Practical Abstractions for Automated Verification of Message Passing Concurrency

Wytse Oortwijn(✉) and Marieke Huisman(✉)

University of Twente, Enschede, The Netherlands
{w.h.m.oortwijn,m.huisman}@utwente.nl

Abstract. Distributed systems are notoriously difficult to develop correctly, due to the concurrency in their communicating subsystems. Several techniques are available to help developers to improve the reliability of message passing software, including deductive verification and model checking. Both these techniques have advantages as well as limitations, which are complementary in nature. This paper contributes a novel verification technique that combines the strengths of deductive and algorithmic verification to reason elegantly about message passing concurrent programs, thereby reducing their limitations. Our approach allows to verify data-centric properties of message passing programs using concurrent separation logic (CSL), and allows to specify their communication behaviour as a process-algebraic model. The key novelty of the approach is that it formally bridges the typical abstraction gap between programs and their models, by extending CSL with logical primitives for proving deductively that a program refines its process-algebraic model. These models can then be analysed via model checking, using mCRL2, to reason indirectly about the program's communication behaviour. Our verification approach is compositional, comes with a mechanised correctness proof in Coq, and is implemented as an encoding in Viper.

1 Introduction

Distributed software is notoriously difficult to develop correctly. This is because distributed systems typically consist of multiple communicating components, which together have too many concurrent behaviours for a programmer to comprehend. Software developers therefore need formal techniques and tools to help them understand the full system behaviour, with the goal to guarantee the reliability of safety-critical distributed software. Two such formal techniques are *deductive verification* and *model checking*, both well-established in research [2,7] and proven successful in practice [12,14]. Nevertheless, both these techniques have their limitations. Deductive verification is often labour-intensive as it requires the system behaviour to be specified manually, via non-trivial code annotations, which is especially difficult for concurrent and distributed systems. Model checking, on the other hand, suffers from the typical abstraction gap [28]

W. Ahrendt and S. L. Tapia Tarifa (Eds.): IFM 2019, LNCS 11918, pp. 399–417, 2019.
https://doi.org/10.1007/978-3-030-34968-4_22

```
1 send (⟨4, 7, 5⟩, 1);
2 xs := recv 2;
3 assert xs = ⟨4, 5, 6, 7, 8⟩;
```

(a) Thread 1

```
4 while (true) {
5   (ys, t) := recv;
6   if (t = 1) then
7     send (ys + ⟨8, 6⟩, 3);
8   else send (ys, 2);
9 }
```

(b) Thread 2

```
10 while (true) {
11   (zs, t) := recv;
12   zs′ := ParSort(zs);
13   send (zs′, t);
14 }
```

(c) Thread 3

Fig. 1. Our message passing example, consisting of three communicating threads.

(i.e., discrepancies between the program and the corresponding model), as well as the well-known state-space explosion problem.

This paper contributes a scalable and practical technique for automated verification of message passing concurrency that reduces these limitations, via a sound combination of deductive verification and model checking. Our verification technique builds on the insight that deductive and algorithmic verification are complementary [3,32,34]: the former specialises in verifying data-oriented properties (e.g., the function `sort`(xs) returns a sorted permutation of xs), while the latter targets temporal properties of control-flow (e.g., any `send` must be preceded by a `recv`). Since realistic distributed software deals with both computation (data) and communication (control-flow), such a combination of complementary verification techniques is needed, to handle all program aspects.

More specifically, our verification approach uses concurrent separation logic (CSL) to reason about data properties of message passing concurrent programs, and allows to specify their communication behaviour as a process-algebraic model. The key innovation is that CSL is used not only to specify data-oriented properties, but also to formally link the program's communication behaviour to the process-algebraic specification of its behaviour, thereby bridging the typical abstraction gap between programs and their models. These process-algebraic models can then be analysed algorithmically, e.g., using mCRL2 [8,13], to reason indirectly about the communication behaviour of the program. These formal links preserve *safety properties*; the preservation of liveness properties is left as future work. This approach has been proven sound using the Coq proof assistant, and has been implemented as an encoding in the Viper concurrency verifier [22].

Running Example. To further motivate the approach, consider the example program in Fig. 1, consisting of three threads that exchange integer sequences via synchronous message passing. The goal is to verify whether the asserted property in Thread 1's program holds. This program is a simplified version of a typical scenario in message passing concurrency: it involves computation as well as communication and has a complicated communication pattern, which makes it difficult to see and prove that the asserted property indeed holds.

Clarifying the program, Thread 1 first sends the sequence $\langle 4, 7, 5\rangle$ to the environmental threads as a message with tag 1, and then receives any outstanding

integer sequence tagged 2. Thread 2 continuously listens for incoming messages of any tag with a wildcard receive, and redirects these messages, possibly with slightly modified content depending on the message tag. Thread 3 is a computing service: it sorts all incoming requests and sends back the result with the original tag. `ParSort` is assumed to be the implementation of an intricate, heavily optimised parallel sorting algorithm. Note that the asserted property holds because the **send** on line 7 is always executed, no matter the interleaving of threads.

Two standard potential approaches for verifying this property are deductive verification and model checking. However, neither of these approaches provides a satisfying solution. Techniques for deductive verification, e.g., concurrent separation logic, have their power in modularity and compositionality: they require modular independent proofs for the three threads, and allow to compose these into a single proof for the entire program. This would not work in our example scenario, as the asserted property is inherently global. One could attempt to impose global invariants on the message exchanges [38], but these are generally hard to come by. Finding a global invariant for this example would already be difficult, since there is no obvious relation between the contents of messages and their tags. Other approaches use ideas from assume-guarantee reasoning [30,36] to somehow intertwine the independent proofs of the threads' programs, but these require extra non-trivial specifications of thread-interference and are difficult to integrate into (semi-)automatic verifiers.

Alternatively, one may construct a model of this program and apply a model checker, which fits more naturally with the temporal nature of the program's communication behaviour. However, this does not give a complete solution either. In particular, certain assumptions have to be made while constructing the model, for example that `ParSort` is correctly implemented. The correctness property of `ParSort` is data-oriented (it relates the output of `ParSort` to its input) and thus in turn fits more naturally with deductive verification. But even when one uses both these approaches—deductive verification for verifying `ParSort` and model checking for verifying communication behaviour—there still is no formal connection between their results: perhaps the model incorrectly reflects the program's communication behaviour.

Contributions and Outline. This paper contributes a novel approach that allows to make such formal connections, by extending CSL with primitives for proving that a program *refines* a process-algebraic model, with respect to send/receive behaviour. Section 2 introduces the syntax and semantics of programs and process-algebraic models. Notably, our process algebra language is similar to mCRL2, but has a special *assertion primitive* of the form $?b$, that allows to encode Boolean properties b into the process itself, as logical assertions. These properties can be verified via a straightforward reduction to mCRL2, and can subsequently be used (relied upon) inside the deductive proof of the program, via special program annotations of the form **query** b, (allowing to "query" for properties b proven on the process level). Section 3 illustrates in detail how this works on the example program of Fig. 1, before Sect. 4 discusses the under-

lying logical machinery and its soundness proof. This soundness argument has been mechanised using Coq, and the program logic has been encoded in Viper. Section 5 discusses various extensions of the approach. Finally, Sect. 6 relates our work to existing approaches and Sect. 7 concludes.

