Discourje: Run-Time Verification of Communication Protocols in Clojure – Live at Last

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

Open University of the Netherlands, Heerlen, the Netherlands
ssj@ou.nl

Abstract. Multiparty session typing (MPST) is a formal method to make concurrent programming simpler. The idea is to use type checking to automatically prove safety (protocol compliance) and liveness (communication deadlock freedom) of implementations relative to specifications. Discourje is an existing run-time verification library for communication protocols in Clojure, based on dynamic MPST. The original version of Discourje can detect only safety violations. In this paper, we present an extension of Discourje to detect also liveness violations.

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 on top of *shared memory*, for—supposedly—*channels* are easier to use than *locks*. However, empirical research shows that, actually, "message passing does not necessarily make multithreaded programs less error-prone than shared memory" [36]. 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 that "bad" channel actions never happen: <u>if</u> a channel action happens in I, <u>then</u> it is allowed to happen by S (protocol compliance). Liveness means that "good" channel actions eventually happen (communication deadlock freedom).

Multiparty session typing (MPST). MPST [17] is a formal method to automatically prove safety and liveness of implementations relative to specifications. The idea is to implement communication protocols as *sessions* (of communicating threads), specify them as *behavioural types* [1, 21], and verify the former against the latter using behavioural type checking. Formally, the central theorem is that well-typedness implies safety and liveness. Over the past fifteen years, much progress has been made, including the development of many tools to combine MPST with mainstream languages (e.g., F# [31], F* [37], Go [9], Java [19,20], OCaml [22], Rust [26,27], Scala [3,10,11,34], and TypeScript [29]).

Behavioural type checking can be done *statically* at compile-time or *dynamically* at run-time. The disadvantage of static MPST is, it is conservative: statically checking *each possible run* of a session is often prohibitively complicated—if computable at all—so sessions are often unnecessarily rejected. In contrast, the advantage of dynamic MPST is, it is liberal: dynamically checking *one actual run* of a session is much simpler, so sessions are never unnecessarily rejected.

This work. *Discourje* (pronounced "discourse") [13,14,18] is a library that adds dynamic MPST to *Clojure*[1]. It has a specification language to write behavioural types (embedded as an internal DSL in Clojure) and a verification engine to dynamically type-check sessions against them. The **key design goals** have been to achieve high expressiveness (cf. static MPST) and to be particularly mindful of ergonomics (i.e., make Discourje's usage as frictionless as possible).

In a nutshell, at run-time, Discourje's dynamic type checker simulates behavioural type S—as if it were a state machine—alongside session I. Each time when a channel action is about to happen in I, the dynamic type checker intervenes and first verifies if a corresponding transition can happen in S. If so, both the channel action and the transition happen. If not, an exception is thrown.

However, while safety violations are detected in this way (protocol incompliance), liveness violations are not (communication deadlocks: threads cyclically depend on each others' channel actions, and so, they collectively get stuck). This is a serious limitation relative to static MPST. In this paper, we present an extension of Discourje to detect also liveness violations. Achieving this, *without compromising the key design goals*, has been an elusive problem that for years we did not know how to solve (e.g., we could not reuse variants of existing techniques for static MPST at run-time, as this would negatively affect expressiveness).

Sect. 2 of this paper demonstrates that it can be done, while Sect. 3 outlines how. The key idea is to use "mock" channels, which mimic "real" channels, to track ongoing communications: before any channel action happens on a real channel, it is first tried on a corresponding mock channel, allowing us to check if *all* threads would get stuck in a *total* communication deadlock as a result.

2 Demonstration

We demonstrate the extension to detect liveness violations with two examples. For reference, Fig. 1 summarises the main elements of Discourje and Clojure.

Example 1. The *Two-Buyer* protocol consists of *Buyer1*, *Buyer2*, and *Seller* [17]: "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."

Fig. 2 shows a behavioural type and a session. It is safe and live. In contrast, if we had accidentally written `(<!! c3)` on line 11 (i.e., Buyer1 tries to receive

[1] A Lisp that runs on the JVM, with core support for channel-based message passing.

