Analysis of Specifications of Multiparty Sessions with dcj-lint

Erik Horlings
Open University
Heerlen, The Netherlands

Sung-Shik Jongmans
Open University
Heerlen, The Netherlands
Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands

ABSTRACT

Multiparty session types constitute a method to automatically detect violations of protocol implementations relative to specifications. But, when a violation is detected, does it symptomise a bug in the implementation or in the specification? This paper presents `dcj-lint`: an analysis tool to detect bugs in protocol specifications, based on multiparty session types. By leveraging a custom-built temporal logic model checker, `dcj-lint` can be used to efficiently perform: (1) generic sanity checks, and (2) protocol-specific property analyses. In our benchmarks, `dcj-lint` outperforms an existing state-of-the-art model checker (up to 61× faster).

CCS CONCEPTS

• **Software and its engineering → Specification languages**.

KEYWORDS

model checking, communication protocols, specifications, Clojure

ACM Reference Format:
Erik Horlings and Sung-Shik Jongmans. 2021. Analysis of Specifications of Multiparty Sessions with dcj-lint. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3468264.3473127

1 INTRODUCTION

1.1 Background

To take advantage of modern multi-core processors, concurrent programming—a notoriously complex enterprise—is becoming increasingly important. To alleviate some of the complexities, besides "low-level" *synchronisation*, several programming languages have started to offer core support for "higher-level" *communication* as well, in the guise of message passing through *channels* (e.g., Go, Rust, Clojure). The idea is that, beyond usage in distributed computing, channels can also serve as a programming abstraction for shared memory, supposedly less prone to concurrency bugs.

However, evidence suggests that channels have issues, too: after studying 171 concurrency bugs in popular Go programs [35], Tu et al. find that "message passing does not necessarily make multi-threaded programs less error-prone than shared memory."

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3473127

From the programmer's perspective, a major challenge is this: given a specification S of the *roles* (threads) and the *protocols* (sessions of communications between threads) that an implementation I should fulfill, how to guarantee that I is indeed *safe* relative to S? Safety means that "bad" channel actions never happen: if a channel action happens in I, then it is allowed to happen by S.

Multiparty session types (MPST) [18] constitute a method to automatically prove safety of implementations relative to specifications. The idea is to specify protocols as *behavioural types* [1, 22] against which threads are type-checked; the method ensures that well-typedness implies safety. Over the past decade, substantial progress in both MPST theory (e.g., extensions with time [3, 26], security [5–8], parametrisation [9, 13, 29]) and practice (e.g., tools for Clojure, Erlang, F#, Go, Java, Scala [9, 15, 20, 27, 28, 30]) has been made.

1.2 This Paper

When implementation I violates specification S, there are two cases: **(A)** S is right, while I is wrong. In this case, existing debugging/analysis tools for the implementation language can readily be used. **(B)** Alternatively, I could actually be right, while S is wrong. In this case, debugging/analysis tools for the specification language are needed. However, such tools do not yet exist for MPST; this is problematic, as MPST-based specifications get increasingly complex. Therefore:

> **Contribution 1:** We present `dcj-lint`: an analysis tool to detect bugs in MPST-based specifications.

Technically, `dcj-lint` is based on a custom-built temporal logic model checker. The advantage of "building our own" is twofold: on our side, it enables us to add/tailor features to our specific needs; on the programmer's side, it enables them to effortlessly deploy `dcj-lint`, without the need to install third-party model checkers or other external dependencies. However, a disadvantage of "building our own"—instead of reusing an existing model checker—is missing out on previously optimised code and algorithms. To study the magnitude of this disadvantage, we evaluated `dcj-lint` in quantitative performance experiments. Intriguingly, our benchmarks show:

> **Contribution 2:** `dcj-lint` outperforms the state-of-the-art model checker mCRL2 [4, 12].

In Section 2, we present an overview of `dcj-lint`, by example. In Section 3, we present design and implementation details. In Section 4, we present our evaluation. As part of the Discourje project, `dcj-lint` is now available at https://github.com/discourje.