2 Programs and Processes

This section introduces the programming language (Sect. 2.1) and the process algebra language of models (Sect. 2.2) that are used to formalise the approach.

2.1 Programs

The syntax of our simple concurrent pointer language, inspired by [6,25], is as follows, where $x, y, z, \cdots \in \mathit{Var}$ are *variables* and $v, w, \cdots \in \mathit{Val}$ are *values*.

Definition 1 (Expressions, Conditions, Commands)

$$
\begin{aligned}
e \in \mathit{Expr} &::= v \mid x \mid e + e \mid e - e \mid \cdots \\
b \in \mathit{Cond} &::= \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b \wedge b \mid e = e \mid e < e \mid \cdots \\
C \in \mathit{Cmd} &::= \mathbf{skip} \mid C; C \mid C \parallel C \mid x := e \mid x := [e] \mid [e] := e \mid \mathbf{send}\ (e, e) \mid \\
&\quad\ (x, y) := \mathbf{recv} \mid x := \mathbf{recv}\ e \mid x := \mathbf{alloc}\ e \mid \mathbf{dispose}\ e \mid \\
&\quad\ \mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ b\ \mathbf{do}\ C \mid \mathbf{atomic}\ C \mid \mathbf{query}\ b
\end{aligned}
$$

This language has instructions to handle dynamically allocated memory, i.e., heaps, as well as primitives for message passing, to allow reasoning about both shared-memory and message passing concurrency models, and their combination.

The notation $[e]$ is used for *heap dereferencing*, where e is an expression whose evaluation determines the heap location to dereference. Memory can be dynamically allocated on the heap using the **alloc** e instruction, where e will be the initial value of the fresh heap cell, and be deallocated using **dispose**.

The command **send** (e_1, e_2) sends a message e_1 to the environmental threads, where e_2 is a *message tag* that can be used for message identification. Messages are received in two ways: $x :=$ **recv** e receives a message with a tag that matches the expression e, whereas $(x, y) :=$ **recv** receives *any* message and writes the message tag to the extra variable y, i.e., a *wildcard* receive.

The specification command **query** b is used to connect process-algebraic reasoning to deductive reasoning: it allows the deductive proof of a program to rely on (or *assume*) a Boolean property b, which is proven to hold (or *guaranteed*) via process-algebraic analysis. This is a ghost command that does not interfere with regular program execution.

The function $\mathsf{fv} : \mathit{Expr} \to 2^{\mathit{Var}}$ is used to determine the set of free variables of expressions as usual, and is overloaded for conditions. Substitution is written $e_1[x/e_2]$ (and likewise for conditions) and has a standard definition: replacing each occurrence of x by e_2 in e_1.

$$(\textbf{send}\ (e_1, e_2), h, \sigma) \xrightarrow{\mathit{send}([\![e_1]\!]\sigma, [\![e_2]\!]\sigma)} (\textbf{skip}, h, \sigma)$$

$$((x, y) := \textbf{recv}, h, \sigma) \xrightarrow{\mathit{recv}(v, v')} (\textbf{skip}, h, \sigma[x \mapsto v, y \mapsto v'])$$

$$(x := \textbf{recv}\ e, h, \sigma) \xrightarrow{\mathit{recv}(v, [\![e]\!]\sigma)} (\textbf{skip}, h, \sigma[x \mapsto v]) \qquad (\textbf{query}\ b, h, s) \xrightarrow{\mathit{qry}} (\textbf{skip}, h, s)$$

$$\frac{(C_1, h, \sigma) \xrightarrow{\mathit{send}(v_1, v_2)} (C_1', h, \sigma) \qquad (C_2, h, \sigma) \xrightarrow{\mathit{recv}(v_1, v_2)} (C_2', h, \sigma')}{(C_1 \parallel C_2, h, \sigma) \xrightarrow{\mathit{comm}(v_1, v_2)} (C_1' \parallel C_2', h, \sigma')}$$

Fig. 2. An excerpt of the small-step operational semantics of programs.

Semantics. The denotational semantics of expressions $[\![e]\!]\sigma$ and conditions $[\![b]\!]\sigma$ is defined in the standard way, with $\sigma \in \mathit{Store} \triangleq \mathit{Var} \to \mathit{Val}$ a *store* that gives an interpretation to variables. Sometimes $[\![e]\!]$ is written instead of $[\![e]\!]\sigma$ when e is closed, and likewise for $[\![b]\!]$.

The operational semantics of commands is defined as a labelled small-step reduction relation $\longrightarrow \subseteq \mathit{Conf} \times \mathit{Label} \times \mathit{Conf}$ between configurations $\mathit{Conf} \triangleq \mathit{Cmd} \times \mathit{Heap} \times \mathit{Store}$ of programs. Heaps $h \in \mathit{Heap} \triangleq \mathit{Loc} \rightharpoonup_{\mathsf{fin}} \mathit{Val}$ are used to model shared memory and are defined as finite partial mappings, with $\mathit{Loc} \subseteq \mathit{Val}$ an infinite domain of heap locations. The transition labels represent the atomic (inter)actions of threads, and are defined as follows.

$$l \in \mathit{Label} ::= \ \mathit{send}(v, v) \mid \mathit{recv}(v, v) \mid \mathit{comm}(v, v) \mid \mathit{cmp} \mid \mathit{qry}$$

Transitions labelled $\mathit{send}(v, v')$ indicate that the program sends a value v from the current configuration, together with a tag v'. These can be received by a thread, as a transition labelled $\mathit{recv}(v, v')$. By doing so, the sending and receiving threads *communicate*, represented by the comm label. Internal computations that are not related to message passing are labelled cmp, e.g., heap reading or writing. The only exception to this are the reductions of **query** commands, which are given the label qry instead, for later convenience in proving soundness.

Figure 2 gives an excerpt of the reduction rules for message exchanging. All other rules are standard in spirit [21,35] and are therefore deferred to [1]. For ease of presentation, a synchronous message passing semantics is used for now. However, our approach can easily be extended to asynchronous message passing.

2.2 Processes

In this work the communication behaviour of programs is specified as a process algebra with data, whose language is defined by the following grammar.

Definition 2 (Processes)

$$P, Q ::= \ \varepsilon \mid \delta \mid \mathsf{send}(e, e) \mid \mathsf{recv}(e, e) \mid ?b \mid b : P \mid P \cdot P \mid P + P \mid P \parallel P \mid \Sigma_x\, P \mid P^*$$

Successful termination

$$\varepsilon\downarrow \qquad P^*\downarrow \qquad \frac{P\downarrow \quad Q\downarrow}{P\cdot Q\downarrow} \qquad \frac{P\downarrow}{P+Q\downarrow} \qquad \frac{P[x/v]\downarrow}{\Sigma_x P\downarrow} \qquad \frac{[\![b]\!] \quad P\downarrow}{b:P\downarrow}$$

