Safe Sessions of Channel Actions in Clojure: A Tour of the Discourje Project

Ruben Hamers[1] and Sung-Shik Jongmans[1,2](✉)

[1] Open University, Heerlen, The Netherlands
ssj@ou.nl

[2] CWI, Amsterdam, The Netherlands

Abstract. To simplify shared-memory concurrent programming, in addition to low-level synchronisation primitives, several modern programming languages have started to offer core support for higher-level communication primitives as well, in the guise of message passing through channels. Yet, a growing body of evidence suggests that channel-based programming abstractions for shared memory also have their issues.

The Discourje project aims to help programmers cope with message-passing concurrency bugs in Clojure programs, based on run-time verification and dynamic monitoring. The idea is that programmers write not only implementations, but also specifications (of sessions of channel actions). Discourje then offers a library to ensure that implementations run safely relative to specifications (= "bad" channel actions never happen).

This paper gives a tour of the current state of Discourje, by example; it is intended to serve both as a general overview for readers who are unfamiliar with previous work on Discourje, and as an introduction to new features for readers who are familiar.

1 Introduction

Background. To take advantage of today's and tomorrow's multi-core processors, shared-memory concurrent programming—a notoriously complex enterprise—is becoming increasingly important. To alleviate some of the complexities, in addition to low-level *synchronisation primitives*, several modern programming languages have started to offer core support for higher-level *communication primitives* as well, in the guise of message passing through *channels* (e.g., Go [21], Rust [40], Clojure [11]). The idea is that, beyond their usage in distributed computing, channels can also serve as a *programming abstraction* for shared memory, supposedly less prone to concurrency bugs than locks, semaphores, and the like. Notably, the official Go documentation recommends programmers to "not communicate by sharing memory; instead, share memory by communicating" [20].

Yet, a growing body of evidence suggests that channel-based programming abstractions for shared memory also have their issues. For instance, in the 2016–2018 editions of the annual Go survey [16–18], "[respondents] *least agreed* that

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12476, pp. 489–508, 2020.
https://doi.org/10.1007/978-3-030-61362-4_28

they are able to effectively debug uses of Go's concurrency features", while in the 2019 edition [19], "debugging concurrency" has the *lowest satisfaction rate* of all eleven "very or critically important" topics (as indicated by the majority of respondents). Moreover, after studying 171 concurrency bugs in popular open source Go programs [45], Tu et al. find that "message passing does not necessarily make multi-threaded programs less error-prone than shared memory."

Recently, several research projects emerged that aim to help programmers cope with concurrency bugs in Go programs [7,30,31,36,43], based on compile-time verification and static analysis; the resulting tools complement Go's static type-checker in a natural fashion, and their compile-time usage integrates well—at least potentially—with established Go programming practices. However, while similar *compile-time techniques* may suit other *statically typed languages* (e.g., Rust) at least as well, their appropriateness seems less obvious for *dynamically typed languages* (e.g., Clojure): for such languages, technical and cultural differences mean that *run-time techniques* may be preferable. The Discourje project is a research vehicle to develop and study such techniques: ultimately, the aim is to help programmers cope with concurrency bugs in Clojure programs,[1] based on run-time verification and dynamic monitoring.

The Discourje Project. Discourje,[2] pronounced "discourse", addresses the following problem: given a specification S of the *roles* (threads), the *infrastructure* (channels between threads), and the *sessions* (communications through channels) that an implementation I in Clojure should fulfill, how to check—at run-time—that an execution of I is indeed *safe* relative to S? Safety means that "bad" channel actions never happen: if a channel action happens in I, then it can happen in S. For instance, typical specifications rule out common channel-based concurrency bugs [45], such as sends without receives, receives without sends, and type mismatches (actual type sent $\neq$ expected type received).

Roughly, the idea is to execute specification S—as if it were a state machine—alongside implementation I, in "perfect synchrony"; this means that, to provide safety, a channel action in I happens if and only if a corresponding transition happens in S. To achieve this, following standard run-time verification practices [3], two extra components are needed: a *monitor* (of S) and *instrumentation* (of I). Specifically, every time that a channel action is about to happen in I, the instrumentation quickly intervenes and first asks the monitor if S can make a corresponding transition. If the monitor answers "yes", both the channel action in I and the corresponding transition in S happen; if "no", an exception is thrown, while the channel action is aborted (= safe). Facilitating this approach, Discourje offers the programmer easy-to-learn libraries to write specifications, add monitors, and add instrumentation to Clojure programs.

[1] Clojure [11,23] is a dynamically typed, functional language (impure) that compiles to Java bytecode. As a dialect of Lisp, Clojure follows the code-as-data philosophy, offers a powerful macro system, and is written in parenthesised prefix notation.

[2] https://github.com/discourje.

In recent editions of the annual Clojure survey [9,10], respondents indicated that "ease of development" is one of Clojure's most important strengths (more important than "runtime performance"). For this reason, and to make Discourje non-invasive to deploy and start using, we emphasise *ergonomics* (including expressiveness) in the design and implementation of the libraries. Notably:

1. We leverage Clojure's macro system to offer the specification language as a library of macros. As a result, the programmer can write specifications and implementations in the same syntactic style, using the same editor (no external tools needed), towards a seamless specification–implementation experience. Monitors can subsequently be added with simple function calls.
2. Control-flow operators in the specification language have the same names as those in Clojure, for a gentle learning curve.
3. Normally, in Clojure, channel-based programming abstractions can be used by loading standard library `clojure.core.async`. To add instrumentation, the only thing the programmer needs to change, is load `discourje.core.async` instead, for the primitives Discourje currently supports: **`thread`** (new thread), **`chan`** (new channel; unbuffered or buffered),[3] **`close!`** (close), **`>!!`** (send), **`<!!`** (receive), and **`alts!!`**| (select). This means, in particular, that the programmer does not need to write the implementation with Discourje in mind: instrumentation can straightforwardly be added afterwards.

When `clojure.core.async` was introduced in 2013 [8], already, it was suggested that "certain kinds of automated correctness analysis" are possible, but at the time, "no work ha[d] been done on that front". To our knowledge, Discourje is the first project that addresses this open problem.

This Paper. This paper gives a tour of the current state of Discourje, by example. It is geared towards demonstrating two core concepts of `clojure.core.async` that we did not support before [22], and which significantly improve applicability: *unbuffered channels* (to perform handshake communications) and *selects* (to await enabledness of one of several channel actions). This paper has two aims and intends to address two audiences: **(i)** for readers who *are not* familiar with previous work on Discourje [22], this paper serves as a gentle overview of the general idea and expressiveness; **(ii)** for readers who *are* familiar with previous work, this paper introduces our new support for unbuffered channels and selects.

