International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-022-00674-y

GENERAL

Special Issue: TACAS 2020

The Discourje project: run-time verification of communication protocols in Clojure

Ruben Hamers[1] · **Erik Horlings**[1] · **Sung-Shik Jongmans**[1,2]

Accepted: 29 August 2022

Abstract
To simplify shared-memory concurrent programming, languages have started to offer core support for high-level communications primitives, in the form of message passing though channels, in addition to lower-level synchronisation primitives. Yet, a growing body of evidence suggests that channel-based programming abstractions also have their issues. The Discourje project aims to help programmers cope with channels and concurrency bugs in Clojure programs, based on dynamic analysis. The idea is that programmers write not only implementations of communication protocols in their Clojure programs, but also specifications. Discourje then offers a run-time verification library to ensure that channel actions in implementations are safe relative to specifications. The aim of this paper is to provide a comprehensive overview of the current state of Discourje, including case studies, theoretical foundations, and practical aspects.

Keywords Runtime verification · Multiparty session types · Concurrency

1 Introduction

To take advantage of modern multi-core processors, shared-memory concurrent programming—a notoriously difficult enterprise—has become increasingly important. In the wake of this development, languages have started to offer core support for high-level *communication primitives*, in the form of message passing through *channels* (e.g. Go, Rust, Clojure), in addition to lower-level *synchronisation primitives*. The idea is that channels can also serve as a *programming abstraction* for shared memory beyond their usage in distributed systems. Supposedly channels are less prone to concurrency bugs than locks, semaphores, and the like. For instance, the official Go documentation recommends programmers to "not communicate by sharing memory; instead, share memory by communicating" [1].

✉ Sung-Shik Jongmans
ssj@ou.nl

[1] Department of Computer Science, Open University of the Netherlands, Valkenburgerweg 177, 6419 AT Heerlen, Limburg, The Netherlands

[2] Centrum Wiskunde and Informatica (CWI), Stichting Nederlandse Wetenschappelijk Onderzoek Instituten (NWO-I), Science Park 123, 1098 XG Amsterdam, North Holland, The Netherlands

Yet, a growing body of evidence suggests that channel-based programming abstractions also have their issues. For instance, in the 2016–2018 editions of the annual Go survey [2–4], "[respondents] *least agreed* that they are able to effectively debug uses of Go's concurrency features", while in the 2019 edition [5], "debugging concurrency" has the *lowest satisfaction rate* of all eleven "very or critically important" topics. Moreover, after studying 171 concurrency bugs in popular open source Go programs [6], Tu et al. conclude that "message passing does not necessarily make multi-threaded programs less error-prone than shared memory".

Several research projects have emerged that aim to help programmers cope with channels and concurrency bugs in Go programs (e.g. [7–11]), based on *static analysis*. The idea is to employ *compile-time verification* to complement Go's static type-checker in a way that fits established Go programming techniques, practices, and culture. However, while similar techniques may be likely to suit other statically typed languages as well (e.g. Rust), it remains an open question if they are equally appropriate for dynamically typed languages (e.g. Clojure); technically, practically, and culturally, *run-time verification* may fit such languages better. **Discourje**—pronounced "discourse"—is a research project that aims to help programmers cope with channels and concurrency bugs in Clojure programs, based on *dynamic analysis*.

Published online: 20 September 2022

1.1 Discourje in a nutshell

1.1.1 From the programmer's perspective

A major challenge to cope with channels and concurrency bugs is as follows: how to ensure that an implementation I is *safe* relative to a specification S, where S prescribes the *roles* (implemented as threads), the *network* (implemented as channels between threads), and the *protocols* (implemented as sessions of communications through channels) that I should fulfil. Safety means that "bad" channel actions never happen: if a channel action happens in I, then it is allowed to happen in S. For instance, typical specifications rule out common concurrency bugs [6], such as sends without receives, receives without sends, and type mismatches (i.e. actual type sent $\neq$ expected type received).

The Discourje project offers a run-time verification library in Clojure, called `discourje`, to ensure safety of I relative to S. The idea is to execute specification S—as if it were a state machine—alongside implementation I using two typical run-time verification components (e.g. [12]): a *monitor* (of S) and *instrumentation* (of I). Every time 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", only an exception is thrown. Thus, a channel action in I happens if, and only if, a corresponding transition happens in S, in lockstep (i.e. "bad" channel actions never happen).

The `discourje` library facilitates writing specifications, adding monitors, and adding instrumentation to implementations written in Clojure. To make `discourje` easy and non-invasive to start using, and inspired by recent editions of the annual Clojure survey [13,14] (respondents indicate that "ease of development" is one of Clojure's most important strengths; more so than "runtime performance"), we emphasise *ergonomics* in `discourje`'s development:

- We leverage Clojure's macro system to offer the specification language for protocols as an embedded domain-specific language (DSL). As a result, the programmer can write both specifications and implementations in similar notation, using the same editor (no external tools needed), towards a seamless specification–implementation experience. Monitors can subsequently be added with simple function calls.
- To add instrumentation, the only things the programmer needs to change in an existing implementation are: (1) to load `discourje.core.async` instead of standard library `clojure.core.async` for channels; (2) to add a bit of configuration data when channels are created. This means, in particular, that the programmer does not need to write an implementation with `discourje` in mind: instrumentation can straightforwardly be added afterwards.
- The following main functions and macros from `clojure.core.async` are currently supported: **`thread`** (new thread), **`chan`** (new channel), **`close!`** (closing), **`>!!`** (send), **`<!!`** (receive), and **`alts!!`** (selection).

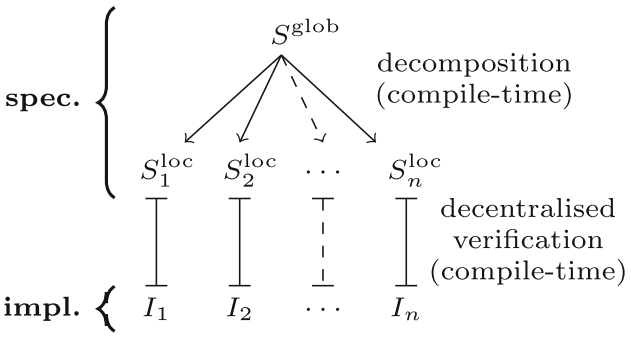


Fig. 1 Traditional MPST [16,17]

When `clojure.core.async` was introduced in 2013 [15], already, it was suggested that "certain kinds of automated correctness analysis" are possible, but at the time, "no work [had] been done on that front". To our knowledge, Discourje is the first project that addresses this open problem.

1.1.2 From a researcher's perspective

The Discourje project was originally conceived to explore a new direction in research on *multiparty session types* (MPST): since the early achievements [16,17], while substantial progress had been made both in MPST theory (e.g. extensions with time [18,19], security [20–23], parametrisation [7,24,25]) and in MPST practice (e.g. tools for F# [26], Go [7], Java [27,28], Scala [29]), nearly all efforts had targeted the domain of *statically typed languages and distributed systems*. By targeting the domain of *dynamically typed languages and shared-memory concurrent programs* instead, the Discourje project set out to enter uncharted waters. In particular, the main research question that has been driving the project from the start has been how to take advantage of the unique properties of the target domain to deliver "better" (by some definition) tools. As a result, the "Discourje approach" has diverged considerably from the "traditional MPST approach".

To explain the two fundamental differences in more detail, first, Fig. 1 visualises the traditional MPST approach. It works as follows:

1. Initially, the programmer manually writes a "global" specification S^{glob}; it prescribes the communication

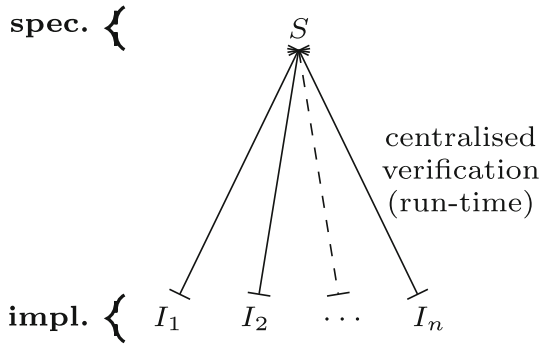


Fig. 2 Discourje (this paper)

behaviour of all roles, collectively, from a shared perspective (e.g. "first, a number is communicated from Alice to Bob; next, a Boolean is communicated from Bob to Carol or Dave".)
2. Subsequently, an MPST tool automatically decomposes S^{glob} into role-specific "local" specifications S_1^{loc}, S_2^{loc}, $\ldots$, S_n^{loc}; every S_i^{loc} prescribes the communication behaviour of one role, individually, from its own perspective (e.g. for Bob: "first, receive a number from Alice; next, send a Boolean to Carol or Dave").
3. Finally, an MPST tool automatically verifies every thread I_i in the implementation against S_i^{loc} by means of static type-checking (in the style of *behavioural type systems* [30,31]). Now, MPST theory guarantees that well-typedness at compile time implies safety at run time.

In contrast, Fig. 2 visualises the Discourje approach. It fundamentally differs from the traditional MPST approach on two accounts:

- In the traditional MPST approach, to fit established programming techniques, practices, and culture of statically typed languages, compile-time verification has been a non-negotiable requirement. However, the Discourje approach targets dynamically typed languages, which are technically, practically, and culturally different. As a result, the Discourje approach uses **run-time verification** instead of compile time.
- In the traditional MPST approach, to fit established programming practices for distributed systems, decentralised verification (i.e. type-checking against local specifications on a per-role basis) has been a non-negotiable requirement. However, the Discourje approach targets shared-memory concurrent programs, without any form of distribution (i.e. all threads are executed on the same machine). As a result, the Discourje approach uses **centralised verification without decomposition** instead of decentralised.

Due to these two fundamental differences, the Discourje approach *substantially improves expressiveness* by removing two limitations of the traditional MPST approach. The first limitation pertains to compile-time verification vs. run time: the traditional MPST approach statically rejects ill-typed-but-safe implementations (i.e. it is sound but not complete), whereas the Discourje approach dynamically rejects only unsafe implementations (i.e. it is sound and complete). The second limitation pertains to decentralised verification vs. centralised: the traditional MPST approach relies on decomposition and rejects specifications that cannot be decomposed in a behaviour-preserving way (i.e. many grammatical specifications are unsupported; e.g. [32]), whereas the Discourje approach does not rely on decomposition (i.e. all grammatical specifications are supported).

Besides these two fundamental differences, the following strengths of the traditional MPST approach remain consolidated:

- Fully automated verification of *concrete programs* (vs. abstract models);
- User-friendly *programming language-based notation* to write specifications (vs. dynamic logic or temporal logic).

1.2 This paper

The aim of this paper is to provide a comprehensive overview of the current state of the Discourje project. In Sect. 2, we present a few preliminaries on Clojure. In Sect. 3, we demonstrate the usage of the `discourje` library in a number of case studies. In Sect. 4, we present the theoretical foundations on which `discourje` is built. In Sect. 5, we discuss practical aspects, including details of `discourje`'s internals and results of performance experiments.

This paper substantially extends our TACAS 2020 paper [33] with material from our ISoLA 2020 paper [34] (notably: case studies and new features) and our ESEC/FSE 2021 paper [35] (notably: a built-in model checker for specifications). To improve the presentation, the new material is integrated throughout the paper instead of isolated in separate new sections.

Finally, Discourje is open-source: https://github.com/discourje.

2 Preliminaries on Clojure

Clojure [36–38] is a dynamically typed, functional language (impure) that compiles to Java bytecode and runs on the JVM. It is a dialect of Lisp and has a powerful macro system. In the 2019 edition of the Stack Overflow Developer Survey [39], Clojure was the 7th most loved language, outranking languages including Go, C#, Scala, Java, C++, and C.

Channel-based programming abstractions are offered in Clojure through standard library `clojure.core.async` [15]. It has both *unbuffered* and *buffered* 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. In the presence of a bounded, n-capacity, order-preserving buffer, sends are blocking until the buffer is non-full (next, a value is enqueued to the back of the buffer), while receives are blocking until the buffer is non-empty (next, a value is dequeued from the front of the buffer).

For reference, Fig. 3 summarises the main Clojure functions and macros relevant to this paper; we clarify their usage in the next sections, by example.

3 A tour of discourje

To demonstrate the usage of the `discourje` library, we take a 4-stop tour. The first stop (Sect. 3.1) presents the intended workflow of `discourje`. The remaining three stops (Sects. 3.2–3.4) present three Clojure programs that we can specify and verify using `discourje`, each of which simulates a game and requires unique features (i.e. Tic–Tac–Toe, Rock–Paper–Scissors, and Go Fish). In each of these case studies, the safety property that `discourje` ensures is that the players (i.e. threads) never violate the "interaction rules" of the game (e.g. proper turn-taking), as stated in the specifications. We note that `discourje` does not check full functional correctness (e.g. it ensures that players properly take turns to make moves, but it does not ensure that every move is valid in the current game state).