Operational semantics

$$\mathsf{send}(e_1,e_2) \xrightarrow{\mathit{send}([\![e_1]\!],[\![e_2]\!])} \varepsilon \qquad \mathsf{recv}(e_1,e_2) \xrightarrow{\mathit{recv}([\![e_1]\!],[\![e_2]\!])} \varepsilon \qquad \frac{[\![b]\!]}{?b \xrightarrow{\mathit{assn}} \varepsilon}$$

$$\frac{P \xrightarrow{\alpha} P'}{P\cdot Q \xrightarrow{\alpha} P'\cdot Q} \qquad \frac{P\downarrow \quad Q \xrightarrow{\alpha} Q'}{P\cdot Q \xrightarrow{\alpha} Q'} \qquad \frac{P[x/v] \xrightarrow{\alpha} P'}{\Sigma_x P \xrightarrow{\alpha} P'} \qquad \frac{[\![b]\!] \quad P \xrightarrow{\alpha} P'}{b:P \xrightarrow{\alpha} P'}$$

$$\frac{P \xrightarrow{\alpha} P'}{P^* \xrightarrow{\alpha} P'\cdot P^*} \qquad \frac{P \xrightarrow{\mathit{send}(v_1,v_2)} P' \quad Q \xrightarrow{\mathit{recv}(v_1,v_2)} Q'}{P \parallel Q \xrightarrow{\mathit{comm}(v_1,v_2)} P' \parallel Q'}$$

Fault semantics

$$\frac{\neg[\![b]\!]}{?b \longrightarrow \lightning} \qquad \frac{P\downarrow \quad Q \longrightarrow \lightning}{P\cdot Q \longrightarrow \lightning} \qquad \frac{P[x/v] \longrightarrow \lightning}{\Sigma_x P \longrightarrow \lightning} \qquad \frac{[\![b]\!] \quad P \longrightarrow \lightning}{b:P \longrightarrow \lightning} \qquad \frac{P \longrightarrow \lightning}{P^* \longrightarrow \lightning}$$

Fig. 3. An excerpt of the small-step operational semantics of processes.

Clarifying the standard connectives, ε is the empty process without behaviour, and δ is the deadlocked process that neither progresses nor terminates. The process $\Sigma_x P$ is the infinite summation $P[x/v_0] + P[x/v_1] + \cdots$ over all values $v_0, v_1, \ldots \in \mathit{Val}$. Sometimes $\Sigma_{x_0,\ldots,x_n} P$ is written to abbreviate $\Sigma_{x_0} \cdots \Sigma_{x_n} P$. The guarded process $b : P$ behaves as P if the guard b holds, and otherwise behaves as δ. The process P^* is the Kleene iteration of P and denotes a sequence of zero or more P's. The infinite iteration of P is derived to be $P^\omega \triangleq P^* \cdot \delta$.

Since processes are used to reason about send/receive behaviour, this process algebra language exclusively supports two actions, $\mathsf{send}(e_1, e_2)$ and $\mathsf{recv}(e_1, e_2)$, for sending and receiving data elements e_1, together with a message tag e_2.

Finally, $?b$ is the *assertive process*, which is very similar to guarded processes: $?b$ is behaviourally equivalent to δ in case b does not hold. However, assertive processes have a special role in our approach: they are the main subject of process-algebraic analysis, as they encode the properties b to verify, as logical assertions. Moreover, they are a key component in connecting process-algebraic reasoning with deductive reasoning, as their properties can be relied upon in the deductive proofs of programs via the **query** b ghost command.

The function fv is overloaded to determine the set of unbound variables in process terms. As always, any process P is defined to be *closed* if $\mathsf{fv}(P) = \emptyset$.

Semantics. Figure 3 presents the operational semantics of processes, which is defined in terms of a labelled binary reduction relation $\xrightarrow{\cdot} \subseteq \mathit{Proc} \times \mathit{ProcLabel} \times$

Proc between processes. The labels of the reduction rules are defined as follows.

$$\alpha \in \mathit{ProcLabel} ::= \mathit{send}(v, v) \mid \mathit{recv}(v, v) \mid \mathit{comm}(v, v) \mid \mathit{assn}$$

The labels *send*, *recv* and *comm* are used in the same manner as those of program transitions, whereas *assn* indicates reductions of assertional processes.

The reduction rules are mostly standard [10,13]. Processes are assumed to be closed as a well-formedness condition, preventing the need to include stores. Moreover, it is common to use an explicit notion of *successful termination* in process algebras with ε [4]. The notation $P \downarrow$ intuitively means that P has the choice to have no further behaviour and thus to behave as ε. The send and recv actions communicate in the sense that they synchronise as a *comm* transition.

The property of interest for process-algebraic verification is to check for absence of faults. Any closed process P *exhibits a fault*, denoted $P \longrightarrow \lightning$, if P is able to violate an assertion. Furthermore, any process P is defined to be *safe*, written $P\checkmark$, if P can never reach a state that exhibits a fault, while following the reduction rules of the operational semantics.

Definition 3 (Process safety). *The $\checkmark$ predicate is coinductively defined such that, if $P\checkmark$ holds, then $P \not\longrightarrow \lightning$, and $P \xrightarrow{\alpha} P'$ implies $P'\checkmark$ for any α and P'.*

Given any closed process P, determining whether $P\checkmark$ holds can straightforwardly and mechanically be reduced to an mCRL2 model checking problem. This is done by modelling an explicit fault state that is reachable whenever an assertive process is violated, as a distinctive $\lightning$ action. Checking for fault absence is then reduced to checking the μ-calculus formula $[\mathsf{true}^* \cdot \lightning]\mathsf{false}$ on the translated model, meaning "no faulty transitions are ever reachable".

Process bisimilarity is defined as usual, and preserves faults and termination.

Definition 4 (Bisimulation). *A binary relation $\mathscr{R} \subseteq \mathit{Proc} \times \mathit{Proc}$ is a* bisimulation relation *over closed processes if, whenever $P\mathscr{R}Q$, then*

- *$P \downarrow$ if and only if $Q \downarrow$.*
- *$P \longrightarrow \lightning$ if and only if $Q \longrightarrow \lightning$.*
- *If $P \xrightarrow{\alpha} P'$, then there exists a Q' such that $Q \xrightarrow{\alpha} Q'$ and $P'\mathscr{R}Q'$.*
- *If $Q \xrightarrow{\alpha} Q'$, then there exists a P' such that $P \xrightarrow{\alpha} P'$ and $P'\mathscr{R}Q'$.*

Two closed processes P and Q are defined to be *bisimilar*, or *bisimulation equivalent*, denoted $P \cong Q$, if there exists a bisimulation relation $\mathscr{R}$ such that $P\mathscr{R}Q$. Any bisimulation relation constitutes an equivalence relation. In our Coq encoding [1], we have proven soundness of various standard bisimulation equivalences for this language. As usual, bisimilarity is a congruence for all process-algebraic connectives. Moreover, process safety is closed under bisimilarity.

3 Verification Example

Before discussing the logical details of our approach, let us first demonstrate it on the example program of Fig. 1. Application of the technique consists of the following three steps:

1. Constructing a process-algebraic model that captures the program's send/recv behaviour;
2. Analysing the model to determine whether the value received by Thread 1 is always the sorted sequence $\langle 4,5,6,7,8\rangle$, via a reduction to an mCRL2 model checking problem; and
3. Deductively verifying whether the program correctly implements the process-algebraic model with respect to send/receive behaviour, by using concurrent separation logic.

The remainder of this section discusses each of these three steps in detail.