The tour consists of three Clojure programs, each of which simulates a game: it starts in Sect. 2 with Tic–Tac–Toe; it continues in Sect. 3 with Rock–Paper–Scissors; it ends in Sect. 4 with Go Fish. In each of these examples, essentially,

[3] With *unbuffered channels*, in the absence of a buffer, both sends and receives are blocking until a reciprocal channel action is performed on the other end of the channel. With *buffered channels*, in the presence of a bounded, n-capacity, order-preserving buffer, sends are blocking until the buffer is not full (then, a message is added to the back of the buffer), while receives are blocking until the buffer is not empty (then, a message is removed from the front of the buffer).

```
1  (defrole :alice) (defrole :bob)
2
3  (defsession :ttt []
4    (alt (:ttt-turn :alice :bob)
5         (:ttt-turn :bob :alice)))
6  (defsession :ttt-turn [r1 r2]
7    (cat (--> Long r1 r2)
8         (alt (:ttt-turn r2 r1)
9              (par (close r1 r2)
10                  (close r2 r1)))))
```

Fig. 1. Specification of Tic–Tac–Toe

the safety property that we aim to ensure is that players never violate the “interaction rules” of the game (e.g. proper turn-taking), as prescribed by the specifications; however, we do not check full functional correctness (e.g., we check if players properly take turns to make moves, but we do not check if every move is actually valid in the current state of the game). In Sect. 5, we give a brief overview of the underlying formal foundation. For reference, a summary of Clojure’s core functions and macros is given in Appendix A. Full code (specifications and implementations) can be downloaded via the project’s website (footnote 2).

2 The Tour: Tic–Tac–Toe

Overview. We start the tour with a program that simulates a game of Tic–Tac–Toe.[4] The program consists of two threads and two oppositely directed channels through which these threads communicate. The threads take turns to make plays on thread-local copies of the grid; at the end of its turn, the active thread sends its play to the other thread and becomes passive, while the other thread receives the play, becomes active, updates its copy of the grid accordingly, and makes the next play. This example demonstrates the following features:

– Specification: roles; unbuffered communication (binary);[5] close; concatenation; choice; parallel; session parameters (roles).
– Implementation: channels; send; receive; close; monitor; instrumentation.

Specification. A Discourje specification of Tic–Tac–Toe is shown in Fig. 1. Core Discourje functions and macros are typeset in **`font`**.

Line 1 defines two roles (**`defrole`**), identified by `:alice` and `:bob`. Lines 3–10 define two sessions (**`defsession`**), identified by `:ttt` (zero formal parameters) and `:ttt-turn` (two formal parameters for roles, identified by `r1` and `r2`).

Session `:ttt-turn` represents one turn of `r1` (active) against `r2` (passive). It prescribes a concatenation (**`cat`**):

1. First, a message of type `Long` is communicated from `r1` to `r2`, unbuffered (**`-->`**). The idea is that `r1` sends its play this turn to `r2`.

[4] Tic–Tac–Toe is a two-player game played on a 3×3 grid. Players take turns to fill the initially blank spaces of the grid with crosses (“X”) and noughts (“O”). The first player to fill three consecutive spaces, in any direction, with the same symbol wins.
[5] A version of this example with buffered communication appears elsewhere [22].

2. Then, there is a choice (**`alt`**):
   (a) Either, there is another instance of session `:ttt-turn`, but now with `r2` active and `r1` passive. The idea is that `r1` did not win or draw this turn, so the game continues.
   (b) Or, channels are closed (**`close`**), in parallel (**`par`**). The idea is that `r1` did win or draw this turn, so the game ends.
   We note that the closes may happen in any order; this is crucial, as neither one of the closes is causally related to the other. In the implementation, additional "covert interaction" (= synchronisation/ communication outside the specification) would be needed to order them.

Session `:ttt` represents the whole game. It prescribes a choice between either an initial instance of session `:ttt-turn` with actual parameters `:alice` and `:bob`, or `:bob` and `:alice`, depending on who takes the first turn. Thus, at the specification level, it is undecided who goes first; this is an implementation detail.

As concatenation, choice, and recursion are supported, any regular expression (over unbuffered communications and closes) can be written. However, for convenience, shorthands are available for the following patterns: 0-or-more repetitions (**`*`**), 1-or-more (**`+`**), and 0-or-1 (**`?`**). Thus, the programmer never needs to use recursion to write regular expressions. The syntax and semantics of the remaining five macros are the same as those in standard library `clojure.spec.alpha`, to make Discourje easy to learn. For the same reason, the notation to define and "call" sessions is similar to the notation to define and call functions.

Implementation. A Clojure implementation of Tic–Tac–Toe is shown in Fig. 2. Core Clojure functions and macros are typeset in **`font`**.

Line 1 loads five functions and macros from `clojure.core.async`. Lines 3–11 define constants (`blank`, `cross`, `nought`, `initial-grid`) and functions (`get-blank`, `put`, `not-final?`) to represent Tic–Tac–Toe concepts. Lines 11–12 define unbuffered channels (`a->b` and `b->a`) that implement the infrastructure through which the threads communicate. As these channels are unbuffered, sends and receives block until reciprocal channel actions are performed.

Lines 16–26 and 27–37 define threads that implement roles `:alice` and `:bob`. Both threads execute a loop, starting with a blank initial grid. In each iteration, `:alice` first gets the index of a blank space on the grid, then plays a cross in that space, then sends a message to `:bob` to communicate the index (a message of type `Long`), then awaits a message from `:bob`, and then updates the grid accordingly; `:bob` acts symmetrically. After every grid update, `:alice` or `:bob` checks if it has reached a final grid; if so, the loop is exited and channels are closed.[6]