As a notational convention, in the rest of this paper, the main Clojure functions and macros are typeset in **`blue font`**, while the main `discourje` functions and macros are typeset in **`red font`**.

3.1 The workflow

Figure 4 summarises the intended workflow of `discourje`:

- First, the programmer writes a specification S using `discourje` and, possibly independently, an implementation I in Clojure.
- Next, the programmer runs I with S: during the run, a channel action in I happens if, and only if, a corresponding transition happens in S (Sect. 1.1.1).
- When an unsafe channel action is attempted, an exception is thrown.
- Next, the programmer diagnoses the problem: if it is "clearly" a bug in I, then they can fix I; else, they can analyse S using a built-in model checker for S. In the latter case, the aim is to rule out bugs in S, so the programmer can more confidently focus their attention on fixing I (even if the problem is not "clearly" a bug in I, it can still be one, *especially* with concurrency). The built-in model checker supports both generic sanity checks and protocol-specific temporal requirements.

To illustrate the workflow, we consider a classical example from the MPST literature, namely the *Two-Buyer* program: "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" [40].

Below, ▢ and ◇ indicate "actions" and "decisions" in Fig. 4, respectively.

▢ First, we write the specification in Fig. 5. Lines 1–3 specify the roles, identified by `:buyer1`, `:buyer2`, and `:seller`, while lines 4–12 specify the protocol, identified by `:two-buyer`. In general, (**`->`** t p q) specifies a *communication* of a value of type t through the unbuffered channel from p to q; (**`close`** p q) specifies *closing* of the channel from p to q; (**`cat`** S_1 … S_n) and (**`par`** S_1 … S_n) specify concatenation (i.e. sequential composition) and interleaving (i.e. parallel composition).[1,2] Additional features will be presented in the next subsections. Because `discourje` is built on top of Clojure/Java, we can also use a few Clojure/Java features to write specifications (e.g. "colon-prefixed" identifiers from Clojure and data types from Java).

Thus: lines 5–9 specify communications of a `String` (book) from `:buyer1` to `:seller`, an `Integer` (quote) from `:seller` to `:buyer1` and `:buyer2`, an `Integer` (contribution) from `:buyer1` to `:buyer2`, and a `Boolean` (accept/reject) from `:buyer2` to `:seller`; lines 10–12 specify closings of all channels, in no particular order.

▢ Next, we write the implementation in Fig. 6. Lines 1–3 implement the channels, while lines 4–24 implement the threads. In general (Fig. 3), (**`>!!`** c v) sends v through c, (**`<!!`** c) receives a value through c, and (**`close!`** c) closes c.

Thus, the quote of `:seller` is `19` (variable `x` at `:buyer1` and `:buyer2`); the contribution of `:buyer1` is half of the quote (variable `y` at `:buyer2`), and the decision of `:buyer2` is to reject (variable `z`).

[1] *Opening* of channels is currently not part of the specification language; an extension along the lines of Hu and Yoshida [28] is possible, though.

[2] We note that channels are referred to in specifications through the intended sender and receiver instead of through a separate channel identity. The advantage is that it makes specifications easier to write and understand (higher level of abstraction); the disadvantage is that multiple channels between the same two threads, in the same direction, are indistinguishable in specifications.

Standard library `clojure.core`:

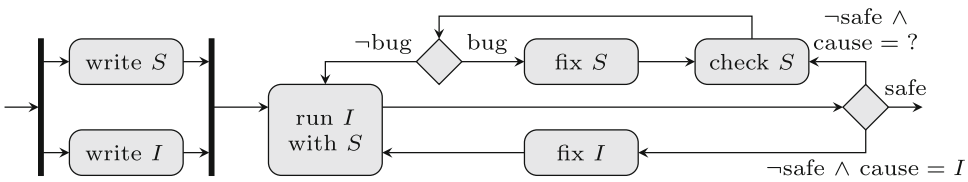
- (**def** x e): first evaluates e to v; next binds x to v in the environment.
- (**if** e_1 e_2 e_3): first evaluates e_1; if `true`, evaluates e_2; else, evaluates e_3.
- (**let** [x_1 e_1 ... x_n e_n] e): first evaluates e_1 to v_1; next evaluates e_2 to v_2 with x_1 bound to v_1; ...; next evaluates e_n to v_n with x_1, ..., x_{n-1} bound to v_1, ..., v_{n-1}; next 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; next, 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; next 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**): creates a new unbuffered channel.
- (**chan** e): first evaluates e to number $n>0$; next creates a new buffered channel of capacity n.
- (**close!** e): first evaluates e to channel c; next closes c.
- (**>!!** e_1 e_2): first evaluates e_1 to channel c; next evaluates e_2 to v; next sends v through c.
- (**<!!** e): first evaluates e to channel c; next receives a value through c.
- (**alts!!** [a_1 ... a_n]): for every a_i of the form [$e_{i,1}$ $e_{i,2}$] (send) or e_i (receive), evaluates $e_{i,1}$ or e_i to channel c_i, and next, evaluates $e_{i,2}$ to v; next, waits until one of these channel actions can be performed; next, performs a channel action that can be performed (non-deterministically selected if multiple are possible).

Fig. 3 Main Clojure functions and macros

Fig. 4 Intended workflow of `discourje`




```
(defrole :buyer1)
(defrole :buyer2)
(defrole :seller)
```

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

Fig. 5 Specification of the two-buyer protocol

```
1  (def c1 (chan)) (def c2 (chan)) ;; from :buyer1 to :buyer2 and :seller
2  (def c3 (chan)) (def c4 (chan)) ;; from :buyer2 to :buyer1 and :seller
3  (def c5 (chan)) (def c6 (chan)) ;; from :seller to :buyer1 and :buyer2

4  (thread ;; :buyer1
5    (>!! c2 "book")
6    (let [x (<!! c5)
7          y (/ x 2)]
8      (>!! c1 y))
9    (close! c1)
10   (close! c2))

11 (thread ;; :buyer2
12   (let [x (<!! c6)
13         y (<!! c1)
14         z (= x y)]
15     (>!! c4 z))
16   (close! c3)
17   (close! c4))

18 (thread ;; :seller
19   (<!! c2)
20   (>!! c5 (int 19))
21   (>!! c6 (int 19))
22   (println (<!! c4))
23   (close! c5)
24   (close! c6))
```

Fig. 6 Implementation of a two-buyer session

Fig. 7 Adding a monitor and instrumentation to the implementation in Fig. 6, using the specification in Fig. 5

```
3.5a (def m (monitor (session :two-buyer)))
3.5b (link c1 :buyer1 :buyer2 m) (link c2 :buyer1 :seller m)
3.5c (link c3 :buyer2 :buyer1 m) (link c4 :buyer2 :seller m)
3.5d (link c5 :seller :buyer1 m) (link c6 :seller :buyer2 m)
```

☐ Next, we run the implementation with the specification. To do this, we first need to add the lines in Fig. 7 between lines 3–4 in Fig. 6. That is, we create a monitor for the specification in Fig. 5 and link it to every channel, along with the intended sender and the intended receiver. Furthermore, we need to load `discourje.core.async` instead of `clojure.core.async`. Besides these little changes, no other changes are needed: notably, the code for `:buyer1`, `:buyer2`, and `:seller` in Fig. 6 stays exactly the same. This demonstrates that `discourje` is non-invasive to start using.

◇ Next, we observe an exception:

```
[SESSION FAILURE] Action ?(19/2,buyer1,buyer2)

is not enabled in current state(s): [3].

LTS in Aldebaran format:

des (0,4,5)
(0,"?(String,buyer1,seller)",1)
(1,"?(Integer,seller,buyer1)",2)
(2,"?(Integer,seller,buyer2)",3)
(3,"?(Integer,buyer1,buyer2)",4)
*** state 4 not yet expanded ***
```

The first two lines report that the implementation of `:buyer1` attempts to send value `19/2` to `:buyer2`, but that this is not allowed in the specification's current state `3`. The remaining lines show the relevant part of the *state space* of the specification, as a list of *transitions*. By matching the unsafe action reported on the first line, `?(19/2,buyer1,buyer2)`, against the label of the transition out of current state `3`, `?(Integer,buyer1,buyer2)`, we can infer that a communication from `:buyer1` to `:buyer2` is actually allowed, but that the type of the value must be `Integer`, which `19/2` is not; it is a `Ratio` value that we forgot to round down. Thus, "clearly", the problem is a bug in the implementation.

☐ Next, we fix the bug by replacing `(/ x 2)` on line 7 in Fig. 6 with `(int (/ x 2))`, to round the `Ratio` down to an `Integer`.

☐ Next, we re-run the implementation with the specification.

◇ Next, we observe *another* exception:

```
[SESSION FAILURE] Action C(buyer1,buyer2)

is not enabled in current state(s): [4].

LTS in Aldebaran format:

des (0,5,6)
(0,"?(String,buyer1,seller)",1)
(1,"?(Integer,seller,buyer1)",2)
(2,"?(Integer,seller,buyer2)",3)
(3,"?(Integer,buyer1,buyer2)",4)
(4,"?(Boolean,buyer2,seller)",5)
*** state 5 not yet expanded ***
```

By matching the unsafe action reported on the first line, `C(buyer1,buyer2)`, against the label of the transition out of current state `4`, `?(Boolean, buyer2, seller)`, we can infer that the implementation of `:buyer1` attempts to close its channel to `:buyer2`, but that the specification allows only a communication from `:buyer2` to `:seller` at this point. Thus, there seems to be a timing issue with `:buyer1`'s closing. This is not "clearly" a bug in the implementation: the specification prescribes all closings to happen at the end (Fig. 5, lines 10–12), and indeed, every thread closes its channels at the end of its run (Fig. 6, lines 9–10, 16–17, 23–24), so what goes wrong?

☐ Next, we check the specification using `discourje`'s built-in model checker, by having it automatically perform

```
?(String,buyer1,seller)
?(Integer,seller,buyer1)
?(Integer,seller,buyer2)
?(Integer,buyer1,buyer2)
?(Boolean,buyer2,seller)
C(buyer2,buyer1)
```

Fig. 8 A channel is closed, but never used

```
?(String,buyer1,seller)
?(Integer,seller,buyer1)
?(Integer,seller,buyer2)
?(Integer,buyer1,buyer2)
?(Boolean,buyer2,seller)
C(buyer1,buyer2)
```

Fig. 9 Causally unrelated actions are strictly ordered

seven generic sanity checks: three checks pertain to termination (the protocol *must always* terminate; it *may always* terminate; it *can never* terminate), three checks pertain to closings (if a channel is used, it must be closed; if a channel is closed, it must have been used; if a channel is closed, it cannot be used again), and one check pertains to causality (clarified below).

◇ Next, the model checker reports three issues. The first issue is that the specification cannot never terminate. This is intended, so we can immediately ignore it (and disable the check). The second issue is that, apparently, one of the channels can be closed before it is used.

To help debugging, the model checker provides the *witness* in Fig. 8 (i.e. a violating sequence of actions). It clarifies that after five communications, the specification allows `:buyer2` to close its channel to `:buyer1`, but actually, that channel is never used. While this is not a bug per se, it is "smelly" (cf. dead code and unused variables).

☐ Next, we remove `(close :buyer2 :buyer1)` from line 11 in Fig. 5, and also `(def c3 (chan))` and `(close! c3)` from lines 2 and 16 in Fig. 6.

☐ Next, we re-check the specification using the model checker.

◇ Next, only the third issue remains reported: at some point, apparently, two *causally unrelated* actions are allowed to happen in one order, but *not* in the other order. This can be problematic, because in the absence of a causal relation between the actions, it is impossible to write an implementation that fulfils one order but not the other, unless "covert interaction" is used (i.e. synchronisation or communication outside the specification).

To help debugging, the model checker provides the witness in Fig. 9. It clarifies that after five communications, the specification allows `:buyer1` to close its channel to `:buyer2`, but it forbids `:buyer1` to do so before `:buyer2` and `:seller` have communicated (penultimate action of the witness). However, as a non-participant in that communication, `:buyer1` cannot know when `:buyer2` and `:seller` are done (i.e. no causality), so the specification cannot be fulfilled; this is a specification bug.

☐ Next, we fix the bug by observing that the specification is too restrictive: it requires *all* channels to be closed at the end, but since `:buyer1`'s part in the protocol is already done at line 8 in Fig. 5, the specification should allow `:buyer1` to close its channels from that point onwards. We therefore replace lines 9–12 with the following:

```
9    (par (--> Boolean :buyer2 :seller)
10        (close :buyer1 :buyer2)
11        (close :buyer1 :seller))
12   (par (close :buyer2 :buyer1)
13        (close :buyer2 :seller)
14        (close :seller :buyer1)
15        (close :seller :buyer2))))
```

(The closing of the unused channel from `:buyer2` to `:buyer1` was removed in a previous step.) Thus, by judiciously introducing a new **par**-block, the specification now allows `:buyer1` to close its channels in parallel to the communication from `:buyer2` to `:seller`.

☐ Next, we re-check the specification using the model checker.

