asynchronous interactions between the master and worker threads (i.e., the master may send a second task message, while the worker sends a result).

Programming with `kmclib`. To enable safe message-passing programs, `kmclib` provides two communication primitives, `send` and `receive`, and two primitives for channel creation (`KMC` and `%kmc.gen`). Our library supports all the features of traditional MPST implementations and have similar limitations (fixed number of participant, no delegation, etc). We only give a user-oriented description of these primitives here (see [7, §A] for an overview of their implementations).

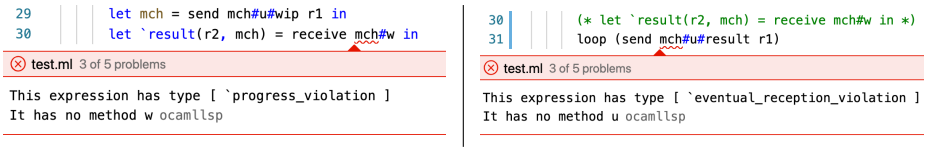

Fig. 3: Examples of type errors: progress (left) and eventual reception (right).

The crux of `kmclib` is the **session channel creation**: `[%kmc.gen (u,m,w)]` at Line 1. This primitive takes a tuple of *role names* as argument (i.e., `(u,m,w)`) and returns a tuple of communication channels, which are bound to `(uch,mch,wch)`. These channels will be used by the threads implementing roles `user` (Lines 3-13), `worker` (Lines 15-21), and `master` (Lines 22-34). Channels are implemented using concurrent queues from Multicore OCaml (`Domainslib.Chan.t`) but other underlying transports can easily be provided.

Threads send and receive messages over these channels using the communication primitives provided by `kmclib`. The send primitive requires three *arguments*: a channel, a destination role, and a message. For instance, the `user` sends a request to the `master` with `send uch#m#compute 20` where `uch` is the user's communication channel, `m` indicates the destination, and `compute 20` is the message (consisting of a label and a payload). Observe that a sending operation returns a new channel which is to be used in the continuation of the interactions, e.g., `uch` bound at Line 4, which must be used linearly (see [7] for details). Receiving messages works in a similar way to sending messages, e.g., see Line 6 where the `user` waits for a message from the `master` with `receive uch#m`. We use OCaml's pattern matching to match messages against their labels and bind the payload and continuation channel. See, e.g., Lines 7-10 where the `user` expects either a `wip` or `result` message. The receive primitive returns the payload `res` and a new communication channel `uch`.

New thread instances are spawned in the usual way; see Lines 36-39. The code at Line 40 waits for them to terminate.

Compatibility and error reporting. While the code in Figure 2 may appear unremarkable, it hides a substantial machinery that guarantees that, if a program type-checks, then its constituent threads are safe, i.e., no thread gets permanently stuck and all messages that are sent are eventually received. This property is ensured by `kmclib` using OCaml's type inference and PPX plugins to infer a session type from each thread then check whether these session types are *k-multiparty compatible* (*k*-MC) [15].

If a system of session types is *k*-MC, then it is safe [15, Theorem 1], i.e., it has the *progress* property (no role gets permanently stuck in a receiving state) and the *eventual reception* property (all sent messages are eventually received). Checking *k*-MC notably involves checking that all their executions (where each channel contains at most k messages) satisfy progress and eventual reception.

The *k*-MC-checker [15] performs a bounded verification to discover the *least* k for which a system is *k*-MC, up-to a specified upper bound N. In the `kmclib`

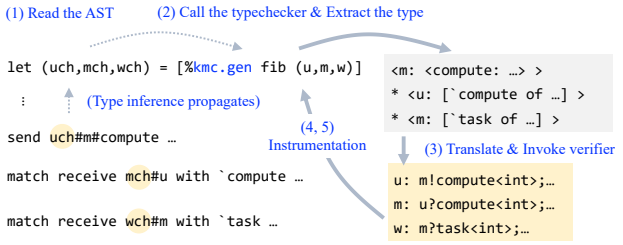

Fig. 4: Inference of session types from OCaml code.