Step 1: Constructing a Process-Algebraic Model. The communication behaviour of the example program can straightforwardly be captured as a process $\mathsf{P} = \mathsf{P}_1 \parallel \mathsf{P}_2 \parallel \mathsf{P}_3$ (assuming that the expression language is rich enough to handle sequences), so that P_i captures Thread i's send/receive behaviour, where

$$\mathsf{P}_1 \triangleq \mathsf{send}(\langle 4,7,5\rangle, 1) \cdot \Sigma_{xs}\, \mathsf{recv}(xs, 2) \cdot ?(xs = \langle 4,5,6,7,8\rangle)$$

$$\mathsf{P}_2 \triangleq \mathsf{P}_2'^{\omega}, \text{ with } \mathsf{P}_2' \triangleq \Sigma_{ys,t}\, \mathsf{recv}(ys, t) \cdot (t = 1 : \mathsf{send}(ys +\!\!+ \langle 8,6\rangle, 3) + t \neq 1 : \mathsf{send}(ys, 2))$$

$$\mathsf{P}_3 \triangleq \mathsf{P}_3'^{\omega}, \text{ with } \mathsf{P}_3' \triangleq \Sigma_{zs,t}\, \mathsf{recv}(zs, t) \cdot \mathsf{send}(\mathtt{sort}(zs), t)$$

Observe that P_1 encodes the property of interest as the assertion $?(xs = \langle 4,5,6,7,8\rangle)$. The validity of this assertion is checked by mCRL2 on the translated model, as described in the next paragraph. Moreover, `sort` is assumed to be the functional description of a sorting algorithm. Such a description can axiomatically be defined in mCRL2. The `sort` mapping can easily act as a functional specification for the implementation of more intricate sorting algorithms like `ParSort`. Deductive verifiers are generally well-suited to relate such functional specifications to implementations via pre/postcondition reasoning:

```
1 ensures \result = sort(xs);
2 seq⟨nat⟩ ParSort(seq⟨nat⟩ xs) { ··· }
```

Step 2: Analysing the Process-Algebraic Model. The composite process P can straightforwardly be translated to mCRL2 input and be analysed. Our translation can be found online at [1]. This translation has been done manually, yet it would not be difficult to write a tool that does it mechanically (we are actively working on this).

Notably, assertive processes $?b$ are translated into $\mathsf{check}(b)$ actions. The action $\mathsf{check}(\mathsf{false})$ can be seen as the encoding of $\lightning$. Checking for process safety $\mathsf{P}\checkmark$ can be reduced to checking the μ-calculus formula $\phi = [\mathsf{true}^* \cdot \mathsf{check}(\mathsf{false})]\mathsf{false}$, stating that no $\mathsf{check}(\mathsf{false})$ action can ever be performed, or equivalently that the process is free of faults. mCRL2 can indeed confirm that P is fault-free by checking ϕ, and thus that the asserted property holds. In Step 3 we formally prove that the program adheres to the communication behaviour described by P, which allows to project these model checking results onto program behaviour.

```
1  {Proc(P1)}
2  send (⟨4, 7, 5⟩, 1);
3  {Proc(Σx recv(x, 2) · ?(x = ⟨4, 5, 6, 7, 8⟩))}
4  xs := recv 2;
5  {Proc(?(xs = ⟨4, 5, 6, 7, 8⟩))}
6  query xs = ⟨4, 5, 6, 7, 8⟩;
7  {Proc(ε) ∗ xs = ⟨4, 5, 6, 7, 8⟩}
8  assert xs = ⟨4, 5, 6, 7, 8⟩;
9  {Proc(ε) ∗ xs = ⟨4, 5, 6, 7, 8⟩}
```

(a) Proof of Thread 1's program

```
1  {Proc(P3)}
2  while (true) invariant Proc(P3'^ω) {
3    {Proc(P3'^ω)}
4    {Proc(P3' · P3'^ω)}
5    (zs, t) := recv;
6    {Proc(send(sort(zs), t) · P3'^ω)}
7    zs' := ParSort(zs);
8    {Proc(send(sort(zs), t) · P3'^ω) ∗
9        zs' = sort(zs)}
10   send (zs', t);
11   {Proc(P3'^ω)}
12 }
```

(b) Proof of Thread 3's program

Fig. 4. Proofs for Threads 1 and 3 of our example. Thread 2 is proven likewise.

Step 3: Connecting Processes to Program Behaviour. The final step is to deductively prove that Fig. 1's program refines P, with respect to communication behaviour, using CSL. To do this, we extend CSL with predicates of the form $\mathsf{Proc}(P)$, which express that the remaining program will communicate as prescribed by the process P—the program's model. More specifically, the proof system enforces that every **send** (e, e') instruction must be prescribed by a $\mathsf{Proc}(\mathsf{send}(e, e') \cdot P)$ predicate in the logic, and likewise for **recv**, thereby enforcing that the process-algebraic model can perform a matching send or recv action. These actions are then *consumed* in the logic, while following the structure of the program. Similarly, **query** b annotations must be prescribed by a $\mathsf{Proc}(?b \cdot P)$ predicate, and allow to assume b in the logic as result of Step 2, by which $?b$ is consumed from the Proc predicate.

Figure 4 illustrates this, by giving the intermediate steps of the proofs of Threads 1 and 3. An extra **query** annotation has been added in Thread 1's program for obtaining the asserted property from P_1. Moreover, the annotated **invariant** in Thread 3 is a loop invariant that states that $\mathsf{Proc}(\mathsf{P}_3'^{\omega})$ prescribes the communication behaviour of every loop iteration.

Another feature of the logic is that $\mathsf{Proc}(P)$ predicates can be *split* and *merged* along parallel compositions inside P, in the style of CSL. This is used in the top-level proof of the example program, shown in Fig. 5. The $*$ connective is the *separating conjunction* from separation logic, which now expresses that different threads will use different parts of the model. This makes the approach both modular and compositional, by allowing the program's top-level proof to be composed out of the individual independent proofs of its threads.

We encoded the program logic into the Viper concurrency verifier and used it to fully mechanise the deductive proof of the example program. The Viper files are available online at [1]. This encoding primarily consists of an axiomatic domain for processes, containing constructors for the process-algebraic connectives, supported by standard axioms of process algebra (which we have proven

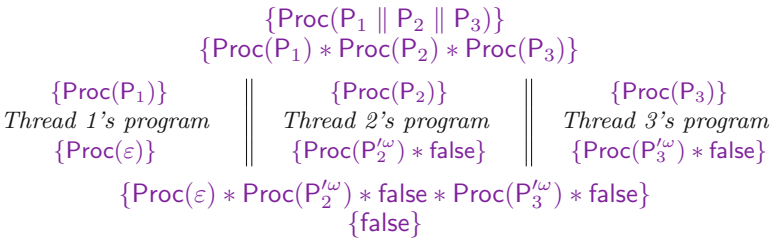

Fig. 5. The top-level specification of the example program.

sound in our Coq encoding). The Proc assertions are then encoded as unary predicates over these process types. Viper can verify correctness of the example program in under 3 s.

4 Formalisation

This section discusses the assertion language and entailment rules of the program logic (Sect. 4.1), the Hoare-triple rules for message passing and querying (Sect. 4.2), and their soundness (Sect. 4.3).

4.1 Program Logic

The program logic extends intuitionistic concurrent separation logic [17,35], where the assertion language is defined by the following grammar.