◇ Next, another causality issue is reported. The last two actions of the witness are `C(buyer1,seller)` and `C(buyer2,seller)`. Thus, the specification allows `:buyer1` and `:buyer2` to close their channels to `:seller` in that order, but not in the reverse order; since `:buyer2` cannot know when `:buyer1` is done, this is a specification bug.

☐ Next, we fix the bug by observing that the updated specification is still too restrictive: it unnecessarily requires `:buyer1` to close its channels before `:buyer2` and `:seller` can close theirs. We therefore refine lines 9–12 with another **par**-block, as follows:

```
9    (par (cat (--> Boolean :buyer2 :seller)
10             (par (close :buyer2 :buyer1)
11                  (close :buyer2 :seller)
12                  (close :seller :buyer1)
13                  (close :seller :buyer2)))
14        (close :buyer1 :buyer2)
15                  (close :buyer1 :seller))
```

☐ Next, we re-check the specification using the model checker.

◇ Next, no more issues are reported.

☐ Next, we re-run the implementation with the specification.

◇ At last, no more exceptions are reported. Thus, in several iterations, we detected and fixed bugs in both the implementation and the specification. We note that lines

9–12 of the final specification reveal intricate timing constraints. On the one hand, as a result, the specification is not easy to write, which may discourage potential users. On the other hand, the intricate timing constraints exist *regardless of whether a specification is written*; this can make writing the specification, and subsequently enjoying the benefits of runtime verification, all the more valuable. The model checker is, however, important to assist in getting the specification right.

Having demonstrated the intended workflow of `discourje`, we proceed with three case studies to systematically present the features and expressiveness of the Discourje approach (Sect. 1.1.2, Fig. 1)

3.2 Case study: Tic–Tac–Toe

3.2.1 Preface

Our first case study is a program that simulates a game of Tic–Tac–Toe.[3] It consists of two threads and two 1-capacity buffered channels through which they 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 case study demonstrates the following features:

- SPECIFICATION: roles; asynchronous communication through buffered channels; closings; concatenation (sequential composition); choice; interleaving (parallel composition); role-based parametrisation.
- IMPLEMENTATION: channels; sends; receives; closings.

3.2.2 Specification

A specification of the Tic–Tac–Toe program is shown in Fig. 10. Lines 1-2 specify two roles (**`defrole`**), identified by `:alice` and `:bob`. Lines 4–11 specify two protocols (**`defsession`**), identified by `:ttt` (zero formal parameters) and `:ttt-turn` (two formal parameters for roles, identified by `r1` and `r2`).

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

1. First, a value of type `Long` is communicated through a buffered channel from `r1` to `r2` (**`-»`**; we recall that **`->`** is used to specify unbuffered communications). The idea is that `r1` sends its play this turn to `r2`.
2. Next, there is a choice (**`alt`**):
   (a) Either, there is another instance of `:ttt-turn`, but now with `r2` as active player and `r1` as passive player. 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.
   The closings may happen in any order; this is important, as neither one of the closings is causally related to the other (i.e. in the implementation, covert interaction would be needed to order them).

Specification `:ttt` represents the whole game. It specifies a choice between either an initial instance of `: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 (implementation detail).

Since concatenation, choice, and recursion are supported in `discourje`, any regular expression (over 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 explicit recursion to write regular expressions.

3.2.3 Implementation

An implementation of the Tic–Tac–Toe program is shown in Fig. 11. Lines 1–9 define constants (`blank`, `cross`, `nought`, `initial-grid`) and functions (`get-blank`, `put`, `not-final?`) to represent Tic–Tac–Toe concepts. Lines 11-12 define buffered channels of capacity 1 (`a->b` and `b->a`) that implement the network through which the threads communicate.

Lines 14–24 and 25–35 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 value to `:bob` to communicate the index, then awaits a value 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.[4]

[3] Tic–Tac–Toe is a two-player game played on a 3×3 grid. Each player is assigned its own symbol: a cross ("X") or a nought ("O"). Players take turns to fill the initially blank spaces of the grid with their assigned symbol. The first player to fill three consecutive spaces, in any direction, wins.

[4] Many data structures in Clojure—including the vector that implements the grid—are *persistent* and, thus, effectively *immutable*: every

```
(defrole :alice)
(defrole :bob)

(defsession :ttt []
  (alt (:ttt-turn :alice :bob)
       (:ttt-turn :bob :alice)))

(defsession :ttt-turn [r1 r2]
  (cat (-->> Long r1 r2)
       (alt (:ttt-turn r2 r1)
            (par (close r1 r2)
                 (close r2 r1)))))
```

Fig. 10 Specification of the Tic–Tac–Toe program

```
(def blank " ") (def cross "x") (def nought "o")

(def initial-grid [blank blank blank   ;; an initial 3x3 grid of blank spaces,
                   blank blank blank   ;;   implemented as a vector of length 9
                   blank blank blank]) ;;   (persistent data structure)

(def get-blank  (fn [g]          ...)) ;; returns a blank space in g
(def put        (fn [g i x-or-o] ...)) ;; returns g, but with i set to x-or-o
(def not-final? (fn [g]          ...)) ;; returns true iff g is not final

(def a->b (chan 1)) (def b<-a a->b) ;; b<-a is an alias of a->b
(def b->a (chan 1)) (def a<-b b->a) ;; a<-b is an alias of b->a

(thread ;; :alice
  (loop [g initial-grid]
    (let [i (get-blank g)
          g (put g i cross)]
      (>!! a->b i)
      (if (not-final? g)
        (let [i (<!! a<-b)
              g (put g i nought)]
          (if (not-final? g)
            (recur g))))))
  (close! a->b))

(thread ;; :bob
  (loop [g initial-grid]
    (let [i (<!! b<-a)
          g (put g i cross)]
      (if (not-final? g)
        (let [i (get-blank g)
              g (put g i nought)]
          (>!! b->a i)
          (if (not-final? g)
            (recur g))))))
  (close! b->a))
```

Fig. 11 Implementation of the Tic–Tac–Toe program

A monitor and instrumentation can be added to the implementation in the same way as shown in Fig. 7. Interestingly, in this case study, the implementation is actually unsafe relative to the specification: the specification states that channels are allowed to be closed only *after* (the receive of) the previous communication is done, but in the implementation, `:alice` or `:bob` can attempt to close already *before*. There are several ways to fix this bug. One solution is to use unbuffered channels instead of buffered ones. Another solution is to mix channels with a *synchronisation barrier* from Java's standard library `java.util.concurrent` (readily usable in Clojure), to let `:alice` and `:bob` first await each other and then close (i.e. covert interaction). The next case study further demonstrates the latter idea.

3.3 Rock–Paper–Scissors

Our second case study is a program that simulates a game of Rock–Paper–Scissors.[5] 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 case study demonstrates the following features:

- SPECIFICATION: indexed roles; synchronous communication through unbuffered channels; conditional choice; local bindings; existential and unordered-universal quan-

Footnote 4 continued
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 data races cannot happen: if threads communicate only persistent data structures, freedom of data races is guaranteed.

[5] 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.

tification; index-based parameters; set operations; implicit non-determinism.
- IMPLEMENTATION: selection; covert interaction (synchronisation barrier).

3.3.1 Specification

A specification of Rock–Paper–Scissors is shown in Fig. 12; auxiliary `discourje` functions are typeset in `font`. Line 1 specifies one role, identified by `:player`. Lines 3–16 specify two protocols, identified by `:rps` (one formal parameter for role indices) and `:rps-round` (two formal parameters). There are two key differences with Fig. 10 in Sect. 3.2:

- 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*. For instance, with 0-based indexing, `(:player 5)` represents the thread that implements the sixth player.
- Whereas formal parameters of specification `:ttt-turn` in Tic–Tac–Toe range over roles, those of specifications `:rps` and `:rps-round` range over (sets of) role indices.

Specification `: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. When at least two threads are still in (**`if`**), `:rps-round` specifies 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 the domains to local variables (cf. parallel for-loops). In this particular example, every such branch specifies a communication of a value of type `String` through an unbuffered channel from `(:player i)` to `(:player j)` (**`->`**). The idea is that every `(:player i)` sends its chosen item to every other in-game `(:player j)`, in no particular order (implementation detail).
2. Next, 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 specifies a binding (**`let`**) of local variable `loser-ids` to "`ids` without `winner-ids`" (`difference`), after which:
   - There is another instance of :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.
   - 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" specifies the closing 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 `(: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 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.

Specification `:rps` represents the whole game. It specifies an initial instance of `:rps-round`, when all threads are in, and no threads are out (`empty-set`).

In addition to existential quantification and unordered-universal quantification, there is also 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 also note that 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.

3.3.2 Implementation

An implementation of the Rock–Paper–Scissors program is shown in Fig. 13; auxiliary `discourje` functions are typeset in `font`; shading indicates external Java calls for covert interaction.

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 network, intended to be used as a fully connected mesh; the threads are represented by indices in the range from 0 to k (exclusive). We note that `mesh` is an auxiliary `discourje` function to simplify defining collections of channels; just as the other auxiliary `discourje` functions used in Fig. 13, it works also without adding a monitor or instrumentation. Line 10 defines

Fig. 12 Specification of the Rock–Paper–Scissors program

```
1  (defrole :player)
2
3  (defsession :rps [ids]
4    (:rps-round ids empty-set))
5
6  (defsession :rps-round [ids co-ids]
7    (if (> (count ids) 1)
8      (cat (par-every [i ids
9                       j (disj ids i)]
10           (--> String (:player i) (:player j)))
11         (alt-every [winner-ids (power-set ids)]
12           (let [loser-ids (difference ids winner-ids)]
13             (par (:rps-round winner-ids (union co-ids loser-ids))
14                  (par-every [i loser-ids
15                              j (disj (union ids co-ids) i)]
16                    (close (:player i) (:player j)))))))))
```

a reusable synchronisation barrier, imported from 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. We clarify the following aspects:

- According to the specification (Fig. 12), in the first half of every round (lines 8–10), the items that are chosen by in-game threads are communicated among them. This is potentially problematic: as channels are unbuffered, sends and receives are blocking until reciprocal channel actions are performed, so unless threads collectively agree on a global order to perform reciprocal channel actions, deadlocks may occur. However, such global orders are hard to get right and brittle to maintain.
  An alternative solution is to use *selections*: in general, a selection consumes a list 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 selection performs *one* channel action from a list, depending on its enabledness at run time.
  In this particular example, instead of performing globally ordered reciprocal sends and receives, every thread performs a series of selections (`alts!!`) in the inner loop (Fig. 13, lines 17–24). Initially, the list of channel actions consists of all sends (`puts`) and receives (`takes`) that a thread needs to perform in a round. When a selection finishes, the channel action that was performed is removed from the list, and the inner loop continues. Because every thread behaves in this way, reciprocal channel actions will always be enabled.
- According to the specification (Fig. 12), 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 is potentially problematic: additional synchronisation is needed to ensure that "fast threads"—those that perform their channel actions early—wait for "slow threads" to catch up.
  To solve this, in this case study, we mix channels with a synchronisation barrier from `java.util.concurrent` (shaded code in Fig. 13). This demonstrates that channel-based programming abstractions (verified using `discourje`) can be mixed seamlessly with other concurrency libraries (not verified), which is common practice [6,41].

A monitor and instrumentation can be added to the implementation in the same way as shown in Fig. 7. In this case study, the implementation is safe relative to the specification.

3.4 Go fish

Our third case study is a program that simulates a game of Go Fish.[6] 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 covert interaction). This example demonstrates the following features:

[6] 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. Next, 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 [42]).

```
1  (def k ...) ;; number of threads (e.g., read from stdin)
2
3  (def rock "rock") (def paper "paper") (def scissors "scissors") ;; items
4
5  (def rock-or-paper-or-scissors (fn []    ...)) ;; returns an item
6  (def winner-ids                (fn [r]   ...)) ;; returns winners in round r
7  (def winner-or-loser?          (fn [r i] ...)) ;; returns true iff thread i is
8                                                  ;;   winner or loser in round r
9  (def chans   (mesh chan (range k)))
10 (def barrier (java.util.concurrent.Phaser. k))
11
12 (doseq [i (range k)]
13   (thread ;; (:player i)
14     (loop [ids (range k)]
15       (let [item (rock-or-paper-or-scissors)
16             opponent-ids (remove #{i} ids)
17             round (loop [acts (into (puts  chans [i item] opponent-ids)
18                                     (takes chans opponent-ids i))
19                         round {}] ;; map from ids to items (initially empty)
20                    (if (empty? acts)
21                      (assoc round i item)
22                      (let [[v c] (alts!! acts)]
23                        (recur (remove #{[c item] c} acts)
24                               (assoc round (putter-id chans c) v))))))]
25         (.arriveAndAwaitAdvance barrier)
26         (if (winner-or-loser? round i)
27           (do (.arriveAndDeregister barrier)
28               (doseq [j (remove #i (range k))]
29                 (close! (chans i j))))
30           (recur (winner-ids round)))))))
```

Fig. 13 Implementation of the Rock–Paper–Scissors program

- SPECIFICATION: user-defined data types; repetition; ordered-universal quantification; explicit non-determinism.
- IMPLEMENTATION: data type-based control flow.