API, this bound can be optionally specified with `[%kmclib.gen` *roles* `~bound:`N`]`. The k-MC-checker emits an error if the bound is insufficient to guarantee safety.

The `[%kmc.gen (u,m,w)]` primitive also feeds the results of k-MC checking back to the code. If the inferred session types are k-MC, then channels for roles `u`, `m` and `w` can be generated, otherwise a type error is raised. We have modified the k-MC-checker to return counterexample traces when the verification fails. This helps give actionable feedback to the programmer, as counterexample traces are translated to OCaml types and inserted at the hole corresponding to `[%kmc.gen]`. This has the effect of reporting the precise location of the errors.

To report errors in a function parameter, we provide an *optional* macro for types: `[%kmc.check rolename]` (see faded code in Line 23). Figure 3 shows examples of such error reports. The left-hand-side shows the reported error when Line 26 is commented out, i.e., the master sends one task, but expects two result messages; hence progress is violated since the master gets stuck at Line 30. The right-hand-side shows the reported error when Line 30 is commented out. In this case, variable `mch` in Line 31 (`master`) is highlighted because the `master` fails to consume a message from channel `mch`.

3 Inference of Session Types in `kmclib`

The `kmclib` API. The `kmclib` primitives allow the vanilla OCaml typechecker to infer the session structure of a program, while simultaneously providing a user-friendly communication API for the programmer. To enable inference of session types from concurrent programs, we leverage OCaml's structural typing and row polymorphism. In particular, we reuse the encoding from [10] where input and output session types are encoded as polymorphic variants and objects in OCaml. In contrast to [10] which relies on programmers writing global types prior to type-checking, `kmclib` infers and verifies local session types automatically, without requiring any additional type or annotation.

Typed PPX Rewriter. To extract and verify session types from a piece of OCaml code, the `kmclib` library makes use of OCaml PreProcessor eXtensions (PPX) plugins which provide a powerful meta-programming facility. PPX plugins are invoked during the compilation process to manipulate or translate the

abstract syntax tree (AST) of the program. This is often used to insert additional definitions, e.g., pretty-printers, at compile-time.

A key novelty of `kmclib` is the combination of PPX with a form of *type-aware translation*, whereas most PPX plugins typically perform purely syntactic (type-unaware) translations. Figure 4 shows the workflow of the PPX rewriter, overlayed on code snippets from Figure 2. The inference works as follows.

(1) The plugin reads the AST of the program code to replace the `[%kmc.gen]` primitive with a *hole*, which can have *any* type.
(2) The plugin invokes the *typechecker* to get the *typed* AST of the program. In this way, the type of the hole is *inferred* to be a tuple of *channel object types* whose structure is derived from their usages (i.e., `mch#u#compute`).
To enable this propagation, we introduce the idiom "`let (KMC ...) = ...`" which enforces the type of the hole to be *monomorphic*. Otherwise, the type would be too general and this would spoil the type propagation, see [7, § B].
(3) The inferred type is translated to a system of (local) *session types*, which are passed to the k-MC-checker.
(4) If the system is k-MC, then it is safe and the plugin *instruments* the code to allocate a fresh channel tuple (i.e., concurrent queues) at the hole.
(5) Otherwise, the k-MC-checker returns a *violation trace* which is translated back to an OCaml type and inserted at the hole, to report a more precise error.

The translation is limited inside the `[%kmc.gen]` expression, retaining a clear correspondence between the original and translated code. It can be understood as a form of *ad hoc polymorphism* reminiscent of type classes in Haskell. Like the Haskell typechecker verifies whether a type belongs to a class or not, the `kmclib` verifies whether the set of session types belongs to the class of k-MC systems.

4 Conclusion

We have developed a practical library for safe message-passing programming. The library enables developers to program and verify arbitrary communication patterns without the need for type annotations or user-operated external tools. Our *automated verification* approach can be applied to other general-purpose programming languages. Indeed it mainly relies on two ingredients: static structural typing and metaprogramming facilities. Both are available, with a varying degree of support, in, e.g., Scala, Haskell, TypeScript, and F#.