[6] Many data structures in Clojure—including the vector that implements the grid—are *persistent* and, thus, effectively *immutable*: every operation on an old data structure leaves it unmodified and, instead, returns a new data structure. In concurrent programs, including Tic–Tac–Toe, persistent data structures can be used as thread-local copies of data, but modifications need to be explicitly communicated. Persistence also means that classical data races cannot happen: if threads communicate only persistent data structures through channels, freedom of data races is guaranteed.

```
 1 (require '[clojure.core.async :refer [thread chan close! >!! <!!]])
 2
 3 (def blank " ") (def cross "x") (def nought "o")
 4
 5 (def initial-grid [blank blank blank   ;; an initial 3x3 grid of blank spaces,
 6                    blank blank blank   ;;   implemented as a vector of length 9
 7                    blank blank blank]) ;;   (persistent data structure)
 8
 9 (def get-blank  (fn [g]          ...)) ;; returns a blank space in g
10 (def put        (fn [g i x-or-o] ...)) ;; returns g, but with i set to x-or-o
11 (def not-final? (fn [g]          ...)) ;; returns true iff g is not final
12
13 (def a->b (chan)) (def b<-a a->b) ;; b<-a is an alias of a->b
14 (def b->a (chan)) (def a<-b b->a) ;; a<-b is an alias of b->a
15
16 (thread ;; for :alice                       27 (thread ;; for :bob
17   (loop [g initial-grid]                    28   (loop [g initial-grid]
18     (let [i (get-blank g)                   29     (let [i (<!! b<-a)
19           g (put g i cross)]                30           g (put g i cross)]
20       (>!! a->b i)                          31       (if (not-final? g)
21       (if (not-final? g)                    32         (let [i (get-blank g)
22         (let [i (<!! a<-b)                  33               g (put g i nought)]
23               g (put g i nought)]           34           (>!! b->a i)
24           (if (not-final? g)                35           (if (not-final? g)
25             (recur g))))))                  36             (recur g))))))
26   (close! a->b))                            37   (close! b->a))
```

Fig. 2. Implementation of Tic–Tac–Toe (dashed arrows: matching send/receive)

Safety. The implementation in Fig. 2 runs fine—*supposedly*—but to really ensure that it satisfies the specification in Fig. 1 (written independently), the programmer can add a monitor and instrumentation. The few changes needed, are shown in Fig. 3: to add instrumentation, on line 1 (which replaces line 1 in Fig. 2), `discourje.core.async` is loaded instead of `clojure.core.async`; to add a monitor, on lines 12–14 (which replace line 12–14), a monitor is created for session `:ttt`, and then, channels `a->b` and `b->a` are associated with a sender, receiver, and monitor. No other changes are needed: notably, the code that implement roles `:alice` and `:bob` in Fig. 2 stays exactly the same. This shows that Discourje is non-invasive to deploy and start using.

With these changes in place, safety is guaranteed: if a non-compliant channel action were to be attempted, the monitor prevents it from happening and throws an exception. Because the implementation in Fig. 2 actually satisfies the specification in Fig. 1, an exception is *never* thrown. In contrast, if the programmer were to change `Long` to `String` on line 7 in Fig. 1, an exception is *always* thrown; if they were to change **`par`** to **`cat`** on line 9, an exception is *sometimes* thrown, depending on the execution and scheduling of threads.

```
(require '[discourje.core.async :refer [thread chan close! >!! <!!]])

(def m (monitor (session :ttt [])))
(link a->b (role :alice) (role :bob) m)
(link b->a (role :bob) (role :alice) m)
```

Fig. 3. Ensuring safety of Tic–Tac–Toe

```
(defrole :player)

(defsession :rps [ids]
  (:rps-round ids empty-set))

(defsession :rps-round [ids co-ids]
  (if (> (count ids) 1)
    (cat (par-every [i ids
                     j (disj player-ids i)]
           (--> String (:player i) (:player j)))
         (alt-every [winner-ids (power-set ids)]
           (let [loser-ids (difference ids winner-ids)]
             (par (:rps-round winner-ids (union co-ids loser-ids))
                  (par-every [i loser-ids
                              j (disj (union ids co-ids) i)]
                    (close (:player i) (:player j)))))))))
```

Fig. 4. Discourje specification of Rock–Paper–Scissors

3 The Tour: Rock–Paper–Scissors

Overview. We continue the tour with a program that simulates a game of Rock–Paper–Scissors.[7] The program consists of k threads and $k^2 - k$ directed channels from every thread to every other thread. In every round, every thread chooses an item—rock, paper, or scissors—and sends it to every other thread; then, when all items have been received, every thread determines if it goes to the next round. This example demonstrates the following features:

– Specification: indexed roles; unbuffered communication (multiparty); conditional; local bindings; quantification (existential; unordered universal); session parameters (role indices); set operations; non-determinism (implicit).
– Implementation: select; external synchronisation.

[7] Rock–Paper–Scissors is a multiplayer game played in rounds. In every round, every remaining player chooses an item—rock, paper, or scissors—and reveals it. A player goes to the next round, unless some other player defeats them, while they defeat no other player, based on the chosen items in the current round ("scissors cuts paper, paper covers rock, rock crushes scissors"). The last player to remain wins.

Specification. A Discourje specification of Rock–Paper–Scissors is shown in Fig. 4. Auxiliary Discourje functions for operations on sets are typeset in `font`.

Line 1 defines one role, identified by `:player`. Lines 3–16 define two sessions, identified by `:rps` (one formal parameter for role indices) and `:rps-round` (two formal parameters). There are two key differences with Fig. 1 in Sect. 2:

– Whereas roles `:alice` and `:bob` in Tic–Tac–Toe are enacted each by a *single* thread, role `:player` in Rock–Paper–Scissors is enacted by *multiple* threads. To distinguish between different threads that enact the same role, roles can be *indexed* in specifications. For instance, with 0-based indexing, `(:player 5)` represents the thread that implements the sixth player.
– Whereas formal parameters of session `:ttt-turn` in Tic–Tac–Toe range over roles, those of sessions `:rps` and `:rps-round` range over (sets of) role indices. This exemplifies that session parameters can range over arbitrary values.

Session `:rps-round` represents one round of the game; threads indexed by elements in set `ids` are still in, while threads indexed by elements in set `co-ids` are already out. If fewer than two threads are still in (**`if`**), the session is effectively empty. Otherwise, session `:rps-round` prescribes a concatenation:

1. First, there is an unordered universal quantification (**`par-every`**) of local variable `i` over domain `ids`, and simultaneously, local variable `j` over domain "`ids` without `i`" (`disj`). In general, an unordered universal quantification gives rise to a "big parallel" of branches, each of which is formed by binding values in domains to local variables (cf. parallel for-loops). In this particular example, every such branch prescribes a communication of a message of type `String` from `(:player i)` to `(:player j)`, unbuffered. The idea is that every `(:player i)` sends its chosen item to every other in-game `(:player j)`, in no particular order; the order is an implementation detail.
2. Then, there is an existential quantification (**`alt-every`**) of local variable `winner-ids` over domain "set of subsets of `ids`" (`power-set`). Similar to unordered universal quantification, in general, existential quantification gives rise to a "big choice" of branches. In this particular example, every such branch prescribes a binding (**`let`**) of local variable `loser-ids` to "`ids` without `winner-ids`" (`difference`), after which there is a parallel:
   (a) Concurrently, there is another instance of session `:rps-round`, but now with only `winner-ids` retained from `ids`, and with `loser-ids` added to `co-ids` (`union`). The idea is that only every `(:player i)` that is a winner this round goes to the next round.
   (b) Concurrently, there is an unordered universal quantification of `i` over `loser-ids`, and simultaneously, `j` over "all indices except `i`". Every branch of this "big parallel" prescribes a close of the channel from `(:player i)` to `(player j)`. The idea is that every `(:player i)` that is a loser this round closes its channel to every other in-game or out-game `(:player j)`.