3.4.1 Specification

A specification of Go Fish is shown in Fig. 14. Line 1 defines two roles, identified by `:dealer` (enacted by a single thread) and `:player` (multiple threads). Lines 3–29 define two protocols, identified by `:gf` and `:gf-turn`. Lines 30–35 define six user-defined data types.

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

1. `(:player j)` replies with the card that it was asked for, which happens to be the last card that `(:player i)` needed (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)` needed, 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.
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.

Specification `:gf` represents the whole game. It specifies 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. Next, 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 without involving `:dealer`.

```
1  (defrole :dealer) (defrole :player)
2
3  (defsession :gf [ids]
4    (cat (par-every [i ids]
5           (cat-every [_ (range 5)]
6             (--> Card :dealer (:player i))))
7         (alt-every [i ids]
8           (cat (--> Turn :dealer (:player i))
9                (:gf-turn i ids)))
10        (par-every [i ids]
11          (cat (close :dealer (:player i))
12               (par (cat (* (--> Card (:player i) :dealer))
13                         (close (:player i) :dealer))
14                    (par-every [j (disj ids i)]
15                      (close (:player i) (:player j))))))))
16
17 (defsession :gf-turn [i ids]
18   (alt-every [j (disj ids i)]
19     (cat (--> Ask (:player i) (:player j))
20          (alt (cat (--> Card (:player j) (:player i))
21                    (--> OutOfCards (:player i) :dealer))
22               (cat (--> Card (:player j) (:player i))
23                    (:gf-turn i ids))
24               (cat (--> Go (:player j) (:player i))
25                    (--> Fish (:player i) :dealer)
26                    (alt (--> Card :dealer (:player i))
27                         (--> OutOfCards :dealer (:player i)))
28                    (--> Turn (:player i) (:player j))
29                    (:gf-turn j ids))))))

30 (defrecord Turn [])
31 (defrecord Ask [suit rank])
32 (defrecord Card [suit rank])
33 (defrecord OutOfCards [])
34 (defrecord Go [])
35 (defrecord Fish [])
```

Fig. 14 Specification of the Go Fish program, including data types

3. Next, 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 (implementation detail).

3.4.2 Implementation

An implementation of Go Fish is shown in Fig. 15 (excerpt; many details are left out to save space). To demonstrate that `discourje` supports data type-based control flow, Fig. 15 shows fragments of code where values are received—directly with **`<!!`** and indirectly with **`alts!!`**—by threads that enact role `:player`. Specifically:

- On line 3, **`alts!!`** is used to receive a value `v` from another `:player` or from `:dealer`. This value is either of type `Turn`/`Ask` (received from another `:player`), 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 value of type `Card` or `Go` from `(:player j)`, to which a value of type `Ask` must have been sent previously (not shown).

A monitor and instrumentation can be added to the implementation in the same way as shown in Fig. 7. In this case study, the implementation is safe relative to the specification.

4 Theory of `discourje`

The `discourje` library is built on a formal foundation, inspired by process algebra (e.g. [43]) and multiparty session types (e.g. [16,17]). This underlying theory consists of a calculus of specifications (Sect. 4.1), a calculus of implementations (Sect. 4.2), and a simulation relation (Sect. 4.3). The aim of this section is to explain the general idea without excessive notation; in the interest of clarity, we therefore focus on the *basic fragments* of `discourje` and Clojure. These fragments consist of channel actions (sends, receives, closings, and selects), choice, and concatenation.

4.1 Specification calculus

Let $\mathbb{R}$ denote the set of *roles*, ranged over by p, q, r. Let $\mathbb{T} = \{\texttt{Bool}, \texttt{Nat}, \ldots\}$ denote the set of *types*, ranged over by t. Let $\mathbb{S}$ denote the set of *specifications*, ranged over by S; it is induced by the following grammar:

$$S ::= \boxed{1} \mid \underbrace{p \twoheadrightarrow q : t}_{\texttt{->}} \mid \underbrace{p \twoheadrightarrow\!\!\!\!\bullet\, q : t}_{\texttt{->>}} \mid \boxed{pq?t} \mid \underbrace{pq\bullet}_{\texttt{close}} \mid \underbrace{S_1 + S_2}_{\texttt{alt}} \mid \underbrace{S_1 \cdot S_2}_{\texttt{cat}} \mid \underbrace{S_1 \parallel S_2}_{\texttt{par}}$$

Term $p \twoheadrightarrow q : t$ specifies a *synchronous communication* of a value of type t through an unbuffered channel from p to q. Term $p \twoheadrightarrow\!\!\!\!\bullet\, q$ specifies an *asynchronous communication* of a value of type t through a buffered channel from p to q; the capacity of the buffer is left unspecified (implementation detail). Term $pq\bullet$ specifies a *closing* of a channel from p to q. Terms $S_1 + S_2$, $S_1 \cdot S_2$, and $S_1 \parallel S_2$ specify a *choice* (i.e. alternative composition), a *concatenation* (i.e. sequential composition), and an *interleaving* (i.e. parallel composition) of S_1 and S_2. The "boxed" terms (i.e. 1 and $pq?t$; the boxes are not part of the grammar) are auxiliary in the sense that they are used only to define the operational semantics below; there are no corresponding `discourje` macros. Term 1 specifies a *skip*; it can only terminate. Term $pq?t$ specifies the asynchronous receive of a value of type t through a buffered channel from p to q (when a send has already happened).

To formally define the operational semantics of specifications, let $\mathbf{\Sigma}$ denote the set of *type-level actions*, ranged over by σ; it is induced by the following grammar:

$$\sigma ::= pq⁉t \mid pq!t \mid pq?t \mid pq\bullet$$

Term $pq⁉t$ specifies a *synchronous send and receive* of a value of type t through an unbuffered channel from p to q. Terms $pq!t$ and $pq?t$ specify an *asynchronous send and receive* of a value of type t through a buffered channel from p to q. Term $pq\bullet$ specifies a closing of a channel from p to q.

The operational semantics of specifications is formally defined in terms of a termination predicate and a labelled reduction relation, denoted by $\downarrow$ and $\rightarrow$; they are induced by the rules in Fig. 16. The rules are standard in process algebra (e.g. [44]). We note that rule [S→-Buf] induces two reductions.

The *state machine* $[\![S]\!]$ of specification S is a triple (Q, q_0, Δ), where Q is a set of *states*, $q_0 \in Q$ is the *initial state*, and $\Delta \subseteq Q \times Q$ is the *transition relation*. Formally, Q is induced by the following rules:

$$\frac{}{(\bot, S) \in Q} \qquad \frac{(\sigma, S') \in Q \quad S' \xrightarrow{\sigma'} S''}{(\sigma', S'') \in Q}$$

Furthermore, $q_0 = (\bot, S)$ and $\Delta = \{((\varsigma, S'), (\sigma', S'')) \mid S' \xrightarrow{\sigma'} S''\}$, where ς ranges over $\mathbf{\Sigma} \cup \{\bot\}$. Thus, in state machines, states instead of transitions are labelled by actions (cf. Kripke structures). A *path* in a state machine $M = (Q, q_0, \Delta)$ is a sequence of states $q_1 \cdots q_n$, such that $(q_i, q_{i+1}) \in \Delta$ for every $1 \leq i < n$, and such that $(q_n, q') \notin \Delta$ for every $q' \in Q$; let $\mathsf{paths}(M, q)$ denote the set of all paths in M that start in q. (It suffices to restrict ourselves to finite paths here, as our specification calculus does not feature loops/recursion.)

State machines can be used to model-check specifications for temporal requirements expressed in *computation tree logic* (CTL) [45]. Let $\mathbf{\Phi}$ denote the set of *formulas*, ranged over by ϕ; it is induced by the following grammar:

$$\phi ::= \sigma \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \mathsf{AX}(\phi) \mid \mathsf{EX}(\phi) \mid \mathsf{AU}(\phi_1, \phi_2) \mid \mathsf{EU}(\phi_1, \phi_2)$$

Formula σ means that σ has just happened in the current state. Formulas $\neg\phi$ and $\phi_1 \vee \phi_2$ mean that the negation of ϕ and the disjunction of ϕ_1 and ϕ_2 are true in the current state. Formula $\mathsf{AX}(\phi)$ and (resp. $\mathsf{EX}(\phi)$) mean that ϕ is true in every (resp. some) next state. Formula $\mathsf{AU}(\phi_1, \phi_2)$ (resp. $\mathsf{EU}(\phi_1, \phi_2)$) means that ϕ_1 is true until ϕ_2 is true on every (resp. some) path that starts in the current state. We note that

Fig. 15 Implementation of the Go Fish program

```
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  ...)))))
```

$$\frac{}{1\downarrow}\ [\text{S}\downarrow\text{-Skip}] \qquad \frac{S_{i\in\{1,2\}}\downarrow}{S_1 + S_2\downarrow}\ [\text{S}\downarrow\text{-Alt}] \qquad \frac{S_1\downarrow \quad S_2\downarrow}{S_1 \cdot S_2\downarrow}\ [\text{S}\downarrow\text{-Seq}] \qquad \frac{S_1\downarrow \quad S_2\downarrow}{S_1 \parallel S_2\downarrow}\ [\text{S}\downarrow\text{-Par}]$$

(a) Termination

$$\frac{}{p \rightarrowtriangle q{:}t \xrightarrow{pq\text{?}t} 1}\ [\text{S}\rightarrow\text{-Unbuf}] \qquad \frac{}{p \twoheadrightarrow q{:}t \xrightarrow{pq!t} pq?t \xrightarrow{pq?t} 1}\ [\text{S}\rightarrow\text{-Buf}] \qquad \frac{}{pq\bullet \xrightarrow{pq\bullet} 1}\ [\text{S}\rightarrow\text{-Close}]$$

$$\frac{S_{i\in\{1,2\}} \xrightarrow{\sigma} S'}{S_1 + S_2 \xrightarrow{\sigma} S'}\ [\text{S}\rightarrow\text{-Alt}] \qquad \frac{S_1 \xrightarrow{\sigma} S_1'}{S_1 \cdot S_2 \xrightarrow{\sigma} S_1' \cdot S_2}\ [\text{S}\rightarrow\text{-Seq1}] \qquad \frac{S_1\downarrow \quad S_2 \xrightarrow{\sigma} S_2'}{S_1 \cdot S_2 \xrightarrow{\sigma} S_2'}\ [\text{S}\rightarrow\text{-Seq2}]$$

$$\frac{S_1 \xrightarrow{\sigma} S_1'}{S_1 \parallel S_2 \xrightarrow{\sigma} S_1' \parallel S_2}\ [\text{S}\rightarrow\text{-Par1}] \qquad \frac{S_2 \xrightarrow{\sigma} S_2'}{S_1 \parallel S_2 \xrightarrow{\sigma} S_1 \parallel S_2'}\ [\text{S}\rightarrow\text{-Par2}]$$

(b) Reduction

Fig. 16 Operational semantics of specifications

other common CTL operators (e.g. AF, EF, AG, EG) can be encoded as usual in CTL.

The semantics of formulas is formally defined in terms of an *entailment relation*, denoted by $\models$; it is induced by the rules in Fig. 17. The rules are standard (e.g. [46]).

4.2 Implementation calculus

Let $\mathbb{X} = \{\texttt{x}, \texttt{y}, \texttt{z}, \ldots\}$ denote the set of *variables*, ranged over by x. Let $\mathbb{C}$ denote the set of *channel identifiers*, ranged over by c. Let $\mathbb{V} = \mathbb{C} \cup \{\texttt{err}, \texttt{true}, \texttt{false}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots\}$ denote the set of *values*, ranged over by v. Let $\mathbb{E}$ denote the set of *expressions*, ranged over by e; it is induced by the following grammar:

$$e ::= v \mid \texttt{(=}\ e_1\ e_2\texttt{)} \mid \texttt{(not}\ e\texttt{)} \mid \texttt{(or}\ e_1\ e_2\texttt{)} \mid \texttt{(+}\ e_1\ e_2\texttt{)} \mid \cdots$$

Let $\mathbb{I}$ denote the set of *implementations*, ranged over by I, J; it is induced by the following grammar:

$$I, J ::= (\textbf{chan}\ e\ x).J \mid (\textbf{send}\ e_1\ e_2).J \mid (\textbf{recv}\ e\ x).J \mid \varnothing \mid \sum\mathcal{I} \mid \textbf{if}\ e\ I_1\ I_2 \mid I_1 \parallel I_2$$

Term $(\textbf{chan}\ e\ x).J$ implements the creation of a channel of capacity e, followed by J; the channel identifier (freshly generated) is bound to x in J. Term $(\textbf{send}\ e_1\ e_2).J$ implements the send of e_2 through the channel identified by e_1, followed by J. Term $(\textbf{recv}\ e\ x).J$ implements the receive of value through the channel identified by e, followed by J; the received value is bound to x in J. Term $\varnothing$ implements emptiness; it can only terminate. Term $\sum\mathcal{I}$, with $\mathcal{I} = \{I_1, \ldots, I_n\}$ for some $I_1, \ldots, I_n$, implements the nondeterministic selection of alternatives in $\mathcal{I}$. Term $\textbf{if}\ e\ I_1\ I_2$ implements the conditional choice of I_1 and I_2. Term $I_1 \parallel I_2$ implements the interleaving of I_1 and I_2 (i.e. parallel composition).