2 OVERVIEW

2.1 Preliminaries & Workflow

`dcj-lint` can analyse specifications written in *Discourje* (pronunciation: "discourse") [15], of implementations written in *Clojure* [16].

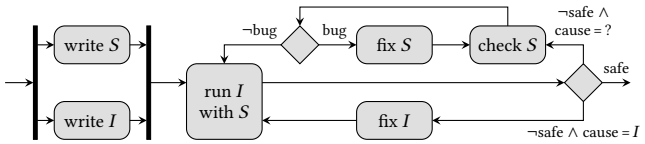


Figure 1: Workflow of debugging with `dcj-lint`.

```
4 (defsession :two-buyer []                          1 (defrole :buyer1)
5   (cat (--> String :buyer1 :seller)                2 (defrole :buyer2)
6        (--> Integer :seller :buyer1)               3 (defrole :seller)
7        (--> Integer :seller :buyer2)
8        (--> Integer :buyer1 :buyer2)
9        (--> Boolean :buyer2 :seller)
10       (par (close :buyer1 :buyer2) (close :buyer1 :seller)
11            (close :buyer2 :buyer1) (close :buyer2 :seller)
12            (close :seller :buyer1) (close :seller :buyer2))))
```

Figure 2: Discourje specification of the Two-Buyer protocol.

In a nutshell, Discourje is an MPST-based specification language that offers dynamic *behavioural type checking* (of channel actions), while Clojure is a Lisp-based implementation language that offers dynamic *data type checking*. Taken together, the unique aspect is that all type checking to ensure safety happens at run-time, of both behavioural types and data types, seamlessly. An advantage of developing `dcj-lint` in this context is that Discourje subsumes other MPST-based specification languages in expressiveness [9, 20, 27, 28, 30], so `dcj-lint` can easily be upgraded to support them.

Figure 1 summarises the envisaged workflow, including `dcj-lint`:

- First, the programmer writes an initial specification S in Discourje and an implementation I in Clojure.
- Next, the programmer "runs I with S". This means that Discourje's dynamic type checker validates every attempt of I to perform a channel action against S: if the channel action is unsafe, then it is *not* performed, but an exception is thrown.
- In case of an exception, the programmer diagnoses the problem: if it is "clearly" a bug in I, then they can fix I; else, they can analyse S using `dcj-lint`. The aim of such analyses is to *rule out* bugs in S, so the programmer can more confidently focus their attention on diagnosing and fixing I (even if the problem is not "clearly" a bug in I, it can still be one, *especially* with concurrency). `dcj-lint` supports both generic sanity checks and protocol-specific property analyses.

2.2 Example: The Two-Buyer Protocol [18]

To illustrate the workflow, we consider MPST's classical *Two-Buyer* protocol [18]: "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" [19].

Below, ▢ and ◇ indicate "actions" and "decisions" in Figure 1:

▢ First, we write the Discourje specification in Figure 2. Lines 1–3 specify the roles, while lines 4–12 specify the protocol. In general, (**-->** $t\ p\ q$) prescribes synchronous *communication* of a message that satisfies t through the channel from p to q, by p and q; (**close** $p\ q$) prescribes *closing* of the channel from p to q, by p; (**cat** $S_1 \ldots S_n$) and (**par** $S_1 \ldots S_n$) prescribe sequencing and interleaving. We note that Discourje has more features, including asynchronous communication, choices, and loops; `dcj-lint` supports those too. We also note that Discourje is built on top of Clojure/Java, so we can use some Clojure/Java features in specifications (e.g., data types).

```
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                 11 (thread ;; Buyer2                18 (thread ;; Seller
5   (>!! c1 "book")                  12   (let [x (<!! c6)              19   (<!! c1)
6   (let [x (<!! c5)                 13         y (<!! c2)              20   (>!! c5 (int 20))
7         y (/ x 2)]                 14         z (= x y)]              21   (>!! c6 (int 20))
8     (>!! c2 (int y)))15              (>!! c4 z))                     22   (println (<!! c4))
9   (close! c1)                      16   (close! c3)                  23   (close! c5)
10  (close! c2))                     17   (close! c4))                 24   (close! c6))
```

Figure 3: Clojure implementation of the Two-Buyer protocol.