Definition 5 (Assertions)

$$\mathcal{P}, \mathcal{Q} ::= \; b \mid \forall x.\mathcal{P} \mid \exists x.\mathcal{P} \mid \mathcal{P} \vee \mathcal{Q} \mid \mathcal{P} * \mathcal{Q} \mid \mathcal{P} \mathrel{-\!\!*} \mathcal{Q} \mid e \hookrightarrow_\pi e \mid \mathsf{Proc}(P) \mid P \approx Q$$

The assertion $e_1 \hookrightarrow_\pi e_2$ is the standard *heap ownership assertion* and expresses the knowledge that the heap holds the value e_2 at heap location e_1. Moreover, $\pi \in (0, 1]_\mathbb{Q}$ is a *fractional permission* in the style of Boyland [5] and determines the type of ownership: write access to e_1 is provided in case $\pi = 1$, and read access is provided in case $0 < \pi < 1$.

The $\mathcal{P} * \mathcal{Q}$ connective is the *separating conjunction* from CSL, and expresses that the ownerships captured by $\mathcal{P}$ and $\mathcal{Q}$ are *disjoint*, e.g., it is disallowed that both express write access to the same heap location. The $\mathrel{-\!\!*}$ connective is known as the *magic wand* and describes hypothetical modifications of the current state.

The assertion $\mathsf{Proc}(P)$ expresses the ownership of the right to send and receive messages as prescribed by the process P. Here P may contain free variables and may be replaced by any process bisimilar to P. To handle such replacements, the assertion $P \approx Q$ can be used, which expresses that P and Q are bisimilar in the current context. To give an example, one may wish to deduce that $\mathsf{Proc}(0 < x : P) * x = 2$ entails $\mathsf{Proc}(P)$. Even though $0 < x : P$ has free variables, it is used in a context where x equals 2, and therefore $0 < x : P \approx P$ can be established. We now discuss the entailment rules for these deductions.

$\hookrightarrow$-SPLIT-MERGE
$e_1 \hookrightarrow_{\pi_1+\pi_2} e_2 \dashv\vdash e_1 \hookrightarrow_{\pi_1} e_2 * e_1 \hookrightarrow_{\pi_2} e_2$

Proc-SPLIT-MERGE
$\mathsf{Proc}(P \parallel Q) \dashv\vdash \mathsf{Proc}(P) * \mathsf{Proc}(Q)$

Proc-$\approx$
$\mathsf{Proc}(P) * P \approx Q \vdash \mathsf{Proc}(Q)$

$\approx$-BISIM
$$\frac{P \cong Q}{\vdash P \approx Q}$$

$\approx$-REFL
$\vdash P \approx P$

$\approx$-SYMM
$P \approx Q \vdash Q \approx P$

$\approx$-TRANS
$P \approx Q * Q \approx R \vdash P \approx R$

$\approx$-COND-TRUE
$b \vdash b : P \approx P$

$\approx$-COND-FALSE
$b \vdash \neg b : P \approx \delta$

Fig. 6. An excerpt of the entailment rules of the program logic.

Proof Rules. Figure 6 shows an excerpt of the proof rules, which are given as sequents of the form $\vdash \mathcal{P}$ and $\mathcal{P} \vdash \mathcal{Q}$. All other proof rules are deferred to [1]. The notation $\mathcal{P} \dashv\vdash \mathcal{Q}$ is shorthand for $\mathcal{P} \vdash \mathcal{Q}$ and $\mathcal{Q} \vdash \mathcal{P}$. All proof rules are sound in the standard sense.

The $\hookrightarrow$-SPLIT-MERGE rule expresses that heap ownership predicates can be *split* (in the left-to-right direction) and *merged* (right-to-left) along π, allowing heap ownership to be distributed over different threads. Likewise, Proc-SPLIT-MERGE allows to split and merge process predicates along the parallel composition, to allow different threads to communicate as prescribed by the different parts of the process-algebraic model. Process terms inside Proc predicates may be replaced by bisimilar ones via Proc-$\approx$. This rule can be used to rewrite process terms to a canonical form used by some other proof rules. The $\approx$ connective enjoys properties similar to $\cong$: it is an equivalence relation with respect to $*$, as shown by $\approx$-REFL, $\approx$-SYMM and $\approx$-TRANS, and is a congruence for all process-algebraic connectives. Finally, $\approx$ allows to use contextual information to resolve guards, via $\approx$-COND-TRUE and $\approx$-COND-FALSE.

Semantics of Assertions. The semantics of assertions is given as a modelling relation $\iota, \sigma, P \models \mathcal{P}$, where the models are abstractions of program states (these can also be seen as partial program states). These state abstractions consist of three components, the first being a *permission heap*. Permission heaps $\iota \in \mathit{PermHeap} \triangleq \mathit{Loc} \rightharpoonup_{\mathsf{fin}} (0,1]_{\mathbb{Q}} \times \mathit{Val}$ extend normal heaps by associating a fractional permission to each occupied heap cell. The second component is an ordinary store σ, and the last component is a *closed* process P that determines the state of the process-algebraic model that may be described by $\mathcal{P}$.

The semantics of assertions relies on the notions of disjointness and disjoint union of permission heaps. Two permission heaps ι_1, ι_2 are said to be *disjoint*, written $\iota_1 \perp \iota_2$, if they agree on their contents and the pairwise addition of the fractional permissions they store are again valid fractional permissions. Furthermore, the *disjoint union* of ι_1 and ι_2, which is written $\iota_1 \uplus \iota_2$, is defined to be the pairwise union of all their disjoint heap cells.

Definition 6 (Disjointness of permission heaps)

$$\iota_1 \perp \iota_2 \triangleq \forall \ell \in \mathit{dom}(\iota_1) \cap \mathit{dom}(\iota_2) \,.\, \iota_1(\ell) \perp_{\mathsf{cell}} \iota_2(\ell), \text{ where}$$
$$(\pi_1, v_1) \perp_{\mathsf{cell}} (\pi_2, v_2) \triangleq v_1 = v_2 \wedge \pi_1 + \pi_2 \in (0,1]_{\mathbb{Q}}$$

Definition 7 (Disjoint union of permission heaps)

$$\iota_1 \uplus \iota_2 \triangleq \lambda \ell \,.\, \begin{cases} \iota_1(\ell) & \text{if } \ell \in \mathit{dom}(\iota_1) \setminus \mathit{dom}(\iota_2) \\ \iota_2(\ell) & \text{if } \ell \in \mathit{dom}(\iota_2) \setminus \mathit{dom}(\iota_1) \\ (\pi_1 + \pi_2, v) & \text{if } \iota_1(\ell) = (\pi_1, v) \wedge \iota_2(\ell) = (\pi_2, v) \wedge \pi_1 + \pi_2 \in (0,1]_{\mathbb{Q}} \end{cases}$$

As one would expect, $\uplus$ is associative and commutative, and $\perp$ is symmetric. Intuitively, if $\iota_1 \perp \iota_2$, then $\iota_1 \uplus \iota_2$ does not lose information w.r.t. ι_1 and ι_2.

The semantics of assertions also relies on a closure operation for closing process terms. The *σ-closure* of any process P is defined as $P[\sigma] \triangleq P[x/\sigma(x)]_{\forall x \in \mathsf{fv}(P)}$.