   Thus, the idea of the existential quantification is, for every possible subset of winners, that the winners stay in the game, while the losers go out.
   We note that the usage of existential quantification in this way makes the

```
(def k ...) ;; number of threads (e.g., read from stdin)

(def rock "rock") (def paper "paper") (def scissors "scissors") ;; items

(def rock-or-paper-or-scissors (fn []    ...)) ;; returns an item
(def winner-ids                (fn [r]   ...)) ;; returns winners in round r
(def winner-or-loser?          (fn [r i] ...)) ;; returns true iff thread i is
                                                ;;   winner or loser in round r
(def chans   (mesh chan (range k)))
(def barrier (java.util.concurrent.Phaser. k))

(doseq [i (range k)]
  (thread ;; for role (:player i)
    (loop [ids (range k)]
      (let [item (rock-or-paper-or-scissors)
            opponent-ids (remove #{i} ids)
            round (loop [acts (into (puts  chans [i item] opponent-ids)
                                    (takes chans opponent-ids i))
                         round {}] ;; map from ids to items (initially empty)
                    (if (empty? acts)
                      (assoc round i item)
                      (let [[v c] (alts!! acts)]
                        (recur (remove #{[c item] c} acts)
                               (assoc round (putter-id chans c) v)))))]
        (.arriveAndAwaitAdvance barrier)
        (if (winner-or-loser? round i)
          (do (.arriveAndDeregister barrier)
              (doseq [j (remove #i (range k))]
                (close! (chans i j))))
          (recur (winner-ids round))))))
```

Fig. 5. Implementation of Rock–Paper–Scissors, excerpt

specification implicitly *non-deterministic*: different branches may start with the exact same (sequence of) channel action(s), until a "distinguishing" channel action happens. This requires non-trivial bookkeeping to support.

Session `:rps` represents the whole game. It prescribes an initial instance of session `:rps-round`, where all threads are in, and no threads are out (`empty-set`).

In addition to existential quantification and unordered universal quantification, there is support for ordered universal quantification (**`cat-every`**): similar to the former two, the latter one gives rise to a "big concatenation" of branches (cf. sequential for-loops). We note that quantification domains need to be finite to ensure that checking whether a channel action is safe can happen in finite time.

The syntax and semantics of the functions for operations on sets are the same as those in standard library `clojure.set`, to make Discourje easy to learn.

Implementation. A Clojure implementation of Rock–Paper–Scissors is shown in Fig. 5 (excerpt; some details are left out to save space). Auxiliary Discourje functions are typeset in `font`; shading indicates external Java calls.

Line 1 defines a constant for the number of threads k. Lines 3–7 define constants and functions to represent Rock–Paper–Scissors concepts. Line 9 defines a collection of $k^2 - k$ unbuffered channels that implement the infrastructure, intended to be used as a fully connected mesh network; the threads are represented by indices in the range from 0 to k (exclusive). We note that `mesh` is "merely" an auxiliary Discourje function to simplify defining collections of channels; just as the other auxiliary Discourje functions used in Fig. 5, it works also without adding a monitor or instrumentation. Line 10 defines a reusable synchronisation barrier, imported from Java standard library `java.util.concurrent`, leveraging Clojure's interoperability with Java; shortly, we clarify the need for this.

Lines 12–30 define k copies of a thread that implements role `:player`. Every such thread executes two parametrised loops: an outer one, each of whose iterations comprises a round, and an inner one, each of whose iterations comprises a channel action (send or receive, indirectly using select). Salient aspects:

– According to the specification (Fig. 4), in the first half of every round (lines 8–10), the items that are chosen by in-game threads are communicated among them. This can be problematic: as channels are unbuffered, sends and receives are blocking until reciprocal channel actions are performed, so unless threads agree on a global order to perform such individual channel actions, deadlocks are looming. But, global orders are hard to get right and brittle to maintain. An alternative solution is to use *selects*: in general, a select consumes a collection of channel actions as input, then blocks until one of those actions becomes enabled, then performs that action, then unblocks, and then produces that action's output as output. Thus, a select performs *one* channel action from a collection, depending on its enabledness at run-time.
In this particular example, instead of performing globally ordered individual sends and receives, every thread performs a series of selects (**`alts!!`**) in the inner loop. Initially, the collection of channel actions consists of all sends (`puts`) and receives (`takes`) that a thread needs to perform in a round. When a select finishes, the channel action that was performed is removed from the collection, and the inner loop continues. Because every thread behaves in this way, reciprocal channel actions are always enabled, so every thread makes progress. Thus, by using selects, the order in which communications happen, is not implemented (nor is it specified), but deadlocks are still avoided.
– According to the specification (Fig. 4), there is a strict order between the first half of every round (lines 8–10) and the second half (lines 11–16): *all* channel actions that belong to the first half need to have happened before proceeding to the second half. This can be problematic: additional synchronisation or timing measures are needed to ensure that "fast threads"—those that perform their channel actions early—wait for "slow threads" to catch up.

One solution is to extend the session with additional communications. An alternative solution is to mix communication primitives with synchronisation primitives. In this particular example, we adopt the latter solution: we mix channels with a barrier from `java.util.concurrent` (shaded code in Fig. 5). This demonstrates that channel-based programming abstractions (checked using Discourje) can be mixed seamlessly with other concurrency libraries (not checked), which is common practice [44,45].

Safety. A monitor and instrumentation can be added as in Fig. 3.

4 The Tour: Go Fish

Overview. We end the tour with a program that simulates a game of Go Fish.[8] Like the Rock–Paper–Scissors program in Sect. 3, the Go Fish program consists of $k + 1$ threads (players, plus dealer), and $k^2 + k$ channels from every thread to every other thread; unlike the Rock–Paper–Scissors program, however, all interactions among threads happen through channels (no need for external barriers, locks, etc.). This example demonstrates the following features:

- Specification: user-defined message types; repetition (0-or-more); quantification (ordered universal); non-determinism (explicit).
- Implementation: message type-based control flow.

Specification. A Discourje specification of Go Fish is shown in Fig. 6.

Line 1 defines two roles, identified by `:dealer` (enacted by a single thread) and `:player` (multiple threads). Lines 3–29 define two sessions, identified by `:gf` and `:gf-turn`. Lines -7–0 define six user-defined message types.