Our work is reminiscent of automated software model checking which has a long history (see [11] for a survey). There are few works on inference and verification of behavioural types, i.e., [3, 13, 14, 19]. However, Perera et al. [19] only present a prototype research language, while Lange et al. [3,13,14] propose verification procedures for Go programs that rely on external tools which are not integrated with the language nor its type system. To our knowledge, ours is the first implementation of type inference for MPST and the first integration of session types compatibility checking within a programming language.

Acknowledgements. This work is partially supported by KAKENHI 17K12662, 21K11827, 21H03415, and EPSRC EP/W007762/1.

References

1. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983). https://doi.org/10.1145/322374.322380
2. 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), https://dl.acm.org/citation.cfm?id=3290342
3. Dilley, N., Lange, J.: Automated Verification of Go Programs via Bounded Model Checking. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021. pp. 1016–1027. IEEE (2021). https://doi.org/10.1109/ASE51524.2021.9678571
4. Harvey, P., Fowler, S., Dardha, O., J. Gay, S.: Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming (ECOOP 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 194, p. 30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.12
5. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008. pp. 273–284 (2008). https://doi.org/10.1145/1328438.1328472
6. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE 2016. pp. 401–418 (2016). https://doi.org/10.1007/978-3-662-49665-7_24
7. Imai, K., Lange, J., Neykova, R.: kmclib: Automated inference and verification of session types (extended version). CoRR **abs/2111.12147** (2021), https://arxiv.org/abs/2111.12147
8. Imai, K., Lange, J., Neykova, R.: kmclib: A communication library with static guarantee on concurrency (2022), https://github.com/keigoi/kmclib
9. Imai, K., Lange, J., Neykova, R.: Kmclib: Artifact for the TACAS 2022 paper (2022). https://doi.org/10.5281/zenodo.5887544
10. Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference). LIPIcs, vol. 166, pp. 9:1–9:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.9
11. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**(4) (Oct 2009). https://doi.org/10.1145/1592434.1592438
12. Kouzapas, D., Dardha, O., Perera, R., Gay, S.J.: Typechecking protocols with Mungo and StMungo. In: PPDP 2016. pp. 146–159 (2016). https://doi.org/10.1145/2967973.2968595
13. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off Go: liveness and safety for channel-based programming. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 748–761. ACM (2017). https://doi.org/10.1145/3009837.3009847
14. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in Go using behavioural types. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. pp. 1137–1148. ACM (2018). https://doi.org/10.1145/3180155.3180157

15. Lange, J., Yoshida, N.: Verifying asynchronous interactions via communicating session automata. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 97–117. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_6
16. Miu, A., Ferreira, F., Yoshida, N., Zhou, F.: Generating Interactive WebSocket Applications in TypeScript. Electronic Proceedings in Theoretical Computer Science **314**, 12–22 (Apr 2020). https://doi.org/10.4204/EPTCS.314.2
17. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A Session Type Provider: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F♯. In: Dubach, C., Xue, J. (eds.) Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria. pp. 128–138. ACM (2018). https://doi.org/10.1145/3178372.3179495
18. Ng, N., de Figueiredo Coutinho, J.G., Yoshida, N.: Protocols by default - safe MPI code generation based on session types. In: CC 2015. pp. 212–232 (2015). https://doi.org/10.1007/978-3-662-46663-6_11
19. Perera, R., Lange, J., Gay, S.J.: Multiparty compatibility for concurrent objects. In: PLACES 2016. pp. 73–82 (2016). https://doi.org/10.4204/EPTCS.211.8
20. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP 2017. pp. 24:1–24:31 (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
21. Sivaramakrishnan, K.C., Dolan, S., White, L., Jaffer, S., Kelly, T., Sahoo, A., Parimala, S., Dhiman, A., Madhavapeddy, A.: Retrofitting parallelism onto ocaml. Proc. ACM Program. Lang. **4**(ICFP), 113:1–113:30 (2020). https://doi.org/10.1145/3408995
22. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The Scribble protocol language. In: Abadi, M., Lluch-Lafuente, A. (eds.) Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8358, pp. 22–41. Springer (2013). https://doi.org/10.1007/978-3-319-05119-2_3
23. 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). https://doi.org/10.1145/3428216