Definition 8 (Semantics of assertions). *The interpretation of assertions $\iota, \sigma, P \models \mathcal{P}$ is defined by structural induction on $\mathcal{P}$, such that*

$$\begin{array}{lll}
\iota, \sigma, P \models b & \textit{iff} & [\![b]\!]\sigma \\
\iota, \sigma, P \models \forall x.\mathcal{P} & \textit{iff} & \forall v \,.\, \iota, \sigma[x \mapsto v], P \models \mathcal{P} \\
\iota, \sigma, P \models \exists x.\mathcal{P} & \textit{iff} & \exists v \,.\, \iota, \sigma[x \mapsto v], P \models \mathcal{P} \\
\iota, \sigma, P \models \mathcal{P} \vee \mathcal{Q} & \textit{iff} & \iota, \sigma, P \models \mathcal{P} \vee \iota, \sigma, P \models \mathcal{Q} \\
\iota, \sigma, P \models \mathcal{P} * \mathcal{Q} & \textit{iff} & \exists \iota_1, P_1, \iota_2, P_2 \,.\, \iota_1 \perp \iota_2 \wedge \iota = \iota_1 \uplus \iota_2 \wedge P \cong P_1 \parallel P_2 \wedge \\
 & & \quad \iota_1, \sigma, P_1 \models \mathcal{P} \wedge \iota_2, \sigma, P_2 \models \mathcal{Q} \\
\iota, \sigma, P \models \mathcal{P} \mathrel{-\!\!*} \mathcal{Q} & \textit{iff} & \forall \iota', P' \,.\, (\iota \perp \iota' \wedge \iota', \sigma, P' \models \mathcal{P}) \Rightarrow \iota \uplus \iota', \sigma, P \parallel P' \models \mathcal{Q} \\
\iota, \sigma, P \models e_1 \hookrightarrow_\pi e_2 & \textit{iff} & \exists \pi' \,.\, \iota([\![e_1]\!]\sigma) = (\pi', [\![e_2]\!]\sigma) \wedge \pi \leq \pi' \\
\iota, \sigma, P \models \mathsf{Proc}(Q) & \textit{iff} & \exists Q' \,.\, P \cong Q[\sigma] \parallel Q' \\
\iota, \sigma, P \models Q_1 \approx Q_2 & \textit{iff} & Q_1[\sigma] \cong Q_2[\sigma]
\end{array}$$

All assertions are interpreted intuitionistically in the standard sense [35], except for the last two cases, which cover the process-algebraic extensions. Both cases rely on σ-closures to resolve any free variables that may have been introduced by some other proof rules (e.g., the Hoare rule for **recv** may do this).

Process ownership assertions $\mathsf{Proc}(Q)$ are satisfied if there exists a (necessarily closed) process Q', which is the "framed" process that is maintained by the environmental threads, such that P is bisimilar to $Q[\sigma] \parallel Q'$. The intuition here is that P must have at least the behaviour that is described by Q. Finally, $Q_1 \approx Q_2$ is satisfied if Q_1 and Q_2 are bisimilar with respect to the current state.

4.2 Program Judgments

Judgments of programs are of the usual form $\mathcal{I} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ and indicate partial correctness of the program C, where $\mathcal{I} \in \mathit{Assn}$ is known as the *resource invariant* [6]. Their intuitive meaning is that, starting from an initial state satisfying

$$\text{HT-SEND}\quad \mathcal{I} \vdash \{\mathsf{Proc}(\mathsf{send}(e_1, e_2) \cdot P)\}\ \mathbf{send}\ (e_1, e_2)\ \{\mathsf{Proc}(P)\}$$

$$\text{HT-RECV}\quad \frac{x \notin \mathsf{fv}(\mathcal{I}) \cup \mathsf{fv}(P) \qquad y \notin \mathsf{fv}(e)}{\mathcal{I} \vdash \{\mathsf{Proc}(\Sigma_y\, \mathsf{recv}(y, e) \cdot P)\}\ x := \mathbf{recv}\ e\ \{\mathsf{Proc}(P[y/x])\}}$$

$$\text{HT-RECV-WILDCARD}\quad \frac{x_1, x_2 \notin \mathsf{fv}(\mathcal{I}) \cup \mathsf{fv}(P) \qquad \{x_1, y_1\} \cap \{x_2, y_2\} = \emptyset}{\mathcal{I} \vdash \{\mathsf{Proc}(\Sigma_{y_1, y_2}\, \mathsf{recv}(y_1, y_2) \cdot P)\}\ (x_1, x_2) := \mathbf{recv}\ \{\mathsf{Proc}(P[y_1/x_1][y_2/x_2])\}}$$

$$\text{HT-QUERY}\quad \mathcal{I} \vdash \{\mathsf{Proc}(?b \cdot P)\}\ \mathbf{query}\ b\ \{\mathsf{Proc}(P) * b\}$$

Fig. 7. An excerpt of the proof rules for program judgments.

$\mathcal{P} * \mathcal{I}$, the invariant $\mathcal{I}$ is maintained throughout execution of C, and any final state upon termination of C will satisfy $\mathcal{Q} * \mathcal{I}$.

Figure 7 gives an overview of the new proof rules that are specific to handling processes. All other rules are standard in CSL and are therefore deferred to [1].

The HT-SEND rule expresses that, as a precondition, any **send** command in the program must be prescribed by a matching send action in the process-algebraic model. Furthermore, it reduces the process term by ensuring a $\mathsf{Proc}(P)$ predicate, with P the leftover process after the performance of send. The HT-RECV rule is similar in the sense that any $x := \mathbf{recv}\ e$ instruction must be matched by a $\mathsf{recv}(y, e)$ action, but now y can be any message. Process-algebraic summation is used here to quantify over all possible messages to receive, and in the post-state of HT-RECV this message is bound to x—the received message. For wildcard receives both the message *and* the tag are quantified over using summation. Finally, HT-QUERY allows to "query" for properties that are verified during process-algebraic analysis.

4.3 Soundness

The soundness statement of the logic relates axiomatic judgments of programs (Sect. 4.2) to the operational meaning of programs (Sect. 2.1). This soundness argument guarantees freedom of data-races, memory safety, and compliance of pre- and postconditions, for any program for which a proof can be derived. The proof rules of Fig. 7 ensure that every proof derivation encodes that the program synchronises with its process-algebraic model. To formulate the soundness statement, this axiomatic notion of synchronisation thus needs to have a matching operational notion of synchronisation. This is defined in terms of an *instrumented semantics* that executes programs in lock-step with their process-algebraic models. The transition rules are shown in Fig. 8 and are expressed as a labelled binary reduction relation $\cdot \xrightarrow{l}\!\!\!\!\rightarrow \cdot$ between extended program configurations.

$$\frac{P \xrightarrow{\mathit{send}(v_1,v_2)} P' \quad (C,h,\sigma) \xrightarrow{\mathit{send}(v_1,v_2)} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{send}(v_1,v_2)} (C',P',h',\sigma')} \qquad \frac{P \xrightarrow{\mathit{recv}(v_1,v_2)} P' \quad (C,h,\sigma) \xrightarrow{\mathit{recv}(v_1,v_2)} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{recv}(v_1,v_2)} (C',P',h',\sigma')}$$

$$\frac{P \xrightarrow{\mathit{comm}(v_1,v_2)} P' \quad (C,h,\sigma) \xrightarrow{\mathit{comm}(v_1,v_2)} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{comm}(v_1,v_2)} (C',P',h',\sigma')} \qquad \frac{P \xrightarrow{\mathit{assn}} P' \quad (C,h,\sigma) \xrightarrow{\mathit{qry}} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{qry}} (C',P',h',\sigma')}$$

$$\frac{(C,h,\sigma) \xrightarrow{\mathit{cmp}} (C',h',\sigma')}{(C,P,h,\sigma) \xrightarrow{\mathit{cmp}} (C',P,h',\sigma')}$$

Fig. 8. The lock-step execution of programs and process-algebraic models.