Discourje:

– (defthread *id*)/(defsession *id* [*args*] *body*) specifies a thread name/protocol.
– (-->>/--> *t* *p* *q*) specifies an asynchronous/synchronous communication of a value of data type *t* through a buffered/unbuffered channel from *p* to *q*.
– (alt ...) and (cat/par ...) specify choice and sequencing/interleaving.
– Names of threads and protocols are prefixed by an otherwise meaningless colon.

Clojure:

– (thread *body*), (chan), and (chan *size*) implement the creation of new thread, a new unbuffered channel, and a new buffered channel.
– (>!! *ch* *expr*) implements the send of the value of *expr* through *ch*.
– (<!! *ch*) implements the receive of a value through *ch*.
– (alts!! [act_1 ... act_n]) implements a selection of one of the channel actions, depending on their dis/enabledness (cf. `select` of POSIX sockets and Go channels). If act_i is a send, it is a pair [*ch* *v*]; if it is a receive, it is just *ch*. The function returns a pair [*v* *ch*] where *v* is the value sent/received, and *ch* is the channel.

Fig. 1: Discourje and Clojure in a nutshell

```
(defthread :buyer1)
(defthread :buyer2)
(defthread :seller)

(defsession :two-buyer []
  (cat
   (-->> String :buyer1 :seller)
   (par
    (cat
     (-->> Double :seller :buyer1)
     (-->> Double :buyer1 :buyer2))
    (-->> Double :seller :buyer2))
   (-->> Boolean :buyer2 :seller)))
```

(a) Specification in Discourje

```
(def c1 (chan 1))
(def c2 (chan 1))
(def c3 (chan 1))
(def c4 (chan 1))
(def c5 (chan 1))
(def c6 (chan 1))

(thread ;; Buyer1
  (>!! c1 "book")
  (let
    [x (<!! c5)
     y (/ x 2)]
    (>!! c2 y)))
```

```
(thread ;; Buyer2
  (let
    [x (<!! c6)
     y (<!! c2)
     z (= x y)]
    (>!! c4 z)))

(thread ;; Seller
  (<!! c1)
  (>!! c5 20.00)
  (>!! c6 20.00)
  (println
    (<!! c4)))
```

(b) Implementation in Clojure

To dynamically type-check the session, the following code creates a *monitor* for the session, and *links* it to each channel along with the intended sender and receiver:

```
(def m (monitor :two-buyer :n 3))
(link c1 :buyer1 :buyer2 m) (link c2 :buyer1 :seller m)
(link c3 :buyer2 :buyer1 m) (link c4 :buyer2 :seller m)
(link c5 :seller :buyer1 m) (link c6 :seller :buyer2 m)
```

Fig. 2: Two-Buyer (Exmp. 1)

from Buyer2 instead of Seller), then it deadlocks. The original Discourje does not detect this liveness violation, but with the extension, an exception is thrown. □

Example 2. The *Load Balancing* protocol consists of *Client*, *Server1*, *Server2*, and *LoadBalancer*. First, a request is communicated synchronously from Client to LoadBalancer, and asynchronously from LoadBalancer to Server1 or Server2. Next, the response is communicated synchronously from that server to Client.

```
(defthread :c) (defthread :s1)
(defthread :b) (defthread :s2)

(defsession :load-balancer []
  (cat
   (-->> Long :c :b)
   (alt
    (cat
     (-->> Long :b  :s1)
     (-->  Long :s1 :c))
    (cat
     (-->> Long :b  :s2)
     (-->  Long :s2 :c)))))
```

(a) Specification in Discourje

```
1  (def c1 (chan))
2  (def c2 (chan))
3  (def c3 (chan))
4  (def c4 (chan 512))
5  (def c5 (chan 1024))
6
7  ;; Load Balancer
8  (thread
9    (let [x (<!! c1)]
10     (alts!!
11       [[c4 x]
12        [c5 x]]))))
13
14 (thread ;; Client
15   (>!! c1 5)
16   (alts!! [c2 c3])))
17
18 (thread ;; Server1
19   (let [x (<!! c2)
20         y (inc x)]
21     (>!! c2 y))))
22
23 (thread ;; Server2
24   (let [x (<!! c3)
25         y (inc x)]
26     (>!! c3 y))))
```