Thus, in this specification: lines 5–9 prescribe 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 prescribe closings of all channels, in no particular order.

▢ Next, we write the Clojure implementation in Figure 3. Lines 1–3 implement the channels, while lines 4–24 implement the threads. In general, (**>!!** $c\ m$) sends m through c, (**<!!** c) receives a message through c, and (**close!** c) closes c; the other keywords are as usual.

Thus, in this implementation: the quote of Seller is `20` (variable `x` at Buyer1 and Buyer2); the contribution of Buyer1 is `10` (variable `y` at Buyer2), and the decision of Buyer2 is to reject (variable `z`).

▢ Next, we run the implementation with the specification. To do this, we first need to enable dynamic behavioural type checking, by adding the following lines between lines 3 and 4 in Figure 3:

```
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)
```

Thus, we create a *monitor* for a session of the Two-Buyer protocol (responsible for dynamic behavioural type checking), and link it to every channel, along with the intended sender and receiver.

◇ Next, we observe an 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 ***
```

The first two lines mean that the implementation of Buyer1 attempts to close its channel to Buyer2, indicated as `C(buyer1,buyer2)`, but that this is not allowed in the specification's current state, identified as `4`. Additionally, the last six lines show the relevant part of the *state space* of the specification (i.e., its formal semantics, computed on-the-fly), as a list of *transitions* of the form $(s, "\alpha", s')$, from state s to state s' with action α. Notably, in current state `4`, the specification allows only a communication from Buyer2 to Seller, indicated by `?(Boolean,buyer2,seller)`; not the attempted closing by Buyer1.

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

Figure 4: 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)
```

Figure 5: Causally unrelated actions are strictly ordered.

Thus, apparently, there is a timing issue with Buyer1's closing. However, this is not "clearly" a bug in the implementation: the specification prescribes all closings to happen at the end (Figure 2, lines 10–12), and indeed, every thread closes its channels at the end of its run (Figure 3, lines 9–10, 16–17, 23–24), so what goes wrong?

▢ Next, we analyse the specification using `dcj-lint`, by having it automatically perform 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, `dcj-lint` 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, `dcj-lint` provides the *witness* in Figure 4 (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 "smells" (cf. dead code and unused variables).

▢ Next, we remove (**close** :buyer2 :buyer1) from line 11 in Figure 2, and also (**close!** c3) from line 9 in Figure 3.

▢ Next, we re-analyse the specification using `dcj-lint`.

◇ 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 is 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.

To help debugging, `dcj-lint` provides the witness in Figure 5. 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 the 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 Figure 2, 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) (close :buyer1 :seller))
11 (par (~~close :buyer2 :buyer1~~) (close :buyer2 :seller)
12      (close :seller :buyer1) (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-analyse the specification using `dcj-lint`.