The semantics of program judgments is defined in terms of an auxiliary predicate $\mathsf{safe}(C,\iota,\sigma,P,\mathcal{I},\mathcal{Q})$, stating that C: (i) executes safely for any number of execution steps with respect to the abstract program state (ι,σ,P); (ii) will preserve the invariant $\mathcal{I}$ throughout its execution; and (iii) will satisfy the post-condition $\mathcal{Q}$ upon termination. To elaborate on (i), a *safe execution of* C means that C is race free, memory-safe, and synchronises with P with respect to $\dot{\longrightarrow}$.

To relate abstract program state to concrete state, a concretisation operation $\lfloor\cdot\rfloor : \mathit{PermHeap} \to \mathit{Heap}$ is used. The *concretisation* $\lfloor\iota\rfloor$ of a permission heap ι is defined to be the heap $\lambda\ell\,.\,\lfloor\iota(\ell)\rfloor_{\mathsf{cell}}$, with $\lfloor(\pi,v)\rfloor_{\mathsf{cell}} \triangleq v$ for any π.

Definition 9 (Execution safety). *The* safe *predicate is coinductively defined so that, if* $\mathsf{safe}(C,\iota,\sigma,P,\mathcal{I},\mathcal{Q})$ *holds, then*

- *If* $C = \mathbf{skip}$*, then* $\iota,\sigma,P \models \mathcal{Q}$.
- C *cannot perform a* data-race *or* memory violation *from the current state (the exact formal meaning of these notions are deferred to [1]).*
- *For any* $\iota_I,\iota_F,P_I,C',h',\sigma'$ *and* l*, if*
  - *i.* $\iota \perp \iota_I$ *and* $\iota \uplus \iota_I \perp \iota_F$*, and*
  - *ii.* $\neg\mathsf{locked}(C)$ *implies* $\iota_I,\sigma,P_I \models \mathcal{I}$*, and*
  - *iii.* $(P \parallel P_I)\checkmark$ *and* $(C,\lfloor\iota\uplus\iota_I\uplus\iota_F\rfloor,\sigma) \xrightarrow{l} (C',h',\sigma')$*,*

  then there exists ι',ι'_I,P',P'_I *such that*
  - *1.* $\iota' \perp \iota'_I$ *and* $\iota' \uplus \iota'_I \perp \iota_F$ *and* $h' = \lfloor\iota'\uplus\iota'_I\uplus\iota_F\rfloor$*, and*
  - *2.* $\neg\mathsf{locked}(C')$ *implies* $\iota'_I,\sigma,P'_I \models \mathcal{I}$*, and*
  - *3.* $(P' \parallel P'_I)\checkmark$ *and* $(C,P\parallel P_I,\lfloor\iota\uplus\iota_I\uplus\iota_F\rfloor,\sigma) \xrightarrow{l} (C',P'\parallel P'_I,h',\sigma')$*, and*
  - *4.* $\mathsf{safe}(C',\iota',\sigma',P',\mathcal{I},\mathcal{Q})$.

The above definition is based on the similar well-known inductive notion of *configuration safety* of Vafeiadis [35]. Vafeiadis's definition however is coinductive rather than inductive, as this matches more naturally with the coinductive definitions of bisimilarity and process safety. Moreover, it encodes that the program

refines the process with respect to send/receive behaviour: any execution step of the program (*iii*) must be matched by the model (*3*), and vice versa, by definition of $\dot{\longrightarrow}$. Furthermore, the $\mathsf{locked}(C)$ predicate determines whether C is locked. Any program is defined to be *locked* if it executes an atomic (sub)program.

Definition 10 (Semantics of program judgments)

$$\mathcal{I} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \triangleq \forall \iota, \sigma, P.\, P\checkmark \implies \iota, \sigma, P \models \mathcal{P} \implies \mathsf{safe}(C, \iota, \sigma, P, \mathcal{I}, \mathcal{Q}).$$

Theorem 1 (Soundness). $\mathcal{I} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \implies \mathcal{I} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$.

The soundness proof has been fully mechanised using Coq. The Coq development can be found online at [1].

5 Extensions

So far the presented approach only deals with synchronous message passing. However, the principles of the approach allow for easy extensions to also reason about asynchronous message passing, message loss and duplication, and collective operations like barriers and broadcasts, in MPI style [20].

The semantics of asynchronous message passing is that **send**s do not block while waiting for a matching **recv**, but instead push the message onto a message queue that is accessible by both threads. The specification language of mCRL2 is rich enough to model such queues, for example as a separate process $\mathsf{Queue}(\eta)$ with η some data-structure that stores messages in order (e.g., a mapping). Then, rather than letting send and recv communicate directly, they should instead communicate with Queue to push and pop messages into η. So to lift the verification approach to programs with an asynchronous communication semantics, one only has to make minor changes to the mCRL2 translation of processes.

Message loss can be integrated in a similar way, by introducing an extra process that "steals" pending messages. For example, one could analyse the process $P \parallel (\Sigma_{x,t}\, \mathsf{recv}(x,t))^{\omega}$ to reason about P's behaviour with the possibility of message loss. Message duplication can be modelled likewise as an extra process that sends multiple copies of any message it receives. Collective operations may require some extra bookkeeping, for example to administer which threads have already received a broadcasted message. However, all collective operations can be implemented using only sends and receives [19], which means that our approach also extends well to collective communication.

Finally, the current biggest limitation of our approach is that mCRL2 is primarily an explicit-state model checker, which limits its ability to reason symbolically about send/receive behaviour. Nonetheless, mCRL2 also comes with a symbolic back-end [24] that, at the time of writing, can handle specifications of limited complexity. We already have some preliminary results on reasoning symbolically about process-algebraic models, and are actively collaborating with the developers of mCRL2 to improve this.

6 Related Work

There are many modern program logics [9,23,29,33] that provide protocol-like specification mechanisms, to formally describe how shared state is allowed to evolve over time. Notably, Sergey et al. [31] employ this notion in a distributed setting, by using state-transition systems combined with invariants as abstractions for the communication behaviour of distributed programs. All these program logics are however purely theoretical, or can at best be semi-automatically applied via a shallow embedding in Coq. Our approach distinguishes itself by focusing on usability rather than expressivity and targets automated concurrency verifiers instead, like the combination of mCRL2 and Viper.

Francalanza et al. [11] propose a separation logic for message passing programs, where the assertion language has primitives for expressing the contents of communication channels. However, their approach circumvents the need to reason about different thread interleavings by targeting deterministic code, thereby sidestepping an important issue that we address: most problems in realistic distributed programs are the result of intricate thread interleavings. Lei et al. [18] propose a separation logic for modular verification of message passing programs. They achieve modularity via assume-guarantee reasoning, but thereby require users of the logic to manually specify thread interference, which is often non-trivial and non-intuitive. Villard et al. [37] propose a similar approach also based on separation logic, but here the main focus is on transferring heap ownership between threads, using message passing techniques.