(b) Implementation in Clojure

To dynamically type-check the session:

```
(def m (monitor :load-balancer :n 4)) (link c4 :b :s1 m) (link c2 :s1 :c m)
                     (link c1 :c :b m) (link c5 :b :s2 m) (link c3 :s2 :c m)
```

Fig. 3: Load Balancing (Exmp. 2

Fig. 3 shows a behavioural type and a session. It is safe but not live. There are two deadlocks. The first one occurs because Server1 and Server2 try to receive from `c2` and `c3` on lines 19 and 23; this should be `c4` and `c5`. The second deadlock occurs because one of the servers will never receive a value and, as a result, block the entire program from terminating. The original Discourje does not detect these liveness violations, but with the extension, exceptions are thrown. □

3 Technical Details

Requirements. In this section, we outline how the extension to detect liveness violations works, focussing on the core deadlock detection algorithm. We begin by stating the rather complicated requirements for this algorithm, as entailed by Discourje's key design goals regarding expressiveness and ergonomics (Sect. 1):

– **Expressiveness:** The algorithm must be applicable to any combination of buffered and unbuffered channels, and to all functions `>!!` (send), `<!!` (receive), and `alts!!` (select). Thus, the programmer can continue to freely mix synchronous and asynchronous sends/receives, possibly selected dynamically.
– **Ergonomics:** The algorithm must call only into the public API of Clojure's standard libraries, without modifying the internals, and without relying on JVM interoperability. Thus, the programmer can write portable code that runs on different versions of Clojure and on different architectures.

The combination of these requirements has made the design of the algorithm elusive. For instance, the expressiveness requirement means that we cannot simply reuse existing distributed algorithms for deadlock detection (e.g., [6, 16, 25, 35]), as they typically do not support mixing of synchrony and asynchrony. The ergonomics requirement means that we cannot instrument Clojure's internal code to manage threads, nor can we use Java's thread monitoring facilities.

Terminology. A *channel action* is either a *send* of *v* through *ch*, represented as `[`*ch v*`]`, or a *receive* through channel *ch*, represented as just *ch* (cf. `alts!!` in Fig. 1). A channel action is *pending* if it has been initiated but not yet completed. A pending channel action is either *enabled* or *disabled*, depending on *ch*:

- when *ch* is a buffered channel, a pending send is enabled iff *ch* is non-full, while a pending receive is enabled iff *ch* is non-empty;
- when *ch* is an unbuffered channel, a pending send is enabled iff a corresponding receive is pending, and vice versa.

When a thread initiates channel actions, but they are disabled, it is *suspended*. When a disabled channel action becomes enabled, the suspended thread is *resumed*. A *communication deadlock* is a situation where each thread is suspended.

Setting the stage. Normally, channel actions are initiated via functions `>!!`, `<!!`, and `alts!!`. When these functions are called using the extension, the dynamic type checker intervenes and first calls `(detect-deadlocks [`act_1 `...` act_n`])` to initiate corresponding "mock" channel actions on "mock" channels. Each mock channel mimics a "real" channel and is used only by the dynamic type checker.

The mock channels have the same un/buffered properties and contents as the real channels, except that values are replaced with tokens. So, if `detect-deadlocks` detects a deadlock on the mock channels, then a deadlock will occur on the real channels, too. (Mock channels are also essential to detect safety violations.)

To initiate the mock channel actions, a separate function in the public API of Clojure's standard libraries is used: `(do-alts` *f acts config*`)`. It resembles `alts!!`, except that it never suspends the calling thread. Instead, a call of `do-alts` immediately returns and, asynchronously, initiates the channel actions in *acts* and calls *f* when one is completed. In this way, initiation of mock channel actions can be decoupled from suspension of threads (demonstrated below).