To formally define the operational semantics of implementations, we introduce the following auxiliary definitions:

- Let $\mathbf{I}$ denote the set of *value-level actions*, ranged over by ι; it is induced by the following grammar:

  $$\iota ::= pq\text{?}v \mid pq!v \mid pq?v \mid pq\bullet \mid \tau$$

  Term $pq\text{?}v$ implements a *synchronous send and receive* of v through an unbuffered channel from p to q. Terms $pq!v$ and $pq?v$ implement an *asynchronous send and receive* of v through a buffered channel from p to q. Term $pq\bullet$ implements a closing of a channel from p to q. Term τ implements any other action. We will use value-level actions as reduction labels.
- Let $\text{-}[\text{-}/\text{-}] : \mathbb{I}\times\mathbb{V}\times\mathbb{X} \to \mathbb{I}$ denote the *substitution function* for implementations (i.e. $I[v/x]$ denotes the substitution of v for every free occurrence of x in I). For instance, $((\textbf{chan}\ \texttt{(+ 5 x) y}).\varnothing)[\texttt{6}/\texttt{x}] = (\textbf{chan}\ \texttt{(+ 5 6) y}).\varnothing$. We will use substitution to bind values to variables.
- Let $\mathsf{eval} : \mathbb{E} \to \mathbb{V}$ denote the *evaluation function* for expressions (i.e. $\mathsf{eval}(e)$ denotes the evaluation of e). For instance, $\mathsf{eval}(\texttt{(+ 5 6)}) = \texttt{11}$. We stipulate that "bogus" expressions are evaluated to $\texttt{err}$. For instance, $\mathsf{eval}(\texttt{(+ 5 true)}) = \texttt{err}$. We will use evaluation to ensure that only values are communicated through channels.
- Let $\mathbb{C} \to (\{\top, \bot\} \times \{0, 1, 2, \ldots\} \times \mathbb{V}^*) \cup \{\bot\}$ denote the set of *network states*, ranged over by N. In words, every network state is a partial function from channel identifiers to *channel states* of the form $(b, n, \vec{w})$, where $b \in \{\top, \bot\}$ is the channel's status ($b{=}\top$ means open; $b{=}\bot$ means closed), n is the channel's capacity ($n{=}0$

$$\frac{}{(M,(\sigma,S'))\models\sigma}\ [\Phi\text{-Atom}] \qquad \frac{(M,q)\not\models\phi}{(M,q)\models\neg\phi}\ [\Phi\text{-Not}] \qquad \frac{(M,q)\models\phi_{i\in\{1,2\}}}{(M,q)\models\phi_1\vee\phi_2}\ [\Phi\text{-Or}]$$

$$\frac{(M,q')\models\phi \text{ for every } (q,q')\in\Delta}{(M,q)\models\mathsf{AX}(\phi)}\ [\Phi\text{-AllNext}] \qquad \frac{(M,q')\models\phi \text{ for some } (q,q')\in\Delta}{(M,q)\models\mathsf{EX}(\phi)}\ [\Phi\text{-ExistsNext}]$$

$$\frac{\begin{array}{c}(M,q_1)\models\phi_1 \text{ and } \cdots \text{ and } (M,q_{i-1})\models\phi_1 \text{ and } (M,q_i)\models\phi_2\\ \text{for some } 1\le i\le n, \text{ for every } q_1\cdots q_n\in\mathsf{paths}(M,q)\end{array}}{(M,q)\models\mathsf{AU}(\phi_1,\phi_2)}\ [\Phi\text{-AllUntil}]$$

$$\frac{\begin{array}{c}(M,q_1)\models\phi_1 \text{ and } \cdots \text{ and } (M,q_{i-1})\models\phi_1 \text{ and } (M,q_i)\models\phi_2\\ \text{for some } 1\le i\le n, \text{ for some } q_1\cdots q_n\in\mathsf{paths}(M,q)\end{array}}{(M,q)\models\mathsf{EU}(\phi_1,\phi_2)}\ [\Phi\text{-ExistsUntil}]$$

Fig. 17 Semantics of CTL formulas, where $M=(Q,q_0,\Delta)$ is a state machine

means unbuffered; $n>0$ indicates buffered), and $\vec{w}\in\mathbb{V}^*$ is the channel's content as a list of buffered values that are in transit, from left to right ($\vec{w}=\varepsilon$ means empty). Regarding notation, we write $N[c\mapsto(b,n,\vec{w})]$ instead of $\{c'\mapsto N(c')\mid c'\in\mathbb{C}\setminus\{c\}\}\cup\{c\mapsto(b,n,\vec{w})\}$.

The operational semantics of implementations is formally defined in terms of a labelled reduction relation, denoted by $\rightarrow$, over *configurations* of the form (I,N); it is induced by the rules in Fig. 18. In words:

- Rule [I-Chan] states that if c is fresh channel identifier (left premise), and if the value of e represents n (right premise),[7] then a channel identified by c can be created in the network (conclusion).
- Rule [I-Send1] states that if c identifies an open channel (left premise), and if the channel is buffered and non-full (right premise), then the value of e can be asynchronously sent through the channel by enqueueing it to the back of the buffer (conclusion). Rule [I-Send2] states that if I is one of the alternatives (left premise), and if the network state allows I to reduce with an asynchronous send (right premise), then I can be selected (conclusion).
- Rule [I-Recv1] states that if c identifies a non-empty channel (premise), then v can be asynchronously received through the channel by dequeuing it from the front of the buffer (conclusion). Rule [I-Recv2] states that if I is one of the alternatives (left premise), and if the network state allows I to reduce with an asynchronous receive (right premise), then I can be selected (conclusion).
- Rule [I-SendRecv1], [I-SendRecv2], [I-SendRecv3], and [I-SendRecv4] state that if c identifies an open, unbuffered channel, then the value of e can be synchronously sent and received through the channel.
  We note that these four rules can be reformulated using two separate rules for sending (similar to [I-Send1] and [I-Send2]), two for receiving (similar to [I-Recv1] and [I-Recv2]), and one to synchronise these actions. However, this would require an auxiliary reduction relation, while the total number of rules is higher.
- Rules [I-If1] and [I-If2] are standard.
- Rule [I-Par1] states that if I_1 can reduce (premise), then the interleaving of I_1 and I_2 can reduce accordingly. Rules [I-Par2] and [I-Par3] state that interleaving is commutative and associative.

A *run* $\rightarrow_I$ of implementation I is a subset of $\rightarrow$ such that:

- There exist ι, I', N' such that $(I,\emptyset)\xrightarrow{\iota}_I(I',N')$. That is, the run has a proper initial configuration.
- If $(I',N')\xrightarrow{\iota_1}_I(I_1'',N_1'')$ and $(I',N')\xrightarrow{\iota_2}_I(I_2'',N_2'')$, then $\iota_1=\iota_2$ and $I_1''=I_2''$ and $N_1''=N_2''$. That is, every configuration in the run has a unique successor.

We note that we do not require runs to be complete, as we also want to verify the safety of partial runs that are not finished yet, but which are safe so far.

4.3 Verification

To formally define safety, we introduce the following auxiliary definitions:

- Let $\mathbb{C}\rightarrow(\mathbb{R}\times\mathbb{R})\cup\{\bot\}$ denote the set of *instrumentations*, ranged over by †. In words, every instrumentation is a partial function from channel identifiers to pairs of roles of the form pq, where p is the intended sender and q is the intended receiver. The idea is that every † establishes links between channel references in an imple-

[7] To be formally precise, we make a distinction between numeric values in $\mathbb{V}$ (i.e. $0,1,2,\ldots$) and "actual" numbers (i.e. $0,1,2,\ldots$). For instance, numeric value 5 represents "actual" number 5.

Fig. 18 Operational semantics of implementations

$$\frac{N(c) = \bot \quad \mathsf{eval}(e) \text{ represents } n}{((\mathbf{chan}\ e\ x)\,.\,J, N) \xrightarrow{\tau} (J[c/x], N[c \mapsto (\top, n, \varepsilon)])}\ [\text{I-Chan}]$$

$$\frac{N(c) = (\top, \vec{w}, n) \quad 0 < |\vec{w}| < n}{((\mathbf{send}\ c\ e)\,.\,J, N) \xrightarrow{c!\mathsf{eval}(e)} (J, N[c \mapsto (\top, \mathsf{eval}(e)\vec{w}, n)])}\ [\text{I-Send1}]$$

$$\frac{I \in \mathcal{I} \quad (I, N) \xrightarrow{c!v} (I', N')}{(\sum \mathcal{I}, N) \xrightarrow{c!v} (I', N')}\ [\text{I-Send2}]$$

$$\frac{N(c) = (b, \vec{w}v, n)}{((\mathbf{recv}\ c\ x)\,.\,J, N) \xrightarrow{c?v} (J[v/x], N[c \mapsto (b, \vec{w}, n)])}\ [\text{I-Recv1}]$$

$$\frac{I \in \mathcal{I} \quad (I, N) \xrightarrow{c?v} (I', N')}{(\sum \mathcal{I}, N) \xrightarrow{c?v} (I', N')}\ [\text{I-Recv2}]$$

$$\frac{N(c) = (\top, \varepsilon, 0)}{(((\mathbf{send}\ c\ e)\,.\,J_1) \parallel ((\mathbf{recv}\ c\ x)\,.\,J_2), N) \xrightarrow{c?\mathsf{eval}(e)} (J_1 \parallel J_2[\mathsf{eval}(e)/x], N)}\ [\text{I-SendRecv1}]$$

$$\frac{(\mathbf{recv}\ c\ x)\,.\,J_2 \in \mathcal{I}_2 \quad N(c) = (0, \top, \varepsilon)}{((\mathbf{send}\ c\ e)\,.\,J_1 \parallel \sum \mathcal{I}_2, N) \xrightarrow{c?\mathsf{eval}(e)} (J_1 \parallel J_2[\mathsf{eval}(e)/x], N)}\ [\text{I-SendRecv2}]$$

$$\frac{(\mathbf{send}\ c\ e)\,.\,J_1 \in \mathcal{I}_1 \quad N(c) = (0, \top, \varepsilon)}{(\sum \mathcal{I}_1 \parallel (\mathbf{recv}\ c\ x)\,.\,J_2, N) \xrightarrow{c?\mathsf{eval}(e)} (J_1 \parallel J_2[\mathsf{eval}(e)/x], N)}\ [\text{I-SendRecv3}]$$

$$\frac{(\mathbf{send}\ c\ e)\,.\,J_1 \in \mathcal{I}_1 \quad (\mathbf{recv}\ c\ x)\,.\,J_2 \in \mathcal{I}_2 \quad N(c) = (0, \top, \varepsilon)}{(\sum \mathcal{I}_1 \parallel \sum \mathcal{I}_2, N) \xrightarrow{c?\mathsf{eval}(e)} (J_1 \parallel J_2[\mathsf{eval}(e)/x], N)}\ [\text{I-SendRecv4}]$$

$$\frac{\mathsf{eval}(e) = \mathtt{true}}{(\mathbf{if}\ e\ I_1\ I_2, N) \xrightarrow{\tau} (I_1, N)}\ [\text{I-If1}] \qquad \frac{\mathsf{eval}(e) = \mathtt{false}}{(\mathbf{if}\ e\ I_1\ I_2, N) \xrightarrow{\tau} (I_2, N)}\ [\text{I-If2}]$$

$$\frac{(I_1, N) \xrightarrow{\iota} (I_1', N)}{(I_1 \parallel I_2, N) \xrightarrow{\iota} (I_1' \parallel I_2, N)}\ [\text{I-Par1}]$$

$$\frac{(I_2 \parallel I_1, N) \xrightarrow{\iota} (I', N')}{(I_1 \parallel I_2, N) \xrightarrow{\iota} (I', N')}\ [\text{I-Par2}] \qquad \frac{((I_1 \parallel I_2) \parallel I_3, N) \xrightarrow{\iota} (I', N')}{(I_1 \parallel (I_2 \parallel I_3), N) \xrightarrow{\iota} (I', N')}\ [\text{I-Par3}]$$

mentation (characterised by their identifiers) and channel references in a specification (characterised by roles).

