Practical Abstractions for Automated Verification of Shared-Memory Concurrency

Wytse Oortwijn[1], Dilian Gurov[2], and Marieke Huisman[3]

[1] ETH Zürich, Switzerland
[2] KTH Royal Institute of Technology, Sweden
[3] University of Twente, the Netherlands

Abstract. Modern concurrent and distributed software is highly complex. Techniques to reason about the correct behaviour of such software are essential to ensure its reliability. To be able to reason about realistic programs, these techniques must be modular and compositional as well as practical by being supported by automated tools. However, many existing approaches for concurrency verification are theoretical and focus on expressivity and generality. This paper contributes a technique for verifying behavioural properties of concurrent and distributed programs that makes a trade-off between expressivity and usability. The key idea of the approach is that program behaviour is abstractly modelled using process algebra, and analysed separately. The main difficulty is presented by the typical abstraction gap between program implementations and their models. Our approach bridges this gap by providing a deductive technique for formally linking programs with their process-algebraic models. Our verification technique is modular and compositional, is proven sound with Coq, and has been implemented in the automated concurrency verifier VerCors. Moreover, our technique is demonstrated on multiple case studies, including the verification of a leader election protocol.

1 Introduction

Modern software is typically composed of multiple concurrent components that communicate via shared or distributed interfaces. The concurrent nature of the interactions between (sub)components makes such software highly complex as well as notoriously difficult to develop correctly. To ensure the reliability of modern software, verification techniques are much-needed to aid software developers to comprehend all possible concurrent system behaviours. To be able to reason about *realistic* programs, these techniques must be modular and compositional, but must also be practical by being supported by automated verifiers.

Even though verification of concurrent and distributed software is a very active research field [13,11,50,30,41,44], most work is theoretical and focuses primarily on expressivity and generality. This paper contributes a scalable and practical technique for verifying global behavioural properties of concurrent and distributed programs that makes a trade-off between expressivity and usability: rather than aiming for a unified approach to concurrency reasoning, we propose

a powerful sound technique that is implemented in an automated verification tool, to reason automatically about realistic programs.

Reasoning about complex concurrent program behaviours is only practical if conducted at a suitable level of abstraction that hides irrelevant implementation details. This is because any real concurrent programming language with shared memory, threads and locks, has only very little algebraic behaviour. In contrast, *process algebra* offers an abstract, mathematically elegant way of expressing program behaviour. For this reason, many believe that process algebra provides a language for modelling and reasoning about the behaviour of concurrent programs at a suitable level of abstraction [1]. Our approach therefore uses process algebra as a language for *specifying* program behaviour. Such a specification can be seen as a model, the properties of which can additionally be checked (say, by model checking against temporal logic formulas). The main difficulty of this approach is dealing with the typical abstraction gap between program implementations and their models. The unique contribution of our approach is that it bridges this gap by providing a deductive technique for formally linking programs with their process-algebraic models. These formal links preserve *safety* properties; we leave the preservation of liveness properties for future work.

The key idea of the approach rests in the use of concurrent separation logic to reason not only about data races and memory safety, which is standard, but also about process-algebraic models (i.e., specified program behaviours), viewing the latter as *resources* that can be split and consumed. This results in a modular and compositional approach to establish that a program behaves as specified by its abstract model. Our approach is formally justified by correctness results that have mechanically been proven using Coq, including a machine-checked soundness proof of the proof system, stating that any verified program is a refinement of its abstract model. The verification technique has been been implemented in the VerCors verifier for automated deductive verification of concurrent software [6]. Finally, the approach has been applied on various case studies [34], including a leader election protocol that is included in this paper.

We also recently successfully applied the techniques presented in this paper on an industrial case study, concerning the formal verification of a safety-critical traffic tunnel control system that is currently in use in Dutch traffic [36]. For this case study we made a process algebraic model of the control software that we analysed with mCRL2, and used the techniques presented in this paper to prove that this model is a sound abstraction of the program's behaviour.

An extended version of this paper is available as a technical report [46], which contains more details on the formalisation of the approach and the case study.

Contributions. This paper contributes a verification technique to reason about the behaviour of shared-memory concurrent programs that is modular, compositional, sound (proven with Coq), and implemented in an automated verifier.

First Section 2 illustrates the technique on a small Owicki–Gries example. Then Section 3 gives theoretical justification of the verification technique, as a concurrent separation logic with special constructs to handle process-algebraic

models. Section 4 gives more details on the Coq embedding of the program logic and its soundness proof, and on its implementation in VerCors. Section 5 demonstrates the approach on a larger case study: the verification of a leader election protocol. Finally, Section 6 discusses related work and Section 7 concludes.

2 Approach

We first illustrate the approach on a simple example. In short, we abstractly specify concurrent program behaviour as process algebra terms. Process algebra terms are composed of atomic, indivisible *actions*. In our approach the actions are *logical descriptions of shared-memory modifications*: they describe what changes are allowed to a specified region of shared memory in the program. Actions are then *linked* to the concrete instructions in the program code that compute the memory updates. These links between the program and its abstract model are established deductively, using a concurrent separation logic that is presented later. Well-known techniques for process-algebraic reasoning can then be applied to guarantee safety properties over all possible state changes, as described by their compositions of actions. The novelty of the approach is that these safety properties can then be relied upon in the program logic due to the established formal connection between the program and its process-algebraic model.