Session `:gf-turn` represents one turn of `(:player i)`. It prescribes a "big choice". In every branch, the idea is as follows. First, `(:player i)` asks `(:player j)` for some card. Then, there is a choice:

1. Either, `(:player j)` replies with the card that it was asked for, which happens to be the last card that `(:player i)` needs (to complete its last group), so it informs `(:dealer)`, and the game ends.
2. Or, `(:player j)` replies with the card that it was asked for, which does not happen to be the last card that `(:player i)` needs, so `(:player i)` takes another turn, and the game continues.

We note that the specification is explicitly non-deterministic: the first branch and the second branch both start with the same channel action.

[8] Go Fish is a multiplayer game played with a standard 52-card deck. A dealer shuffles the deck and deals an initial hand to every player. Then, players take turns to collect groups of cards of the same rank. Every turn, the active player asks a passive player for a card. If the asked player has it, the asking player gets it and takes another turn; if not, the asked player tells the asking player ("go"), the asking player gets a card from the dealer ("fish"), and the turn is passed to the asked player. The first player to hold only complete groups wins. (This version of Go Fish is due to Parlett [38].).

```
(defrecord Turn [])
(defrecord Ask [s r])
(defrecord Card [s r])
(defrecord OutOfCards [])
(defrecord Go [])
(defrecord Fish [])
;; above, parameters s and r
;;   abbreviate suit and rank
```

```
(defrole :dealer) (defrole :player)

(defsession :gf [ids]
  (cat (par-every [i ids]
         (cat-every [_ (range 5)]
           (--> Card :dealer (:player i))))
       (alt-every [i ids]
         (cat (--> Turn :dealer (:player i))
              (:gf-turn i ids)))
       (par-every [i ids]
         (cat (close :dealer (:player i))
              (par (cat (* (--> Card (:player i) :dealer))
                        (close (:player i) :dealer))
                   (par-every [j (disj ids i)]
                     (close (:player i) (:player j))))))))

(defsession :gf-turn [i ids]
  (alt-every [j (disj ids i)]
    (cat (--> Ask (:player i) (:player j))
         (alt (cat (--> Card (:player j) (:player i))
                   (--> OutOfCards (:player i) :dealer))
              (cat (--> Card (:player j) (:player i))
                   (:gf-turn i ids))
              (cat (--> Go (:player j) (:player i))
                   (--> Fish (:player i) :dealer)
                   (alt (--> Card :dealer (:player i))
                        (--> OutOfCards :dealer (:player i)))
                   (--> Turn (:player i) (:player j))
                   (:gf-turn j ids))))))
```

Fig. 6. Discourje specification of Go Fish, including message types

3. Or, (:player j) does not reply with the card that it was asked for, so (:player i) tries to "fish" a card from :dealer, after which (:player i) passes the turn to (:player j), and the game continues.

Session :gf represents the whole game. It prescribes a concatenation:

1. First, there is a "big parallel". The idea is that :dealer deals every player an initial hand of five cards, in no particular order (implementation detail).
2. Then, there is a "big choice". The idea is that :dealer passes the first turn to one of the players (implementation detail). During the game, the players pass the turn among themselves unbeknownst to :dealer.
3. Then, there is a "big parallel". The idea is that the game has ended at this point, so :dealer closes its channel to every (:player i), in no particular order (implementation detail), after which every (:player i) sends its hand back to :dealer through the oppositely directed channel, closes that channel, and closes its channel to every other (:player j), in no particular order.

```
 1 (doseq [i (range k)]                    11 (thread ...) ;; for :dealer
 2   (thread ;; for (:player i)
 3     (... (let [[v c] (alts!! ...)]
 4            (condp = (type v)
 5              Turn (... (let [v (<!! ...)]
 6                          (condp = (type v)
 7                            Card ...
 8                            Go   ...))) ;; another <!! and condp in this case
 9              Ask  ...
10              nil  ...)))))
```

Fig. 7. Implementation of Rock–Paper–Scissors, excerpt

Implementation. A Clojure implementation of Go Fish is shown in Fig. 7 (excerpt; many details are left out to save space).

To demonstrate that Discourje supports message type-based control flow, Fig. 7 shows fragments of code where messages are received—directly with `<!!` and indirectly with `alts!!`—by threads that implement role `:player`. Specifically:

- On line 3, `alts!!` is used to receive a message `v` from another `:player` or from `:dealer`. This message is either of type `Turn` (received from another `:player`), or of type `Ask` (idem), or `nil` ("received" from `:dealer`).
  We note that a "receive" of `nil` happens only, and automatically, when the channel from `:dealer` to `(:player i)` is closed. Such a degenerate "receive" is used by `(:player i)` to detect that the game has ended.
- On line 5, `<!!` is used to receive a message of type `Card` or `Go` from `(:player j)`, to which a message of type `Ask` was sent previously (not shown).

Safety. A monitor and instrumentation can be added as in Fig. 3.

5 Foundation

Overview. Discourje is built on a formal foundation, inspired by process algebra (e.g., [15]) and multiparty session types (e.g., [46]). In a nutshell, let $\mathbb{S}$ and $\mathbb{I}$ be sets of specifications and implementations. Then, given a specification $S \in \mathbb{S}$ and an implementation $I \in \mathbb{I}$, the "game" is to check if a trace of I is also a trace of S. We briefly summarise the theory (for unbuffered channels), based on [22].

Specification. Let $\mathbb{R}$ be a set of roles, ranged over by p, q, r. Let $\mathbb{F}$, $\mathbb{V}$, $\mathbb{X}$, and $\mathbb{E}$ be sets of functions, values, variables, and expressions, ranged over by f, v, x, and e, such that $\mathbb{F} \subseteq \mathbb{V} \subseteq \mathbb{E}$ and $\mathbb{X} \subseteq \mathbb{E}$; for simplicity, we leave the elements of $\mathbb{F}$, $\mathbb{V}$, $\mathbb{E}$, and $\mathbb{X}$ unspecified (although, we stipulate that $\mathbb{E}$ contains at least boolean, numerical, and lambda expressions). Let $\tilde{\mathbb{X}}$ and $\tilde{\mathbb{E}}$ be sets of lists of variables and lists of expressions, ranged over by $\tilde{e}$ and $\tilde{x}$. The syntax of specifications is defined as follows (with corresponding Discourje macros):

$$S \in \mathbb{S} ::= \boxed{1} \mid \overbrace{r_1[e_1] \twoheadrightarrow r_2[e_2] : f}^{\texttt{-->}} \mid \overbrace{r_1[e_1] \not\twoheadrightarrow r_2[e_2]}^{\texttt{close}} \mid \overbrace{S_1 \cdot S_2}^{\texttt{cat}} \mid \overbrace{S_1 + S_2}^{\texttt{alt}} \mid$$
$$\underbrace{S_1 \parallel S_2}_{\texttt{par}} \mid \underbrace{e \rhd S_1 \diamond S_2}_{\texttt{if}} \mid \underbrace{X(\tilde{e})}_{\text{“call”}} \mid \langle S \mid \underbrace{X_1(\tilde{x}_1) = S_1}_{\texttt{defsession}}, \ldots, \underbrace{X_n(\tilde{x}_n) = S_n}_{\texttt{defsession}} \rangle$$