◇ 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 Seller2 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~~) (close :buyer2 :seller)
11               (close :seller :buyer1) (close :seller :buyer2)))
12     (close :buyer1 :buyer2) (close :buyer1 :seller))
```

▢ Next, we re-analyse the specification using `dcj-lint`.
◇ Next, no more issues are reported.
▢ Next, we re-run the implementation with the specification.

◇ At last, no more exceptions are reported. Thus, by iteratively using `dcj-lint`, we detected and fixed bugs in the specification that previously caused an exception; also, we alleviated a code smell.

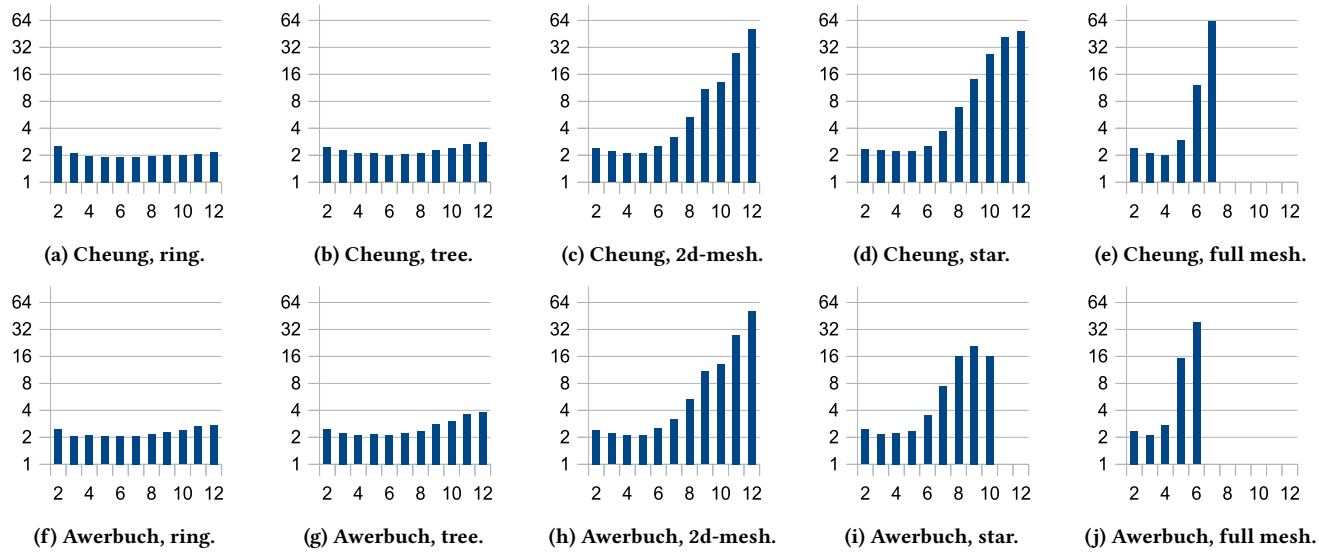
3 DESIGN & IMPLEMENTATION

The core of `dcj-lint` is a custom-built model checker for *computation tree logic* (CTL) [14]; it is implemented in Java 11 (data structures and algorithms) and Clojure 1.10 (front-end, integrated with Discourje), without additional libraries, and runs on the JVM. There are no additional requirements on operating system or hardware (although, as usual, model checking is memory-intensive).

The idea is to: **(1)** define properties of a specification S as CTL formulas; **(2)** compute the state space of S; **(3)** invoke a classical CTL model checking algorithm [11]. Regarding point 1, example propositions are `send(buyer1,seller)` and `close(buyer1,seller)`; these can be used, for instance, to write AG(`send(buyer1,seller)` $\Rightarrow$ AF(`close(buyer1,seller)`)). It states that on every path (A), always (G), if Buyer1 sends to Seller, then on every path, eventually (F), Buyer1 closes its channel to Seller. Regarding point 2, a state is a pair (α, S'), where α is the most recently performed action, and S' is the "remainder"; it has a transition to (β, S'') if S'' "remains" after performing β of S'. States contain enough information to evaluate propositions (e.g., `close(`p`,`q`)` is true in (α, S') iff $\alpha =$ `close(`p`,`q`)`).

Besides performing generic sanity checks, `dcj-lint` also enables the programmer to write and check custom formulas for protocol-specific properties in full CTL (witness generation works only for CTL's universal fragment, though). Other features:

- **Batch mode:** When asked to batch-check multiple formulas, the model checker reuses the state space and bookkeeping information between runs, to avoid double work. Notably, the seven generic sanity checks are performed in batch mode.
- **Past-time operators:** CTL allows one to express properties of the future. However, in our experience, many requirements are more naturally expressed in terms of properties



(a) Cheung, ring. (b) Cheung, tree. (c) Cheung, 2d-mesh. (d) Cheung, star. (e) Cheung, full mesh.

(f) Awerbuch, ring. (g) Awerbuch, tree. (h) Awerbuch, 2d-mesh. (i) Awerbuch, star. (j) Awerbuch, full mesh.

Figure 6: Speed-up of `dcj-lint` relative to mCRL2 (y-axis; log scale), as the number of processes increases (x-axis).

of the past. For instance: "if a channel is closed, then it must have been used" (i.e., one of the generic sanity checks). Therefore, `dcj-lint` also supports *Past CTL* (branching past) [23].

- **API:** Using an API, custom atomic propositions and temporal patterns can be written in Clojure to extend CTL's core. We used this feature to write the generic causality check.

4 EVALUATION

The main advantage of building our own model checker is that we can tailor our version of CTL to our needs (e.g., Past CTL and the API; see Section 3). However, the main disadvantage is missing out on previously optimised code and algorithms in existing model checkers. To study the magnitude of this disadvantage, we evaluated `dcj-lint` in quantitative performance experiments.