Example program. Consider the following program, which is a simple variant of the classical concurrent Owicki–Gries example [38].

$$\textbf{atomic}\ \Big\{X := [E];\ [E] := X + 4\Big\} \ \Big\|\ \textbf{atomic}\ \Big\{Y := [E];\ [E] := Y * 4\Big\}$$

This program consists of two concurrent threads: one that atomically increments the value at heap location E by four, while the other atomically multiplies the value at E by four. The notation $[E]$ denotes *heap dereferencing*, where E is an expression whose evaluation determines the heap location to dereference.

The challenge is to modularly deduce the classical Owicki-Gries postcondition: after termination of both threads, the value at heap location E is either $4 * (old_E + 4)$ or $(4 * old_E) + 4$ (depending on the interleaving of threads), where old_E is the old value at E—the value of E at the pre-state of the computation.

Well-known classical approaches to deal with such concurrent programs [42] include auxiliary state [38] and interference abstraction via rely-guarantee reasoning [20]. Modern program logics employ more intricate constructs, like atomic Hoare triples [41] in the context of TaDa, or higher-order ghost state [23] in the context of Iris. However, the mentioned classical approaches typically do not scale well, whereas such modern, theoretical approaches are hard to integrate into (semi-)automated verifiers like VeriFast or VerCors.

In contrast, our approach is a balanced trade-off between expressivity and usability: it is scalable as well as implemented in an automated deductive verifier.

The approach consists of the following three steps: (1) defining a process-algebraic model $\mathsf{OG} = \mathsf{incr}(4) \parallel \mathsf{mult}(4)$ that is composed out of two actions, incr and mult, that abstract the atomic sub-programs; (2) verifying the Owicki–Gries postcondition algorithmically on the OG process; and (3) deductively verifying that OG is a correct behavioural specification of the program's execution flow (i.e., verifying that all atomic state changes in the program have a corresponding action in OG). The following paragraphs give more detail on these three steps.

Step 1: Specifying program behaviour. The first step is to construct a behavioural specification OG of the example program. This process is defined as the parallel composition of the actions $\mathsf{incr}(4)$ and $\mathsf{mult}(4)$, which specify the behaviour of the atomic increment and multiplication in the program, respectively. In our approach, program behaviour is specified logically, by associating a *contract* to every action. For our example program, incr and mult have the following contract:

requires true;	**requires** true;
ensures $x = \backslash\mathbf{old}(x) + n$;	**ensures** $x = \backslash\mathbf{old}(x) * n$;
action $\mathsf{incr}(\mathbf{int}\ n)$;	**action** $\mathsf{mult}(\mathbf{int}\ n)$;

The variable x is a free, *process-algebraic variable* that is later linked to a concrete heap location in the program (namely E). Moreover, the increment and multiplication of 4 has now been generalised to an arbitrary integer n.

These two actions may be composed into a full behavioural specification of the example program, by also assigning a top-level contract to OG:

requires true;
ensures $x = (\backslash\mathbf{old}(x) + n) * n \vee x = (\backslash\mathbf{old}(x) * n) + n$;
process $\mathsf{OG}(\mathbf{int}\ n) := \mathsf{incr}(n) \parallel \mathsf{mult}(n)$;

Step 2: Process-algebraic reasoning. The next step is to verify that OG satisfies its contract, which can be reduced to standard process-algebraic analysis. We say that OG satisfies its contract if all finite, action contract-complying traces of OG satisfy the ensures clause. The standard approach to analyse OG is to first linearise it to the bisimilar process term $\mathsf{incr}(n) \cdot \mathsf{mult}(n) + \mathsf{mult}(n) \cdot \mathsf{incr}(n)$, and then to prove its correctness by analysing all branches. VerCors currently does the analysis by encoding the linearised process as input to the Viper verifier [29]. VerCors can indeed automatically establish that OG satisfies its postcondition.

Step 3: Deductively linking processes to programs. The key idea of our approach is that, by analysing how contract-complying action sequences change the values of process-algebraic variables, we may indirectly reason about how the content at heap location E evolves over time. So the final step is to project this process-algebraic reasoning onto program behaviour, by annotating the program.

Figure 1 shows the required program annotations. First, x is connected to E by initialising a new model M on line 2 that executes according to $\mathsf{OG}(4)$. The actions incr and mult are then linked to the corresponding subprograms on lines 5–7 and 11–13 by identifying *action blocks* in the code, using special program

```
1  old_E := [E];
2  M := process OG(4) over {x ↦ E};
3  atomic {                          ||  9  atomic {
4    X := [E];                       ||  10   Y := [E];
5    action incr(4) do {             ||  11   action mult(4) do {
6      [E] := X + 4;                 ||  12     [E] := Y * 4;
7    }                               ||  13   }
8  }                                 ||  14 }
15 finish M;
16 assert E ↪¹ (old_E + 4) * 4 ∨ E ↪¹ (old_E * 4) + 4;
```

Fig. 1: The annotated Owicki–Gries example (the annotations are coloured blue).

annotations. We use these **action** annotations to verify in a thread-modular way that the left thread performs the incr(4) action (on lines 5–7) and that the right thread performs mult(4) (lines 11–13). As a result, when the program reaches the **finish** annotation on line 15 all the actions of OG will have been performed. This indirectly means that the content at heap location E has evolved as described by OG, thus allowing the asserted postcondition on line 16 to be derived.