Algorithm. Let `n` be the number of threads. The idea to detect deadlocks is to identify the situation when `n-1` threads are already suspended, while the **last thread** is **about to be suspended**. In that situation, instead of suspending the last thread, an exception is thrown to flag the liveness violation. In code:

```
(defn detect-deadlocks [mock-acts] ;; act1 ... actn
  (let [ret (about-to-be-suspended? mock-acts)]
    (if (true? ret)
      (let [ret (last-thread? mock-acts)]
        (if (true? ret) (throw (ex-info "deadlock!" {})) ret)) ret)))
```

Function `about-to-be-suspended?` checks if any of the `mock-acts` is enabled. If so, it immediately initiates and completes it, and returns the result (of the form `[`*v ch*`]`). If not, the function returns `true` to indicate that the current thread would indeed be suspended if `mock-acts` were to be initiated. In code:

```
(defn about-to-be-suspended? [mock-acts]
  (let [ret @(do-alts (fn [_] nil) mock-acts {:default nil})]
    (if (not= ret [nil :default]) ret true)))
```

On line 7, optional parameter `{:default nil}` configures `alts!!` such that it immediately returns `[nil :default]` when all `mock-acts` are disabled.

Function `last-thread?` increments the number of suspended threads and checks if the number is less than `n`. If so, it initiates `mock-acts`, and actually suspends the current thread. If not, the function returns `true` to indicate that the current thread is indeed the last one, so a deadlock is detected. In code:

```
9  (def i (atom 0)) ;; number of suspended threads (private to the algorithm)
10 
11 (defn last-thread? [mock-acts]
12   (if (< (swap! i inc) n)  ;; increment `i` (`swap!` returns the new value)
13     (let [p (promise)]     ;; create promise to store result of `mock-acts`
14       (do-alts (fn [x] (deliver p x)) mock-acts {}) ;; initiate `mock-acts`,
15                            ;; and store result `x` of one of them in `p`
16                            ;; upon completion, all asynchronously
17       (let [ret (deref p)] ;; suspend thread (`deref` blocks until `deliver`)
18         (swap! i dec)      ;; decrement `i`
19         ret)
20     true))
```

The code shown so far explains the general idea behind the algorithm. However, the details are more involved: our presentation does not yet account for data races, several of which are possible. For instance, suppose that there are two threads (Alice and Bob), that they initiate corresponding channel actions (no deadlock), and that calls of `detect-deadlocks` are scheduled as follows:

> **(1)** Alice executes `about-to-be-suspended?`. It returns `true`. **(2)** Bob executes `about-to-be-suspended?`. It, again, returns `true`, as Alice has not yet executed `last-thread?`. **(3)** Bob executes `last-thread?`. It increments `n` to `1` and suspends Bob. **(4)** Alice executes `last-thread?`. It increments `n` to `2`, detects that Alice is last, and immediately returns `nil`.

At this point, mistakenly, an exception is thrown. There are more subtle data races, too. The core issue is that `about-to-be-suspended?` and `last-thread?` should be run *atomically* to avoid problematic schedules (e.g., the one above). Details appear in the technical report [23, Sect. A]. The actual source code was validated using both unit tests and whole-program tests.

4 Conclusion

Closest to the work in this paper is existing work on dynamic MPST [4,15,30–32] and alternate forms of dynamic behavioural typing [7, 8, 12, 28]. However, none of these tools can check for liveness at run-time. Also closely related is existing work on dynamic deadlock detection in distributed systems (e.g., [6, 16, 25, 35]). However, as stated in Sect. 3, these algorithms do not fit our requirements. Finally, we are aware of two other works that use formal techniques to reason about Clojure programs: the formalisation of an optional type system for Clojure [5], and a translation from Clojure to Boogie [2,33]. In future work, we aim to study and optimise the performance overhead of our deadlock detection algorithm.

Disclosure of Interests. The author has no competing interests to declare that are relevant to the content of this article.

Data Availability Statement

The artifact is available on Zenodo [24]. It contains the new version of Discourje, including the examples of this paper.