Term $\boxed{1}$, which represents a *skip*, is the only term for which no corresponding Discourje macro exists; its shading indicates that it is used primarily/ only to define the operational semantics (it should not be used directly). Conversely, Discourje macro calls for which no corresponding term exist, are encodable. For instance, `(cat-every [x (range 5)] ...)`, with `x` free in the ellipses, corresponds with $\langle X(5) \mid X(x) = (x > 1 \rhd (\ldots \cdot X(x-1)) \diamond (x > 0 \rhd \ldots \diamond \boxed{1}))\rangle$.

$$\frac{}{\boxed{1}\downarrow}\,[\text{S}\downarrow\text{-One}] \qquad \frac{S_1\downarrow \textbf{ and } S_2\downarrow}{S_1 \cdot S_2\downarrow}\,[\text{S}\downarrow\text{-Cat}] \qquad \frac{S_{i\in\{1,2\}}\downarrow}{S_1 + S_2\downarrow}\,[\text{S}\downarrow\text{-Alt}]$$

Fig. 8. Operational semantics of specifications (termination), excerpt

$$\frac{e_i \Downarrow i \textbf{ and } e_j \Downarrow j \textbf{ and } (f\ v) \Downarrow \mathsf{true}}{p[e_i] \twoheadrightarrow q[e_j] : f \xrightarrow{p[i]q[j]!?v} \boxed{1}}\,[\text{S-Unbuf}] \qquad \frac{e_i \Downarrow i \textbf{ and } e_j \Downarrow j}{p[e_i] \not\twoheadrightarrow q[e_j] \xrightarrow{p[i]q[j]\bullet} \boxed{1}}\,[\text{S-Close}]$$

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1 \cdot S_2 \xrightarrow{\alpha} S_1' \cdot S_2}\,[\text{S-Cat1}] \qquad \frac{S_1\downarrow \textbf{ and } S_2 \xrightarrow{\alpha} S_2'}{S_1 \cdot S_2 \xrightarrow{\alpha} S_2'}\,[\text{S-Cat2}] \qquad \frac{S_{i\in\{1,2\}} \xrightarrow{\alpha} S'}{S_1 + S_2 \xrightarrow{\alpha} S'}\,[\text{S-Alt}]$$

Fig. 9. Operational semantics of specifications (reduction), excerpt

The operational semantics of specifications is defined in terms of evaluation relation $\Downarrow$, termination predicate $\downarrow$, and labelled reduction relation $\rightarrow$. Labels, ranged over by α, are of the form $p[i]q[j]!?v$ (unbuffered send and receive; handshake) and $p[i]q[j]\bullet$ (close). A subset of rules are shown in Figs. 8–9; they are standard (cf. Basic Process Algebra [15], plus merge, conditional and recursion).

Implementation. The syntax of implementations is defined as follows (it does not cover all features of Clojure used in Sects. 2–4, but a smaller core set):

$$I \in \mathbb{I} ::= \mathsf{skip} \mid \mathsf{if}\ I_1\ I_2\ I_3 \mid \mathsf{loop}\ \tilde{x}\ \tilde{e}\ I \mid \mathsf{recur}\ \tilde{e} \mid I_1 \cdot I_2 \mid$$
$$I_1 \parallel I_2 \mid \mathsf{chan} \mid \mathsf{close}\ e \mid \mathsf{send}\ e_1\ e_2 \mid \mathsf{recv}\ e\ x \mid \mathsf{select}\ \tilde{I}$$

The operational semantics of the calculus is defined in terms of labelled reductions of pairs $(I, \mathcal{H})$, where $\mathcal{H}$ is a heap (map from locations to channel

$$\frac{e \Downarrow v \qquad (I_v, \mathcal{H}) \xrightarrow{\alpha} (I', \mathcal{H}')}{(\mathsf{if}\ e\ I_{\mathsf{true}}\ I_{\mathsf{false}}, \mathcal{H}) \xrightarrow{\alpha} (I', \mathcal{H}')}\ [\text{I-If}] \qquad \frac{\tilde{e} \Downarrow \tilde{v}\ \textbf{and}\ I[\mathsf{loop}\ \tilde{x}\ \tilde{e}\ I/\mathsf{recur}][\tilde{v}/\tilde{x}] \xrightarrow{\alpha} I'}{(\mathsf{loop}\ \tilde{x}\ \tilde{e}\ I, \mathcal{H}) \xrightarrow{\alpha} (I', \mathcal{H})}\ [\text{I-Loop}]$$

$$\frac{\begin{array}{c}\ell \notin \mathcal{H}\ \textbf{and} \\ (I[\ell/x], \mathcal{H}[\ell \mapsto \top]) \xrightarrow{\alpha} (I', \mathcal{H}')\end{array}}{(\mathsf{chan}\ x \cdot I, \mathcal{H}) \xrightarrow{\alpha} (I', \mathcal{H}')}\ [\text{I-Chan}] \qquad \frac{e \Downarrow \ell\ \textbf{and}\ \ell \in \mathcal{H}}{(\mathsf{close}\ e \cdot I, \mathcal{H}) \xrightarrow{\ell\bullet} (I, \mathcal{H}[\ell \mapsto \bot])}\ [\text{I-Close}]$$

$$\frac{\begin{array}{c}\mathsf{send}\ e_1\ e_v \in \mathcal{I}_1\ \textbf{and}\ \mathsf{recv}\ e_2\ x \in \mathcal{I}_2\ \textbf{and} \\ e_1 \Downarrow \ell\ \textbf{and}\ e_v \Downarrow v\ \textbf{and}\ e_2 \Downarrow \ell\ \textbf{and}\ \mathcal{H}(\ell) = \top\end{array}}{((\mathsf{select}\ \mathcal{I}_1 \cdot I_1) \parallel (\mathsf{select}\ \mathcal{I}_2 \cdot I_2), \mathcal{H}) \xrightarrow{\ell!?v} (I_1 \parallel I_2[v/x], \mathcal{H})}\ [\text{I-Unbuf}]$$

$$\frac{(I_1, \mathcal{H}) \xrightarrow{\alpha} (I_1', \mathcal{H}')}{(I_1 \parallel I_2, \mathcal{H}) \xrightarrow{\alpha} (I_1' \parallel I_2, \mathcal{H}')}\ [\text{I-Par1}] \qquad \frac{(I_2, \mathcal{H}) \xrightarrow{\alpha} (I_2', \mathcal{H}')}{(I_1 \parallel I_2, \mathcal{H}) \xrightarrow{\alpha} (I_1 \parallel I_2', \mathcal{H}')}\ [\text{I-Par2}]$$