First, we wrote specifications of two *distributed* depth-first search algorithms, parametrised in the network topology: *Cheung's algorithm* [10] and *Awerbuch's algorithm* [2]; the latter has more parallelism than the former. We instantiated the specifications with various topologies (ordered by state space size: ring, tree, 2d-mesh, star, and full mesh) and various process numbers (up to twelve).

Next, we used `dcj-lint` and the state-of-the-art model checker *mCRL2* [4, 12] to perform five of the seven generic sanity checks on instantiated specifications; we omitted the remaining two generic sanity checks, as they could not straightforwardly be translated into mCRL2's format for formulas (they involve Past CTL and API-based temporal patterns). The input of `dcj-lint` consisted of instantiated specifications, while the input of mCRL2 consisted of their state spaces (generated by `dcj-lint`; i.e., instead of translating syntax from Discourje to mCRL2, we translate semantics). For each run, we measured the time it took `dcj-lint` or mCRL2 to perform all five checks, *minus* the common work that both tools needed (notably, generating state spaces). For every tool, for every instantiated specification, we performed 30 runs to smooth out variability, on a machine with an Intel E5-2690 v3 processor, Linux (kernel 3.10.0; Red Hat Enterprise Linux Server 7.9), default JVM settings (Oracle JDK 16.0.1), and the latest version of mCRL2 (202006.0).

Figure 6 shows our results; speed-ups are computed as ratios of means (30 measurements). Most standard deviations were less than 5% of the means, so the trends are useful. Missing bars indicate that at least mCRL2 did not finish 30 runs in one hour (time out).

The main finding is that `dcj-lint` outperforms mCRL2. Moreover, mCRL2 performs progressively worse relative to `dcj-lint`, as the number of processes increases; this effect is especially prominent for Awerbuch (more parallelism than Cheung) and for 2d-mesh/star/full mesh (more parallelism than ring/tree). Thus, `dcj-lint` is not only faster in absolute terms, but it also *scales* better.

Our main hypothesis regarding why mCRL2 is slower, is that it is actually *too general*: whereas `dcj-lint` supports (Past) CTL to write properties, mCRL2 supports the μ*-calculus*, for which the model checking problem has a higher computational complexity.

5 CONCLUSION

Regarding related work, recently, interest in using model checking to verify properties of MPST has been growing [21, 24, 31–34]; notably, most of these works employ mCRL2, which is why we used mCRL2 in our experiments. However, none of these works use model checking to debug global MPST-based specifications; this is a novel aspect of `dcj-lint`. More generally, channel-based protocol verification has been an early use case of model checking; a main difference with existing tools (e.g., Spin [17], Uppaal [25], and mCRL2) is that Discourje specifications are written from the global "system perspective" instead of the local "processes perspective"; this makes source-to-source translations to existing tools difficult.

ACKNOWLEDGMENTS

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] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages* 3, 2-3 (2016), 95–230.

[2] Baruch Awerbuch. 1985. A New Distributed Depth-First-Search Algorithm. *Inf. Process. Lett.* 20, 3 (1985), 147–150.

[3] Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *CONCUR (Lecture Notes in Computer Science, Vol. 8704)*. Springer, 419–434.

[4] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. 2019. The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability. In *TACAS (2) (Lecture Notes in Computer Science, Vol. 11428)*. Springer, 21–39.

[5] Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. 2014. Typing access control and secure information flow in sessions. *Inf. Comput.* 238 (2014), 68–105.

[6] Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. 2016. Information flow safety in multiparty sessions. *Math. Struct. Comput. Sci.* 26, 8 (2016), 1352–1394.