Also related are session types [15,16], which are a well-studied type discipline for describing protocols of message passing interaction between processes over communication channels (i.e., sessions). As with our approach, these protocols are specifications of the communication behaviour of processes, and are usually expressed using process algebra, most often (variants of) the π-calculus. However, our approach has a slightly different aim: it uses process algebra not only to structure the communication behaviour, but also to reason about it, and to combine this reasoning with well-known deductive techniques for concurrency verification (viz. CSL) in a sound and practical manner.

This paper builds upon our earlier work [26,27], in which process-algebraic abstractions are used to describe how the heap evolves over time in shared-memory concurrent programs (befitting the notion of protocols given earlier). However, in this paper the abstractions have a different purpose: they instead capture message passing behaviour in a distributed setting. Our abstraction language is therefore different, for example by supporting summation and primitives for communication. Furthermore, in contrast to earlier work, this approach allows to use the result of process-algebraic analysis at intermediate points in the proof derivation of a program, via the novel **query** annotation.

7 Conclusion

This paper demonstrates how a combination of complementary verification techniques can be used to reason effectively about distributed applications, by natu-

rally combining data-oriented reasoning via deductive verification, with temporal reasoning using algorithmic techniques. The approach is illustrated on a small, but intricate example. Our technique uses CSL to reason about data-centric properties of message passing programs, which are allowed to have shared state, and combines this with standard process-algebraic reasoning to verify properties of inter-thread communication. This combination of approaches is proven sound using Coq, and can easily be extended, e.g., to handle asynchronous- and collective communication, message loss, and message duplication.

As future work, we plan to extend process-algebraic reasoning to deal with a reasonable subset of MPI [20], with the goal to develop a comprehensive verification framework that targets real-world programming languages. Moreover, we are actively collaborating with the mCRL2 developers to improve support for symbolic reasoning. We will also apply our approach on larger case studies.

Acknowledgements. This work is partially supported by the NWO VICI 639.023.710 Mercedes project and by the NWO TOP 612.001.403 VerDi project.

References

1. Supplementary material for the paper. https://github.com/utwente-fmt/iFM19-MessagePassingAbstr
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P., Ulbrich, M.: Deductive Software Verification - The KeY Book. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Ahrendt, W., Chimento, J., Pace, G., Schneider, G.: Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. FMSD **51**(1), 200–265 (2017). https://doi.org/10.1007/s10703-017-0274-y
4. Baeten, J.: Process Algebra with Explicit Termination. Eindhoven University of Technology, Department of Mathematics and Computing Science (2000)
5. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
6. Brookes, S.: A semantics for concurrent separation logic. Theoret. Comput. Sci. **375**(1–3), 227–270 (2007). https://doi.org/10.1016/j.tcs.2006.12.034
7. Brookes, S., O'Hearn, P.: Concurrent separation logic. ACM SIGLOG News **3**(3), 47–65 (2016). https://doi.org/10.1145/2984450.2984457
8. Bunte, O., et al.: The mCRL2 toolset for analysing concurrent systems. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 21–39. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_2
9. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14107-2_24
10. Fokkink, W., Zantema, H.: Basic process algebra with iteration: completeness of its equational axioms. Comput. J. **37**(4), 259–267 (1994). https://doi.org/10.1093/comjnl/37.4.259
11. Francalanza, A., Rathke, J., Sassone, V.: Permission-based separation logic for message-passing concurrency. Log. Methods Comput. Sci. **7**, 1–47 (2011). https://doi.org/10.2168/lmcs-7(3:7)2011

12. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: the good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
13. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014)
14. Grumberg, O., Veith, H. (eds.): 25 Years of Model Checking: History, Achievements, Perspectives. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69850-0
15. Honda, K., et al.: Structuring communication with session types. In: Agha, G., et al. (eds.) Concurrent Objects and Beyond. LNCS, vol. 8665, pp. 105–127. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44471-9_5
16. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
17. Hur, C., Dreyer, D., Vafeiadis, V.: Separation logic in the presence of garbage collection. In: LICS, pp. 247–256 (2011). https://doi.org/10.1109/LICS.2011.46
18. Lei, J., Qiu, Z.: Modular reasoning for message-passing programs. In: Ciobanu, G., Méry, D. (eds.) ICTAC 2014. LNCS, vol. 8687, pp. 277–294. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10882-7_17
19. Luo, Z., Zheng, M., Siegel, S.: Verification of MPI programs using CIVL. In: EuroMPI. ACM (2017). https://doi.org/10.1145/3127024.3127032
20. MPI: A Message-Passing Interface standard. http://www.mpi-forum.org/docs. Accessed Apr 2019
21. Milner, R.: Communication and Concurrency. Prentice-Hall Inc., Upper Saddle River (1989)
22. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
23. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G.A.: Communicating state transition systems for fine-grained concurrent resources. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 290–310. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_16
24. Neele, T., Willemse, T.A.C., Groote, J.F.: Solving parameterised Boolean equation systems with infinite data through quotienting. In: Bae, K., Ölveczky, P.C. (eds.) FACS 2018. LNCS, vol. 11222, pp. 216–236. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02146-7_11
25. O'Hearn, P.: Resources, concurrency and local reasoning. Theoret. Comput. Sci. **375**(1–3), 271–307 (2007). https://doi.org/10.1007/978-3-540-28644-8_4
26. Oortwijn, W., Blom, S., Gurov, D., Huisman, M., Zaharieva-Stojanovski, M.: An abstraction technique for describing concurrent program behaviour. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 191–209. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_12
27. Oortwijn, W., Blom, S., Huisman, M.: Future-based static analysis of message passing programs. In: PLACES, pp. 65–72 (2016). https://doi.org/10.4204/EPTCS.211.7
28. Peled, D., Gries, D., Schneider, F. (eds.): Software Reliability Methods. Springer, New York (2001). https://doi.org/10.1007/978-1-4757-3540-6

29. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: a logic for time and data abstraction. In: Jones, R. (ed.) ECOOP 2014. LNCS, vol. 8586, pp. 207–231. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_9
30. de Roever, W., et al.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge University Press, Cambridge (2001)
31. Sergey, I., Wilcox, J., Tatlock, Z.: Programming and proving with distributed protocols. In: POPL, vol. 2 (2017). https://doi.org/10.1145/3158116
32. Shankar, N.: Combining model checking and deduction. Handbook of Model Checking, pp. 651–684. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_20
33. Svendsen, K., Birkedal, L., Parkinson, M.: Modular reasoning about separation of concurrent data structures. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 169–188. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_11
34. Uribe, T.E.: Combinations of model checking and theorem proving. In: Kirchner, H., Ringeissen, C. (eds.) FroCoS 2000. LNCS (LNAI), vol. 1794, pp. 151–170. Springer, Heidelberg (2000). https://doi.org/10.1007/10720084_11
35. Vafeiadis, V.: Concurrent separation logic and operational semantics. MFPS, ENTCS **276**, 335–351 (2011). https://doi.org/10.1016/j.entcs.2011.09.029
36. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_18
37. Villard, J., Lozes, É., Calcagno, C.: Proving copyless message passing. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 194–209. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10672-9_15
38. Wolper, P., Lovinfosse, V.: Verifying properties of large sets of processes with network invariants. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 68–80. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52148-8_6