- Let $:_\dagger \subseteq \mathbf{I} \times \mathbf{\Sigma}$ denote the $\dagger$-*compliance relation* between type-level actions and value-level actions; it is induced by the following rules:

$$\frac{v \text{ is of type } t}{c!v :_\dagger \dagger(c)!t} \qquad \frac{v \text{ is of type } t}{c?v :_\dagger \dagger(c)?t} \qquad \frac{v \text{ is of type } t}{c!v :_\dagger \dagger(c)!t}$$

In words, the rules state that an action implementation ι complies with an action specification σ if: (1) the channel identified by c in ι is linked by $\dagger$ to the intended sender and the intended receiver that occur in σ; (2) the value that occurs in ι is of the type that occurs in σ.

Safety ("bad channel actions never happen") is formally defined in terms of *weak simulation* (e.g. [47]). More precisely, given instrumentation $\dagger$, a run $\rightarrow_I$ of implementation I is $\dagger$-safe relative to specification S, if there exists a binary simulation relation $\preceq$ such that:

- $(I, \emptyset) \preceq S$
- If $(I', N') \preceq S'$ and $(I', N') \xrightarrow{\iota}_I (I'', N'')$ and $\iota \neq \tau$, then there exist σ, S'' such that $(I'', N'') \preceq S''$ and $S' \xrightarrow{\sigma} S''$ and $\iota :_\dagger \sigma$.
- If $(I', N') \preceq S'$ and $(I', N') \xrightarrow{\tau}_I (I'', N'')$, then $(I'', N'') \preceq S'$.

In words, $(I', N') \preceq S'$ iff S' can reduce accordingly to S'' whenever (I', N') can reduce to I'' (and (I'', N'') and S'' are

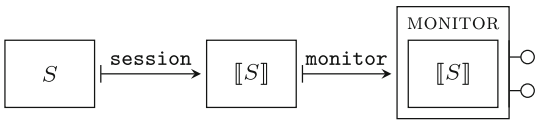


Fig. 19 `discourje.spec`

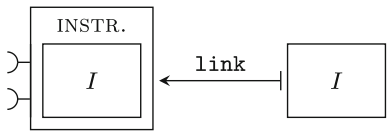


Fig. 20 `discourje.core.async`

again related by $\preceq$), modulo τ-reductions. That is, (I', N') can be mimicked by S', *coinductively*.

5 Practice of `discourje`

In this section, we present two practical aspects of the `discourje` library. First, we explain the main components and their internals in more detail (Sect. 5.1). Next, we present performance experiments using both microbenchmarks and whole-program benchmarks (Sect. 5.2).

5.1 The library

The `discourje` library consists of three main components, each of which corresponds with an activity in the intended workflow (Fig. 4):

- `discourje.core.spec` is a sublibrary to write specifications (Sect. 5.1.1).
- `discourje.core.lint` is a sublibrary to check specifications (Sect. 5.1.2).
- `discourje.core.async` is a sublibrary to write implementations (Sect. 5.1.3).

5.1.1 Writing specifications: `discourje.core.spec`

Sublibrary `discourje.core.spec` consists of macros to write specifications (cf. syntax of the specification calculus; Sect. 4.1); data structures to represent specifications as state machines (cf. operational semantics of the specification calculus); and functions to instantiate these data structures and construct monitors. The idea is visualised in Fig. 19: first, the programmer writes a specification S using the macros; next, at run time, function `spec` is applied to S to expand and evaluate the macros to a state machine $[\![S]\!]$; next, function `monitor` is applied to $[\![S]\!]$ to create a monitor.

The monitor provides two operations, depicted as "lollipops" in Fig. 19: *verifying* if a given channel action ι is allowed in current state q of $[\![S]\!] = (Q, q_0, \Delta)$ (formally: given instrumentation $\dagger$, check if there exist σ, S' such that $(q, (\sigma, S')) \in \Delta$ and $\iota :_\dagger \sigma$), and subsequently *updating* the current state of $[\![S]\!]$ to a successor. In this way, effectively, the monitor builds a simulation relation to ensure safety (Sect. 4.3), incrementally, as channel actions are performed. We note that operations verify and update happen atomically, using lock-free synchronisation (compare-and-set): an update happens only if both verification succeeded and there has been no update in the meantime. Besides this base functionality, `discoure.core.spec` also offers the following extensions:

- *Non-determinism* To support non-deterministic specifications, the monitor maintains *a set* of possible current states $\{q_1, \ldots, q_n\}$ instead of a single state. To verify if channel action ι is allowed, the monitor iterates over all states in the set to find at least one of them that has a corresponding transition (formally: given instrumentation $\dagger$, check if there exist i, σ, S' such that $(q_i, (\sigma, S')) \in \Delta$ and $\iota :_\dagger \sigma$). If so, to subsequently update the set of current states, the monitor collects all possible successors (formally: $\{(\sigma, S') \mid (q, (\sigma, S')) \in \Delta \text{ and } \iota :_\dagger \sigma\}$). In this way, essentially, the state machine is *determinised* using an on-the-fly power set construction.
- *Incremental generation* Instead of generating the whole state machine for S upfront, the monitor can also generate it incrementally, by need. This is advantageous if only a small portion of the state machine is actually needed.

5.1.2 Checking specifications: `discourje.core.lint`

Sublibrary `discourje.core.lint` consists of functions to validate generic sanity checks (Sect. 3.1) and protocol-specific temporal requirements. The core of `discourje.core.lint` is a custom-built model checker for CTL. The idea is to: first, define intended requirements of a specification S as CTL formulas (Fig. 17); next, compute state machine $[\![S]\!]$; next, invoke a classical CTL model checking algorithm [48]. Besides this base functionality, `discourje.core.lint` also offers the following extensions:

- *Batch mode* When asked to batch-check multiple formulas, the model checker reuses the state machine and bookkeeping information across formulas, to avoid double work. Notably, the generic sanity checks are performed in batch mode to improve performance.
- *Past-time operators* CTL allows the programmer to express requirements in terms of properties of the future. However, in our experience, many requirements are more naturally expressed in terms of properties of the past. For instance: "if a channel is closed, then it must have been used before" (i.e. one of the generic san-

ity checks). Therefore, `discourje.core.lint` also supports *Past CTL* (with branching past) [49].

- *Witness generation* To use `discourje.core.lint` effectively for debugging, proper diagnostics must be included when an issue is reported. Therefore, `discourje.lint` can generate witnesses that serve as counterexamples of a CTL formula. As usual for CTL (e.g. [50]), our witness generator works only for the universal fragment of CTL.
- *API* Using an extra API (in Clojure), custom atomic propositions and temporal patterns can be written to extend the core. We used this feature to write the generic causality check, as it cannot be easily expressed using only the standard atomic propositions.

5.1.3 Running implementations: `discourje.core.async`

Sublibrary `discourje.core.async` consists of functions that serve as *proxies* for functions and macros **`thread`** (new thread), **`chan`** (new channel), **`close!`** (closing), **`>!!`** (send), **`<!!`** (receive), and **`alts!!`** (select) in `clojure.core.async`. The idea is visualised in Fig. 20: first, the programmer writes an implementation I; next, at run time, function `link` is applied to the channels in I to create instrumentation. More precisely, function `link` associates a channel with an intended sender, intended receiver, and monitor; it is the practical embodiment of function † (Sect. 4.3). We emphasise that no other changes to I are needed: as the signatures of the supported macros and functions in `clojure.core.async` (listed above) are identical to their proxies in `discourje.core.async`, adding instrumentation in this way is non-invasive and nearly effortless.

In more detail, the proxies of **`>!!`**, **`<!!`**, and **`close`** in `discourje.core.async` work as follows. When one of these functions is invoked, first, it waits until the underlying channel c is *ready* for the operation: in case of a send or receive through an unbuffered channel, a reciprocal receive or send needs to be pending; in case of a send or receive through a buffered channel, the buffer needs to be non-full or non-empty. Next, at time t_1, the monitor linked to c is requested to verify if the attempted send, receive, or closing is allowed. If yes, at time t_2, the monitor is requested to update accordingly and the attempted send, receive, or closing actually takes effect (i.e. a value is synchronously exchanged or asynchronously enqueued/dequeued); if no, an exception is thrown. If, between t_1 and t_2, multiple threads request the monitor to update, only one will succeed; the others need to retry from the start. In this way, safety violations are detected in a way that is both sound (i.e. if an exception is thrown, the violating action really was not allowed) and complete (i.e. if no exception is thrown, all actions were really allowed).

Finally, we note that *Java interoperability* is supported. That is, to leverage the fact that Clojure compiles to Java bytecode and runs on the JVM, we also wrote a thin Java wrapper around `discourje.core.async`, so Java programmers can easily use channels and have them monitored from inside their Java programs, regardless of the threading mechanism (e.g. classical Java threads, thread pools, or parallel streams can be used).

5.2 Performance experiments

From the outset, we had two intended usage types of the `discourje` library:

- *Usage type A* As a *testing/debugging tool* for concurrent programs in development, to find/diagnose communication-related concurrency bugs.
- *Usage type B* As a *fail-safe mechanism* for concurrent programs in production, to prevent propagation of spurious results caused by concurrency bugs to end-users (i.e. it is often preferable to throw a runtime error).

A key factor that determines `discourje`'s fitness for purpose is efficiency. We therefore conducted two kinds of performance experiments: microbenchmarks to study the *scalability* of `discourje` (Sect. 5.2.1) and whole-program benchmarks to study the *overhead* relative to unmonitored code (Sect. 5.2.2).

In all experiments, we used a machine with 32 physical cores and 64 GB of physical memory (far more than needed for our benchmarks), using CentOS Linux 8 (kernel: 4.18) and Java 16.0.1 (HotSpot JVM) with default settings.

5.2.1 Microbenchmarks

In the microbenchmarks, we studied `discourje`'s scalability under "extreme" circumstances in which threads perform *only* sends and receives, without any real computations; this is the worst-case scenario for the lock-free algorithm to synchronise monitor access, as it gives rise to maximal thread contention.

We considered six basic protocols to investigate the core features of the specification language in isolation. The specifications are shown in Fig. 21; their relevant properties are summarised in Table 1 (discussed below).

Specifications `:ring-unbuffered` and `:ring-buffered` combine concatenation with synchronous and asynchronous communication. Specifications `:star-un-buffered-out-wards` and `:star-un-buffered-in-wards` combine choice with synchronous communication; their state machines have only a single state (with an outgoing transition for every `:worker` thread). Specifications `:star-buffered-out-wards` and `:star-`

Fig. 21 Specifications of microbenchmark protocols

```
1  (defrole :worker)
2  (defrole :master)
3
4  (defsession :ring-unbuffered [k]
5    (* (cat-every [i (range k)]
6         (--> Boolean (:worker i) (:worker (mod (inc i) k))))))
7
8  (defsession :ring-buffered [k]
9    (* (cat-every [i (range k)]
10        (-->> Boolean (:worker i) (:worker (mod (inc i) k))))))
11
12 (defsession :star-unbuffered-outwards [k] ;; one-to-many
13   (* (alt-every [i (range k)]
14        (--> Boolean :master (:worker i)))))
15
16 (defsession :star-unbuffered-inwards [k] ;; many-to-one
17   (* (alt-every [i (range k)]
18        (--> Boolean (:worker i) :master))))
19
20 (defsession :star-buffered-outwards [k] ;; one-to-many
21   (par-every [i (range k)]
22     (* (-->> Boolean :master (:worker i)))))
23
24 (defsession :star-buffered-inwards [k] ;; many-to-one
25   (par-every [i (range k)]
26     (* (-->> Boolean (:worker i) :master))))
```

Table 1 Properties of the benchmark protocols

	Pattern	#States	#Trans/state
`:ring-unbuffered`	Ring	Linear	Constant
`:ring-buffered`	Ring	Linear	Constant
`:star-un-buffered-out-wards`	One-to-many	Constant	Linear
`:star-un-buffered-in-wards`	Many-to-one	Constant	Linear
`:star-buffered-out-wards`	One-to-many	Exponential	Linear
`:star-buffered-in-wards`	Many-to-one	Exponential	Linear

`buffered-in-wards` combine interleaving with asynchronous communication; their state machines have exponentially many states due to the combinatorial explosion of the orders in which the communications can be interleaved. Each of these specifications consists of a loop with an unspecified number of iterations (`*`). In every iteration of `:ring-unbuffered` and `:ring-buffered`, the roles need to communicate according to a *ring* pattern; in every iteration of `:star-un-buffered-out-wards` and `:star-un-buffered-in-wards`, the roles need to communicate according to a *one-to-many* pattern; in every iteration of `:star-un-buffered-in-wards` and `:star-buffered-in-wards`, the roles need to communicate according to a *many-to-one* pattern.

We ran every implementation of these protocols with $k \in \{2, 4, 6, 8, 10, 12, 14, 16\}$ `:worker` threads,[8] for 4096 loop iterations, and measured the run times. For every implementation, for every protocol, and for every k, we repeated the run 30 times to smooth out variability, on separate "cold" instances of the JVM to rule out JIT impact across repetitions. We computed the mean m, standard deviation s, and coefficient of variation $\frac{s}{m}$. The means are shown in Fig. 22; the coefficients of variation were all less than 10%, so the general trends are informative.