[7] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. 2010. Session Types for Access and Information Flow Control. In *CONCUR (Lecture Notes in Computer Science, Vol. 6269)*. Springer, 237–252.

[8] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. 2016. Self-adaptation and secure information flow in multiparty communications. *Formal Asp. Comput.* 28, 4 (2016), 669–696.

[9] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *PACMPL* 3, POPL (2019), 29:1–29:30.

[10] To-Yat Cheung. 1983. Graph Traversal Techniques and the Maximum Flow Problem in Distributed Computation. *IEEE Trans. Software Eng.* 9, 4 (1983), 504–512.

[11] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. 1986. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* 8, 2 (1986), 244–263.

[12] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. 2013. An Overview of the mCRL2 Toolset and Its Recent Advances. In *TACAS (Lecture Notes in Computer Science, Vol. 7795)*. Springer, 199–213.

[13] Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised Multiparty Session Types. *Log. Methods Comput. Sci.* 8, 4 (2012).

[14] E. Allen Emerson and Edmund M. Clarke. 1982. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Sci. Comput. Program.* 2, 3 (1982), 241–266.

[15] Ruben Hamers and Sung-Shik Jongmans. 2020. Discourje: Runtime Verification of Communication Protocols in Clojure. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 12078)*. Springer, 266–284.

[16] Rich Hickey. 2020. A history of Clojure. *Proc. ACM Program. Lang.* 4, HOPL (2020), 71:1–71:46.

[17] Gerard J. Holzmann. 2004. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

[18] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *POPL*. ACM, 273–284.

[19] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67.

[20] Raymond Hu and Nobuko Yoshida. 2016. Hybrid Session Verification Through Endpoint API Generation. In *FASE (Lecture Notes in Computer Science, Vol. 9633)*. Springer, 401–418.

[21] Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *FASE (Lecture Notes in Computer Science, Vol. 10202)*. Springer, 116–133.

[22] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1 (2016), 3:1–3:36.

[23] Orna Kupferman and Amir Pnueli. 1995. Once and For All. In *LICS*. IEEE Computer Society, 25–35.

[24] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In *ICSE*. ACM, 1137–1148.

[25] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a Nutshell. *Int. J. Softw. Tools Technol. Transf.* 1, 1-2 (1997), 134–152.

[26] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.* 29, 5 (2017), 877–910.

[27] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. 2018. A session type provider: compile-time API generation of distributed protocols with refinements in F#. In *CC*. ACM, 128–138.

[28] Rumyana Neykova and Nobuko Yoshida. 2017. Let it recover: multiparty protocol-induced recovery. In *CC*. ACM, 98–108.

[29] Nicholas Ng and Nobuko Yoshida. 2015. Pabble: parameterised Scribble. *Serv. Oriented Comput. Appl.* 9, 3-4 (2015), 269–284.

[30] Alceste Scalas, Ornela Dardha, Raymond Hu, and Nobuko Yoshida. 2017. A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming. In *ECOOP (LIPIcs, Vol. 74)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 24:1–24:31.

[31] Alceste Scalas and Nobuko Yoshida. 2019. Less is more: multiparty session types revisited. *PACMPL* 3, POPL (2019), 30:1–30:29.

[32] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Effpi: verified message-passing programs in Dotty. In *SCALA@ECOOP*. ACM, 27–31.

[33] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Verifying message-passing programs with dependent behavioural types. In *PLDI*. ACM, 502–516.

[34] Florian Joost Slob and Sung-Shik Jongmans. 2021. Prut4j: Protocol Unit Testing fo(u)r Java. In *ICST*. IEEE, 448–453.

[35] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *ASPLOS*. ACM, 865–878.