References

1. Ancona et al., D.: Behavioral types in programming languages. Foundations and Trends in Programming Languages **3**(2-3) (2016)
2. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. LNCS, vol. 4111 (2005)
3. 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)
4. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. Theor. Comput. Sci. **669** (2017)
5. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for Clojure. In: ESOP. LNCS, vol. 9632 (2016)
6. Bracha, G., Toueg, S.: Distributed deadlock detection. Distributed Comput. **2**(3), 127–138 (1987)
7. Burlò, C.B., Francalanza, A., Scalas, A.: On the monitorability of session types, in theory and practice. In: ECOOP. LIPIcs, vol. 194, pp. 20:1–20:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021)
8. Burlò, C.B., Francalanza, A., Scalas, A., Trubiani, C., Tuosto, E.: PSTMonitor: Monitor synthesis from probabilistic session types. Sci. Comput. Program. **222**, 102847 (2022)
9. 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) (2019)
10. 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 (2022)
11. Ferreira, F., Jongmans, S.: Oven: Safe and live communication protocols in Scala, using synthetic behavioural type analysis. In: ISSTA. pp. 1511–1514. ACM (2023)
12. Gommerstadt, H., Jia, L., Pfenning, F.: Session-typed concurrent contracts. J. Log. Algebraic Methods Program. **124**, 100731 (2022)
13. Hamers, R., Jongmans, S.: Discourje: Runtime verification of communication protocols in Clojure. In: TACAS (1). LNCS, vol. 12078 (2020)
14. Hamers, R., Jongmans, S.: Safe sessions of channel actions in clojure: A tour of the discourje project. In: ISoLA (1). Lecture Notes in Computer Science, vol. 12476, pp. 489–508. Springer (2020)
15. van den Heuvel, B., Pérez, J.A., Dobre, R.A.: Monitoring blackbox implementations of multiparty session protocols. In: RV. Lecture Notes in Computer Science, vol. 14245, pp. 66–85. Springer (2023)
16. Hilbrich, T., de Supinski, B.R., Nagel, W.E., Protze, J., Baier, C., Müller, M.S.: Distributed wait state tracking for runtime MPI deadlock detection. In: SC. pp. 16:1–16:12. ACM (2013)

17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL (2008)
18. Horlings, E., Jongmans, S.: Analysis of specifications of multiparty sessions with `dcj-lint`. In: ESEC/SIGSOFT FSE. pp. 1590–1594. ACM (2021)
19. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. LNCS, vol. 9633 (2016)
20. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. LNCS, vol. 10202 (2017)
21. 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) (2016)
22. Imai, K., Neykova, R., Yoshida, N., Yuen, S.: Multiparty session programming with global protocol combinators. In: ECOOP. LIPIcs, vol. 166 (2020)
23. Jongmans, S.S.: Discourje: Run-Time Verification of Communication Protocols in Clojure – Live at Last (Technical Report) (2023), `https://arxiv.org/abs/2407.00540`
24. Jongmans, S.: Discourje: Run-time verification of communication protocols in Clojure – live at last (artifact) (2024). `https://doi.org/10.5281/zenodo.12519843`
25. Krivokapic, N., Kemper, A., Gudes, E.: Deadlock detection in distributed database systems: A new algorithm and a comparative performance analysis. VLDB J. **8**(2), 79–100 (1999)
26. Lagaillardie, N., Neykova, R., Yoshida, N.: Implementing multiparty session types in rust. In: COORDINATION. LNCS, vol. 12134 (2020)
27. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay safe under panic: Affine Rust programming with multiparty session types. In: ECOOP. LIPIcs, vol. 222 (2022)
28. Melgratti, H.C., Padovani, L.: Chaperone contracts for higher-order sessions. Proc. ACM Program. Lang. **1**(ICFP), 35:1–35:29 (2017)
29. Miu, A., Ferreira, F., Yoshida, N., Zhou, F.: Communication-safe web programming in TypeScript with routed multiparty session types. In: CC (2021)
30. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Asp. Comput. **29**(5) (2017)
31. 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 (2018)
32. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: CC (2017)
33. Pinzaru, G., Rivera, V.: Towards static verification of Clojure contract-based programs. In: TOOLS. LNCS, vol. 11771 (2019)
34. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP. LIPIcs, vol. 74 (2017)
35. Srinivasan, S., Rajaram, R.: A decentralized deadlock detection and resolution algorithm for generalized model in distributed systems. Distributed Parallel Databases **29**(4), 261–276 (2011)
36. Tu, T., Liu, X., Song, L., Zhang, Y.: Understanding real-world concurrency bugs in Go. In: ASPLOS (2019)
37. Zhou, F., Ferreira, F., Hu, R., Neykova, R., Yoshida, N.: Statically verified refinements for multiparty protocols. Proc. ACM Program. Lang. **4**(OOPSLA) (2020)