To explain the general trends, we model the total run time t of an implementation in terms of its two dominant components, using the following equation: $t = t_{\text{mach}} + t_{\text{act}}$, where

[8] For the ring protocols, the total number of threads is k; for the one-to-many/many-to-one protocols, the total number of threads is $k+1$ (including the master thread).

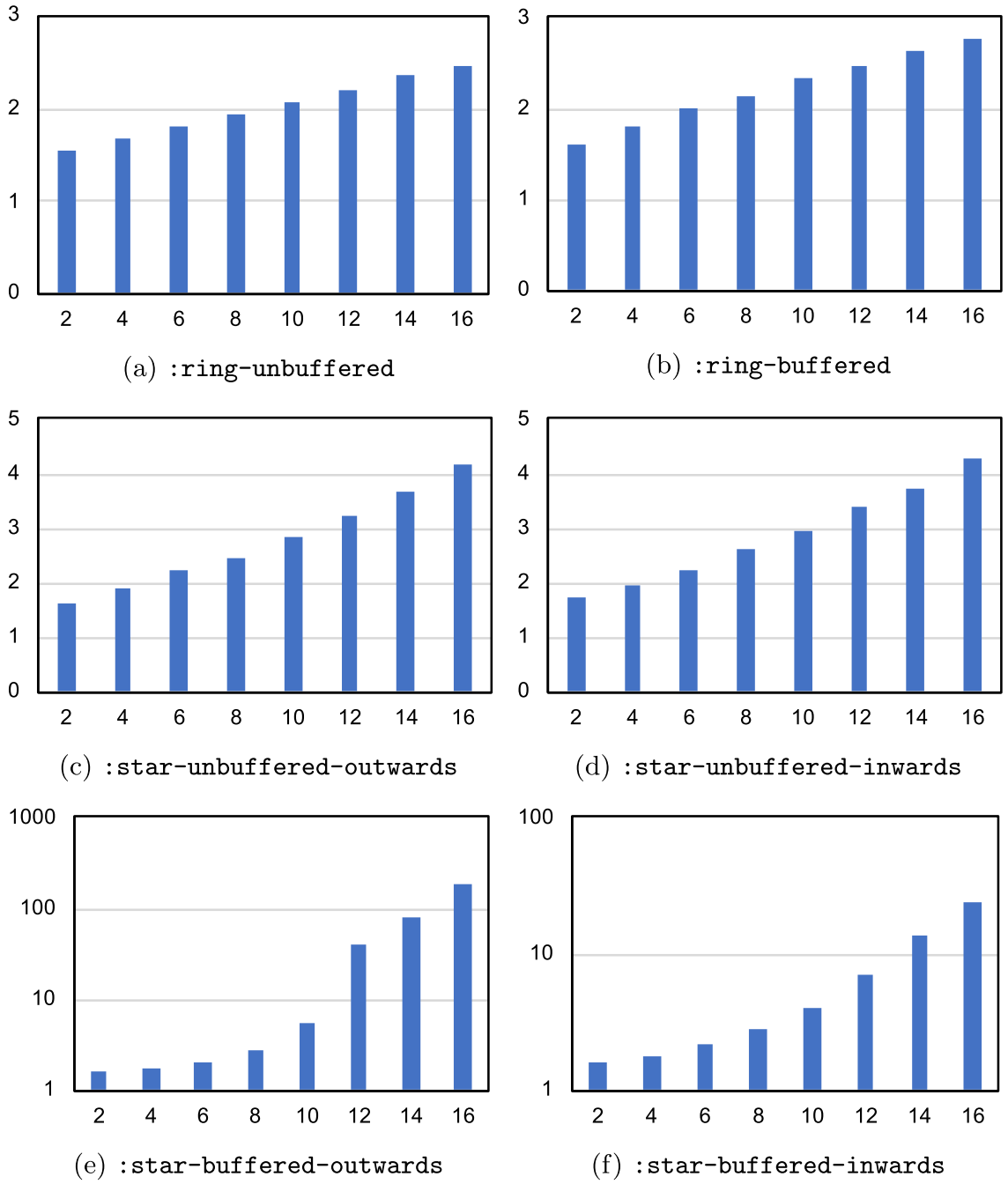


Fig. 22 Microbenchmarks: run times in seconds (y-axis) as the number of worker threads increases (x-axis), as a measure of scalability

t_{mach} is the time required to compute the state machine for the specification, and t_{act} is the time required to perform all sends and receives. Using this model, we summarise the main findings as follows:

- We observe *linear scalability* for `:ring-unbuffered`, `:ring-buffered`, `:star-un-buffered-out-wards`, and `:star-un-buffered-in-wards`.

  To explain this, we first note that the number of states (column "#states" in Table 1) and the number of transitions per state (column "#trans/state") grow linearly in k for these specifications, so t_{mach} grows linearly in k too. We also note that the number of sends and receives grows linearly in k, so t_{act} grows linearly in k too. Thus, $t = t_{\text{mach}} + t_{\text{act}}$ grows linearly in k.
- We observe *exponential scalability* for `:star-buffered-out-wards` and

`:star-buffered-in-wards` (i.e. the scale on the y-axis is logarithmic).
To explain this, we note that the number of states (column "#states" in Table 1) grows exponentially in k, so t_{mach} grows exponentially in k too. Thus, $t = t_{\text{mach}} + t_{\text{act}}$ grows exponentially in k.

We note that we use equation $t = t_{\text{mach}} + t_{\text{act}}$ only as a model to explain the general trends; we have not measured t_{mach} and t_{act} separately.

To conclude, `:ring-unbuffered`, `:ring-buffered`, `:star-un-buffered-out-wards`, and `:star-un-buffered-in-wards` enjoy fine scalability. However, scalability of `star-buffered-out-wards` and `:star-buffered-in-wards` can be improved.

5.2.2 Whole-program benchmarks

In the whole-program benchmarks, we studied `discourje`'s overhead in five real(istic), existing concurrent programs:

- *Chess* Simulates a game of chess between two player threads.
- *Conjugate Gradient* (*CG-k*) Computes an estimate of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of nonzeros, using the conjugate gradient algorithm, with k worker threads.
- *Fourier Transform* (*FT-k*) Computes the solution of a partial differential equation, using the forward and inverse Fast Fourier Transform algorithm, with $2{\cdot}k$ worker threads.
- *Integer Sort* (*IS-k*) Computes a sorted list of uniformly distributed integer keys, using histogram-based integer sorting, with k worker threads.
- *Multi-Grid* (*MG-k*) Computes an approximate solution u to the discrete Poisson problem $\nabla^2 u = v$, using the V-cycle multigrid algorithm, with $4{\cdot}k$ worker threads.

For Chess, we used Clojure code similar to the threads in Tic–Tac–Toe (Fig. 11), combined with invocations of the open source chess engine Stockfish (https://stockfishchess.org) to compute moves. For CG, FT, IS, and MG, we adapted existing Java implementations from the *NAS parallel benchmarks* (NPB) [51] suite, which consists of computational fluid dynamics kernels, by taking advantage of our Java interoperability wrapper (Sect. 5.1.3) to replace the monitor-based synchronisation used in the original versions.

We also wrote specifications using `discourje`. For Chess, the specification is the same as the Tic-Tac-Toe specification (Fig. 10); for CG, FT, IS, and MG, the specifications consist of repetitions of buffered one-to-many and many-to-one patterns (Fig. 21), involving various subsets of worker threads and data types. From a communication perspective, the key difference between CG, FT, IS, and MG is the *frequency* in which repetitions of the one-to-many and many-to-one patterns happen (i.e. communication intensity).

We recorded execution times of each of the implementations without and with monitoring enabled, using standardised computational workloads. For Chess, the workload is controlled by the total amount of time each player has to compute its moves during the entire game; we used the four smallest such workloads supported by the open source chess server Lichess (https://lichess.org), namely {15, 30, 45, 60} seconds, and we limited games to a maximum of 40 turns per player (*UltraBullet chess*). Furthermore, we allow simultaneous "ponder" computations by a player during its opponent's turn, so there is ample parallelism as well. For CG, FT, IS, and MG, the workload is controlled by the input size; we used the standardised inputs that are predefined by NPB.

For every implementation (without and with monitoring enabled), and for every k, we repeated the run 30 times to smooth out variability, and we computed the mean m, standard deviation s, and coefficient of variation $\frac{s}{m}$. All coefficients of variation were smaller than 10%, except for CG-14 (15%) and CG-16 (18%), so the general trends are informative. As a measure of overhead, we computed normalised means $\frac{\mu_{\text{w}}}{\mu_{\text{wo}}}$, where μ_{w} and μ_{wo} are mean run times with and without monitoring enabled; this metric is a dimensionless number that indicates the factor by which monitoring *slows down* the implementation. The normalised means for Chess are 0.979 (15 seconds), 0.999 (30 seconds), 0.996 (45 seconds), and 0.996 (60 seconds); the normalised means for NPB are shown in Fig. 23. We summarise the main findings as follows, relative to intended usage types A and B of the `discourje` library (page 29):

- For Chess, the normalised means are all very close to 1, which indicates that the overhead of monitoring is negligible. This suggests that intended usage types A and B are both possible for Chess.
- For FT and IS, the slowdowns are all less than 12%. This seems low enough not only for usage type A (testing/debugging in development), but also usage type B (fail-safe mechanism in production).
  We also note that the specifications for FT and IS are (extended versions of) specifications `:star-buffered-in-wards` and `:star-buffered-out-wards`, which scaled poorest in the microbenchmarks. This shows that despite poor scalability under the "extreme" circumstances in the microbenchmarks (only sends and receives; no computations), `discourje` can still perform reasonably well in whole programs.

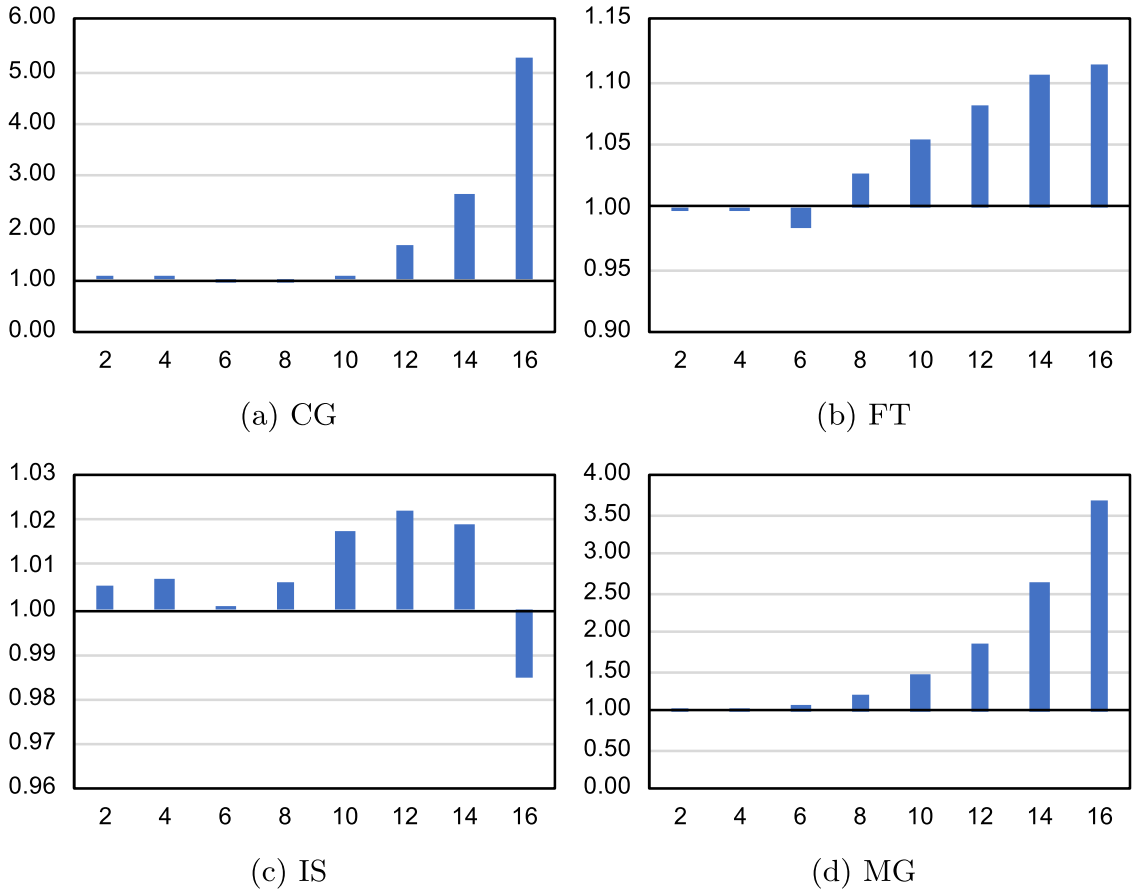


Fig. 23 NPB: slowdown of running implementations with monitoring enabled relative to running them without monitoring enabled (y-axis) as the number of worker threads increases (x-axis), as a measure of overhead

- For CG and MG, the slowdowns are higher: up to 5.3× and 3.7×, respectively. Although this is likely to be too much for usage type B, it seems low enough for usage type A (cf. the industrial-strength Valgrind tool for dynamic analysis of memory management [52], which can inflict similar slowdowns but is nevertheless effectively used in practice).
  The difference in performance between {FT, IS} and {CG, MG} may be explained by the fact the latter are considerably more communication-intensive than the former, so the overhead of monitoring communications is more pronounced.