Fig. 10. Operational semantics of implementations, excerpt

states). As we cover only unbuffered channels in this paper (buffered channels are covered elsewhere [22]), a channel state is represented by $\top$ (if the channel is open) of $\bot$ (closed). Labels are of the form $\ell!?v$ and $\ell\bullet$. A subset of rules are shown in Fig. 10 (notably, a structural congruence rule has been omitted).

Safety. Let $\dagger$ be a function from heap locations to sender–receiver pairs; it corresponds with the linkage of channels to a monitor (Fig. 3). Abusing notation, we write $\dagger(\ell!?v)$ and $\dagger(\ell\bullet)$ instead of $\dagger(\ell)!?v$ and $\dagger(\ell)\bullet$.

We formalise safety ("bad channel actions never happen") in terms of *simulation*. Specifically, implementation I is $\dagger$-simulated by specification S if there exists a $\preceq\ \subseteq \mathbb{I} \times \mathbb{S}$ such that: (1) $I \preceq S$, and (2) for all $\hat{I}, \hat{I}' \in \mathbb{I}$ and $\hat{S} \in \mathbb{S}$, if $\hat{I} \preceq \hat{S}$ and $\hat{I} \xrightarrow{\alpha} \hat{I}'$, then there exists an $\hat{S}' \in \mathbb{S}$ such that $\hat{I}' \preceq \hat{S}'$ and $\hat{S} \xrightarrow{\dagger(\alpha)} \hat{S}'$.

To ensure safety at run-time, a monitor dynamically constructs a simulation relation to check if the implementation is simulated by the specification, incrementally, as channel actions are performed. A subtle—but important—detail is that the relation is constructed not for the whole reduction relation of the implementation, but only for a "linear" subrelation (a trace; the actual execution).

6 Conclusion

Related Work. The Discourje project is strongly influenced by work on *multiparty session types* (MPST) [24]. The idea of MPST is to specify protocols as behavioural types [1,28] against which threads are subsequently type-checked; the theory guarantees that static well-typedness of threads at compile-time implies dynamic safety of their channel actions at run-time. In recent years, several practical implementations were developed, mostly for statically typed languages (e.g., C [37], Java [26,27], Scala [42], F# [34], Go [7]), and to lesser extent for dynamically typed languages (e.g., Python [25], Erlang [35]).

Discourje takes advantage of two key properties of the application domain to offer higher expressiveness than existing MPST tools: we apply run-time verification instead of compile-time analysis, and we target shared-memory programs instead of distributed systems. The former means that no implementations are conservatively rejected (so, Discourje supports more implementations); the latter means that no decomposition of "global" specifications into "local" specifications—one for every role—is required, which is needed in existing MPST tools, but often not possible [7] (so, Discourje supports more specifications). Notably, we support non-deterministic choice and value-dependent control flow in specifications. To our knowledge, in the context of MPST, we are the first to leverage run-time verification and shared memory together, although they have been considered in isolation:

– There are MPST approaches that combine static type-checking with a form of distributed run-time monitoring and/or assertion checking [4,5,13,33,34]. In contrast to Discourje, however, these dynamic techniques still rely on decomposition; none of the specifications in this paper are supported.
– Decomposition-free MPST has also been explored by López et al. [32,41]. Their idea is to specify MPI communication protocols in an MPI-tailored DSL, inspired by MPST, and verify the implementation against the specification using deductive verification tools (VCC [12] and Why3 [14]). However, this approach does not support push-button verification: considerable manual effort is required. In contrast, Discourje is fully automated.

Verification of shared-memory concurrency with channels has received attention in the context of Go [30,31,36,43]. However, in addition to relying on static techniques (unlike Discourje), emphasis in these works is on checking deadlock-freedom, liveness, and generic safety properties, while we focus on program-specific protocol compliance. Castro et al. [7] also consider protocol compliance for Go, but their specification language is substantially less expressive than ours; none of the specifications in this paper are supported.

We are aware of only two other works that use formal techniques to reason about Clojure programs: Bonnaire-Sergeant et al. [6] formalized the optional type system for Clojure and proved soundness, while Pinzaru et al. [39] developed a translation from Clojure to Boogie [2] to verify Clojure programs annotated with pre/post-conditions. Discourje seems the first to target concurrency in Clojure.

Future Work. We are currently working towards several new features: (1) automated *recovery* when a violation is detected, instead of throwing an exception; (2) *meta-verification* of specifications, to detect "insensible" specifications; (3) first-class support for histories, to improve expressiveness with history-based conditionals. Also, we are interested to explore "weaving", as in aspect-oriented programming [29], to further reduce the effort of adding instrumentation [3].

Finally, research is needed to better understand the *effectiveness* of Discourje (e.g., in terms of reduced development costs). In particular, we would like to gain insight into difficulties that programmers face when writing specifications. We try to make Discourje easy to learn and use by supporting standard Clojure idioms wherever possible (e.g., for regular expressions; Sect. 2), but scientific evidence on usability is still to be gathered.

Acknowledgements. We thank Luc Edixhoven and anonymous reviewers for comments on an earlier version of this paper. Funded by the Netherlands Organisation of Scientific Research (NWO): 016.Veni.192.103.

A Clojure

Standard library `clojure.core`:

- `(`**`def`** x e`)`: first evaluates e to v; then binds x to v in the global environment.
- `(`**`if`** e_1 e_2 e_3`)`: first evaluates e_1; if `true`, evaluates e_2; else, evaluates e_3.
- `(`**`let`** `[`x_1 e_1 `...` x_n e_n`]` e`)`: first evaluates e_1 to v_1; then evaluates e_2 to v_2 with x_1 bound to v_1; ...; then evaluates e_n to v_n with x_1, ..., x_{n-1} bound to v_1, ..., v_{n-1}; then evaluates e with x_1, ..., x_n bound to v_1, ..., v_n.
- `(`**`fn`** `[`x_1 `...` x_n`]` e_1 `...` e_m`)`: evaluates to a function with parameters x_1, ..., x_n and creates a recursion point; then, when applied to arguments v_1, ..., v_n, sequentially evaluates e_1, ..., e_m with x_1, ..., x_n bound to v_1, ..., v_n.
- `(`**`loop`** `[`x_1 e_1 `...` x_n e_n`]` e`)`: same as **`let`**, but also creates a recursion point.
- `(`**`recur`** e_1 `...` e_n`)`: first evaluates e_1, ..., e_n to v_1, ..., v_n; then evaluates the nearest recursion point with x_1, ..., x_n bound to v_1, ..., v_n.