6 Conclusion

We presented Discourje: a research project that aims to help programmers cope with channels and concurrency bugs in Clojure, based on dynamic analysis. That is: Discourje offers a run-time verification library in Clojure, called `discourje`, to ensure safety of channel actions in implementations relative to specifications. The formal foundations of `discourje` are based on multiparty session types, but trade in static type checking for dynamic run-time monitoring; a key advantage is higher expressiveness.

An important design principle of `discourje` has been ergonomics: we aim to make `discourje`'s usage as comfortable as possible. In particular, programmers can decide to start using `discourje` at any stage of development (and doing so requires little effort); `discourje` is itself implemented in Clojure (so there is no need to use a different IDE, learn completely new syntax, or install special compilers); and `discourje` can be used seamlessly alongside other concurrency libraries. Furthermore, results in performance experiments indicate that run time overhead can be less than 12% for real(istic), existing concurrent programs. This makes `discourje` suitable both as a testing/debugging tool in development and as a fail-safe mechanism in production.

We close this paper with an overview of related work (Sect. 6.1) and future work (Sect. 6.2).

6.1 Related work

As explained in Sect. 1.1.2, the Discourje project was originally conceived to explore a new direction in research on multiparty session types (MPST). In recent years, several

practical tools were developed, mostly for statically typed languages (e.g. F# [26], Go [7], Java [27,28], Scala [29]), and to lesser extent for dynamically typed languages (e.g. Python [53], Erlang [54]). To our knowledge, in the context of MPST, the Discourje project is the first to leverage run-time verification and decomposition-free verification *together* for a dynamically typed language (Fig. 1), although these characteristics have been considered in isolation:

- There are MPST approaches that combine static type checking with a form of distributed run-time verification and/or assertion checking [19,26,55–57]. In contrast to Discourje, however, these dynamic techniques still rely on decomposition, which negatively affects their expressiveness (e.g. none of the case studies in Sects. 3.2–3.4 are supported).
- Decomposition-free MPST has also been explored by López et al. [58,59]. The 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 [60] and Why3 [61]). However, this approach requires considerable manual effort. In contrast, `discourje` can be used in a fully automated way.

Expressiveness of MPST has been an important research topic in recent years, but efforts have primarily been geared towards adding more advanced features (e.g. time [18,19], security [20–23], and parametrisation [7,24,25]); in contrast, restrictions on the usage of core features such as choice and interleaving have remained, even though they limit MPST's applicability in practice (e.g. none of the case studies in Sects. 3.2–3.4 are supported). Some work has been done to improve expressiveness in this regard using static techniques [62], but the specification language of `discourje` remains more expressive.

Verification of shared-memory concurrency with channels has received attention in the context of Go [8–11]. However, in addition to relying on static techniques, 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 `discourje` (e.g. none of the case studies in Sects. 3.2–3.4 are supported).

We are aware of only two other works that use formal techniques to reason about Clojure programs: Bonnaire-Sergeant et al. [63] formalised the optional type system for Clojure and proved soundness, while Pinzaru et al. [64] developed a translation from Clojure to Boogie [65] to verify Clojure programs annotated with pre/post-conditions. Discourje seems the first research project to target concurrency in Clojure.

6.2 Future work

We aim to improve `discourje` along the following lines:

- *Recovery* We aim to explore the idea that whenever a monitor detects a safety violation, instead of throwing an exception, it should *delay* the violating action as a corrective measure, in an attempt to steer the implementation towards safety. When done naively, such delays can easily give rise to deadlocks, so our plan is to combine this approach with run-time model checking/reachability analysis to ensure that *eventually*, the violating action will be allowed (if yes, delay; if no, throw).
- *Scalability* Our microbenchmarks show that we need better ways to deal with specifications with exponentially sized state machines. Our plan is to study new forms of flexible decomposition that allow us to compute local specifications as in traditional MPST (Fig. 1) to avoid exponential blow-up whenever possible, but without compromising expressiveness (by keeping a centralised component, like the current monitors, if needed).

Orthogonally, we would like to better understand the *effectiveness* of using `discourje` (e.g. in terms of reduced development costs).

Acknowledgements Funded by the Netherlands Organisation of Scientific Research (NWO): 016.Veni.192.103. This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

References

1. Go Team: Effective Go—The Go Programming Language (2009). https://golang.org/doc/effective_go.html. Accessed 28 Oct 2021
2. Go Team: Go 2016 Survey Results—The Go Blog (2017). https://blog.golang.org/survey2016-results. Accessed 28 Oct 2021
3. Go Team: Go 2017 Survey Results—The Go Blog (2018). https://blog.golang.org/survey2017-results. Accessed 28 Oct 2021
4. Go Team: Go 2018 Survey Results—The Go Blog (2019). https://blog.golang.org/survey2018-results. Accessed 28 Oct 2021

5. Go Team: Go Developer Survey 2019 Results—The Go Blog (2020). https://blog.golang.org/survey2019-results. Accessed 28 Oct 2021
6. Tu, T., Liu, X., Song, L., Zhang, Y.: Understanding real-world concurrency bugs in go. In: ASPLOS, pp. 865–878. ACM, New York (2019)
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), 29–12930 (2019)
8. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off go: liveness and safety for channel-based programming. In: POPL, pp. 748–761. ACM, New York (2017)
9. 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, New York (2018)
10. Ng, N., Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis. In: CC, pp. 174–184. ACM, New York (2016)
11. Stadtmüller, K., Sulzmann, M., Thiemann, P.: Static trace-based deadlock analysis for synchronous mini-go. In: APLAS. Lecture Notes in Computer Science, vol. 10017, pp. 116–136 (2016)
12. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification. Lecture Notes in Computer Science, vol. 10457, pp. 1–33. Springer, Berlin (2018)
13. Clojure Team: Clojure—State of Clojure 2019 Results (2019). https://clojure.org/news/2019/02/04/state-of-clojure-2019. Accessed 28 Oct 2021
14. Clojure Team: Clojure—State of Clojure 2020 Results (2019). https://clojure.org/news/2020/02/20/state-of-clojure-2020. Accessed 28 Oct 2021
15. Clojure Team: Clojure—Clojure core.async Channels (2013). https://clojure.org/news/2013/06/28/clojure-clore-async-channels. Accessed 28 Oct 2021
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284. ACM, New York (2008)
17. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: CONCUR. Lecture Notes in Computer Science, vol. 5201, pp. 418–433. Springer, Berlin (2008)
18. Bocchi, L., Yang, W., Yoshida, N.: Timed multiparty session types. In: CONCUR. Lecture Notes in Computer Science, vol. 8704, pp. 419–434. Springer, Berlin (2014)
19. Neykova, R., Bocchi, L., Yoshida, N.: Timed runtime monitoring for multiparty conversations. Formal Asp. Comput. **29**(5), 877–910 (2017)
20. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Typing access control and secure information flow in sessions. Inf. Comput. **238**, 68–105 (2014)
21. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Information flow safety in multiparty sessions. Math. Struct. Comput. Sci. **26**(8), 1352–1394 (2016)
22. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M., Rezk, T.: Session types for access and information flow control. In: CONCUR. Lecture Notes in Computer Science, vol. 6269, pp. 237–252. Springer, Berlin (2010)
23. Castellani, I., Dezani-Ciancaglini, M., Pérez, J.A.: Self-adaptation and secure information flow in multiparty communications. Formal Asp. Comput. **28**(4), 669–696 (2016)
24. Deniélou, P., Yoshida, N., Bejleri, A., Hu, R.: Parameterised multiparty session types. Log. Methods Comput. Sci. **8**(4) (2012)
25. Ng, N., Yoshida, N.: Pabble: parameterised scribble. Serv. Oriented Comput. Appl. **9**(3–4), 269–284 (2015)
26. 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, New York (2018)
27. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: FASE. Lecture Notes in Computer Science, vol. 9633, pp. 401–418. Springer, Berlin (2016)
28. Hu, R., Yoshida, N.: Explicit connection actions in multiparty session types. In: FASE. Lecture Notes in Computer Science, vol. 10202, pp. 116–133. Springer, Berlin (2017)
29. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP. LIPIcs, vol. 74, pp. 24–12431. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Wadern (2017)
30. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. Found. Trends Program. Lang. **3**(2–3), 95–230 (2016)
31. 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), 3–1336 (2016)
32. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party session. Log. Methods Comput. Sci. **8**(1), 1–28 (2012)
33. Hamers, R., Jongmans, S.: Discourje: Runtime verification of communication protocols in Clojure. In: TACAS (1). Lecture Notes in Computer Science, vol. 12078, pp. 266–284. Springer, Berlin (2020)
34. 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, Berlin (2020)
35. Horlings, E., Jongmans, S.: Analysis of specifications of multiparty sessions with dcj-lint. In: ESEC/SIGSOFT FSE, pp. 1590–1594. ACM, New York (2021)
36. Clojure Team: Clojure (2009). https://clojure.org. Accessed 28 Oct 2021
37. Hickey, R.: The Clojure programming language. In: DLS, p. 1. ACM, New York (2008)
38. Hickey, R.: A history of Clojure. Proc. ACM Program. Lang. **4**(HOPL), 71–17146 (2020)
39. Stack Overflow: Stack Overflow Developer Survey 2019 (2019). https://insights.stackoverflow.com/survey/2019. Accessed 28 Oct 2021
40. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9–1967 (2016)
41. Tasharofi, S., Dinges, P., Johnson, R.E.: Why do scala developers mix the actor model with other concurrency models? In: ECOOP. Lecture Notes in Computer Science, vol. 7920, pp. 302–326. Springer, Berlin (2013)
42. Parlett, D.: The Penguin Book of Card Games. Penguin, New york (2008)
43. Fokkink, W.: Introduction to Process Algebra. Texts in Theoretical Computer Science. An EATCS Series, Springer, Berlin (2000)
44. Baeten, J.C.M., Bravetti, M.: A ground-complete axiomatisation of finite-state processes in a generic process algebra. Math. Struct. Comput. Sci. **18**(6), 1057–1089 (2008)
45. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Program. **2**(3), 241–266 (1982)
46. Reynolds, M.: An axiomatization of full computation tree logic. J. Symb. Log. **66**(3), 1011–1057 (2001)
47. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996)

48. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
49. Kupferman, O., Pnueli, A.: Once and for all. In: LICS, pp. 25–35. IEEE Computer Society, Washington, DC (1995)
50. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: DAC, pp. 427–432. ACM Press, New York (1995)
51. Frumkin, M.A., Schultz, M.G., Jin, H., Yan, J.C.: Performance and scalability of the NAS parallel benchmarks in java. In: IPDPS, p. 139. IEEE Computer Society, Washington, DC (2003)
52. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI, pp. 89–100. ACM, New York (2007)
53. Hu, R., Neykova, R., Yoshida, N., Demangeon, R., Honda, K.: Practical interruptible conversations—distributed dynamic verification with session types and python. In: RV. Lecture Notes in Computer Science, vol. 8174, pp. 130–148. Springer, Berlin (2013)
54. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: CC, pp. 98–108. ACM, New York (2017)
55. Bocchi, L., Chen, T., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. Theor. Comput. Sci. **669**, 33–58 (2017)
56. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: CONCUR. Lecture Notes in Computer Science, vol. 6269, pp. 162–176. Springer, Berlin (2010)
57. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. Form. Methods Syst. Des. **46**(3), 197–225 (2015)
58. López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-based verification of message-passing parallel programs. In: OOPSLA, pp. 280–298. ACM, New York (2015)
59. Santos, C., Martins, F., Vasconcelos, V.T.: Deductive verification of parallel programs using why3. In: ICE. EPTCS, vol. 189, pp. 128–142 (2015)
60. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs. Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer, Berlin (2009)
61. Filliâtre, J., Paskevich, A.: Why3—where programs meet provers. In: ESOP. Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer, Berlin (2013)
62. Jongmans, S., Yoshida, N.: Exploring type-level bisimilarity towards more expressive multiparty session types. In: ESOP. Lecture Notes in Computer Science, vol. 12075, pp. 251–279. Springer, Berlin (2020)
63. Bonnaire-Sergeant, A., Davies, R., Tobin-Hochstadt, S.: Practical optional types for Clojure. In: ESOP. Lecture Notes in Computer Science, vol. 9632, pp. 68–94. Springer, Berlin (2016)
64. Pinzaru, G., Rivera, V.: Towards static verification of Clojure contract-based programs. In: TOOLS. Lecture Notes in Computer Science, vol. 11771, pp. 73–80. Springer, Berlin (2019)
65. Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: FMCO. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer, Berlin (2005)