Standard library `clojure.core.async`:

- `(`**`thread`** e`)`: starts a new thread that evaluates e.
- `(`**`chan`**`)`: evaluates to a new unbuffered channel.
- `(`**`close!`** e`)`: first evaluates e to channel c; then closes c.
- `(`**`>!!`** e_1 e_2`)`: first evaluates e_1 to channel c; then evaluates e_2 to v; then sends v through c.
- `(`**`<!!`** e`)`: first evaluates e to channel c; then receives a value through c.
- `(`**`alts!!`** $[a_1 \ldots a_n]$`)`: for every a_i of the form `[`$e_{i,1}$ $e_{i,2}$`]` (send) or e_i (receive), evaluates $e_{i,1}$ and e_i to channel c_i, and then, evaluates $e_{i,2}$ to v; then, waits until one of these channel actions can be performed; then, performs a channel action that can be performed (non-deterministically selected).

References

1. Ancona, D., et al.: Behavioral types in programming languages. Found. Trends Program. Lang. **3**(2–3), 95–230 (2016)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17

3. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
4. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. Theor. Comput. Sci. **669**, 33–58 (2017)
5. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_12
6. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for Clojure. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 68–94. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_4
7. 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)
8. Clojure Team: Clojure - Clojure core.async Channels, 28 June 2013. https://clojure.org/news/2013/06/28/clojure-clore-async-channels. Accessed 1 Sept 2019
9. Clojure Team: Clojure - State of Clojure 2019 Results, 04 February 2019. https://clojure.org/news/2019/02/04/state-of-clojure-2019. Accessed 1 Sept 2019
10. Clojure Team: Clojure - State of Clojure 2020 Results, 20 February 2019. https://clojure.org/news/2020/02/20/state-of-clojure-2020. Accessed 28 May 2020
11. Clojure Team: Clojure (nd). https://clojure.org. Accessed 1 Sept 2019
12. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_2
13. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. Formal Methods Syst. Des. **46**(3), 197–225 (2015)
14. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
15. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-662-04293-9
16. Go Team: Go 2016 Survey Results - The Go Blog, 03 June 2017. https://blog.golang.org/survey2016-results. Accessed 1 Sept 2019
17. Go Team: Go 2017 Survey Results - The Go Blog, 26 February 2018. https://blog.golang.org/survey2017-results. Accessed 1 Sept 2019
18. Go Team: Go 2018 Survey Results - The Go Blog, 28 March 2019. https://blog.golang.org/survey2018-results. Accessed 1 Sept 2019
19. Go Team: Go Developer Survey 2019 Results - The Go Blog, 20 April 2020. https://blog.golang.org/survey2019-results. Accessed 8 May 2020
20. Go Team: Effective Go - The Go Programming Language (nd). https://golang.org/doc/effective_go.html. Accessed 8 May 2020
21. Go Team: The Go Programming Language (nd). https://golang.org. Accessed 1 Sept 2019

22. Hamers, R., Jongmans, S.-S.: Discourje: runtime verification of communication protocols in Clojure. In: Biere, A., Parker, D. (eds.) TACAS 2020. LNCS, vol. 12078, pp. 266–284. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_15
23. Hickey, R.: The Clojure programming language. In: DLS, p. 1. ACM (2008)
24. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284. ACM (2008)
25. Hu, R., Neykova, R., Yoshida, N., Demangeon, R., Honda, K.: Practical interruptible conversations. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 130–148. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_8
26. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wąsowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 401–418. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_24
27. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: Huisman, M., Rubin, J. (eds.) FASE 2017. LNCS, vol. 10202, pp. 116–133. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54494-5_7
28. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016)
29. Kiczales, G., et al.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
30. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off go: liveness and safety for channel-based programming. In: POPL, pp. 748–761. ACM (2017)
31. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in go using behavioural types. In: ICSE, pp. 1137–1148. ACM (2018)
32. López, H.A., et al.: Protocol-based verification of message-passing parallel programs. In: OOPSLA, pp. 280–298. ACM (2015)
33. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Aspects Comput. **29**(5), 877–910 (2017). https://doi.org/10.1007/s00165-017-0420-8
34. 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)
35. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: CC, pp. 98–108. ACM (2017)
36. Ng, N., Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis. In: CC, pp. 174–184. ACM (2016)
37. Ng, N., Yoshida, N., Honda, K.: Multiparty session C: safe parallel programming with message optimisation. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 202–218. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30561-0_15
38. Parlett, D.: The Penguin Book of Card Games. Penguin (2008)
39. Pinzaru, G., Rivera, V.: Towards static verification of Clojure contract-based programs. In: Mazzara, M., Bruel, J.-M., Meyer, B., Petrenko, A. (eds.) TOOLS 2019. LNCS, vol. 11771, pp. 73–80. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29852-4_5
40. Rust Team: Rust Programming Language (nd). https://rust-lang.org. Accessed 1 Sept 2019

41. Santos, C., Martins, F., Vasconcelos, V.T.: Deductive verification of parallel programs using why3. In: ICE. EPTCS, vol. 189, pp. 128–142 (2015)
42. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP. LIPIcs, vol. 74, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
43. Stadtmüller, K., Sulzmann, M., Thiemann, P.: Static trace-based deadlock analysis for synchronous mini-go. In: Igarashi, A. (ed.) APLAS 2016. LNCS, vol. 10017, pp. 116–136. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47958-3_7
44. Tasharofi, S., Dinges, P., Johnson, R.E.: Why do Scala developers mix the actor model with other concurrency models? In: Castagna, G. (ed.) ECOOP 2013. LNCS, vol. 7920, pp. 302–326. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39038-8_13
45. Tu, T., Liu, X., Song, L., Zhang, Y.: Understanding real-world concurrency bugs in go. In: ASPLOS, pp. 865–878. ACM (2019)
46. Yoshida, N., Gheri, L.: A very gentle introduction to multiparty session types. In: Hung, D.V., D'Souza, M. (eds.) ICDCIT 2020. LNCS, vol. 11969, pp. 73–93. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-36987-3_5