3 Formalisation

This section gives theoretical justification of the verification approach and explains the underlying logical machinery. First, Sections 3.1 and 3.2 briefly discuss the syntax and semantics of process algebraic models and programs, respectively. Then Section 3.3 presents the program logic as a concurrent separation logic with assertions that allow to specify program behaviour as a process algebraic model. Section 3.4 discusses the proof rules. Finally, Section 3.6 discusses soundness of the approach. All these components have been fully formalised in Coq.

Due to space constraints, the technical presentation assumes a certain familiarity with process algebra and separation logic. For more details we refer to the accompanying technical report or to the Coq formalisation [46].

3.1 Process algebraic models

Process algebraic models are defined by the language *Proc* as follows, with $a, b, \cdots \in Act$ the domain of *actions*, $x, y, z, \cdots \in ProcVar$ the domain of *process algebraic variables*, and $m, n, \cdots \in Lit$ the domain of *literals*.

Definition 1 (Process expressions, Process conditions, Processes).

$$
\begin{aligned}
e \in ProcExpr &::= m \mid x \mid e + e \mid e - e \mid \cdots \\
b \in ProcCond &::= \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b \wedge b \mid e = e \mid e < e \mid \cdots \\
P, Q \in Proc &::= \varepsilon \mid \delta \mid a \mid P \cdot Q \mid P + Q \mid P \parallel Q \mid P^*
\end{aligned}
$$

Successful termination

$$\varepsilon\downarrow \qquad \frac{P\downarrow \quad Q\downarrow}{P\cdot Q\downarrow} \qquad \frac{P\downarrow}{P+Q\downarrow} \qquad \frac{Q\downarrow}{P+Q\downarrow} \qquad \frac{P\downarrow \quad Q\downarrow}{P \parallel Q\downarrow} \qquad P^*\downarrow$$

Small-step reduction rules (excerpt)

$$\text{PSTEP-SEQ-L}\ \frac{(P,\sigma)\xrightarrow{a}(P',\sigma')}{(P\cdot Q,\sigma)\xrightarrow{a}(P'\cdot Q,\sigma')} \qquad \text{PSTEP-SEQ-R}\ \frac{P\downarrow \quad (Q,\sigma)\xrightarrow{a}(Q',\sigma')}{(P\cdot Q,\sigma)\xrightarrow{a}(Q',\sigma')} \qquad \text{PSTEP-ACT}\ \frac{[\![\mathsf{pre}(a)]\!](\sigma) \quad [\![\mathsf{post}(a)]\!](\sigma')}{(a,\sigma)\xrightarrow{a}(\varepsilon,\sigma')}$$

Fig. 2: The small-step operational semantics of process algebraic models.

As usual, ε is the empty process that has no behaviour, whereas δ is the deadlocked process that neither progresses nor terminates. The process $P \cdot Q$ is the sequential composition of P and Q, while $P + Q$ denotes their non-deterministic choice. The process $P \parallel Q$ is the parallel composition P and Q. Finally, P^* is the Kleene iteration of P and denotes a sequence of zero or more P's.

The verification approach uses process algebraic models in the presence of data, implemented via *action contracts*. These action contracts make the process algebra language non-standard. Action contracts consist of pre- and postconditions that logically describe the state changes imposed by the action. Each action is assumed to have an associated contract that can be obtained via the functions $\mathsf{pre}, \mathsf{post} : \mathit{Act} \to \mathit{ProcCond}$. All pre- and postconditions are of type *ProcCond*, which is the domain of Boolean expressions over process algebraic variables.

Semantics. The operational semantics of processes is expressed as a binary reduction relation $\cdot \xrightarrow{\cdot} \cdot \subseteq \mathit{ProcConf} \times \mathit{Act} \times \mathit{ProcConf}$ over *process configurations* $\mathit{ProcConf} \triangleq \mathit{Proc} \times \mathit{ProcStore}$, labelled with actions from *Act*. The notion of data is implemented via *process stores* $\sigma \in \mathit{ProcStore} \triangleq \mathit{ProcVar} \to \mathit{Val}$ that map process algebraic variables to a semantic domain *Val* of *values*.

Most of the reduction rules are standard. Figure 2 gives an overview of the non-standard rules. All other transition rules are deferred to [46].

To define the transition rule PSTEP-SEQ-R for sequential composition, it is common in process algebra with ε to use an explicit notion of *successful termination* [4]. Successful termination $P\downarrow$ of any process P intuitively means that P has the choice to have no further behaviour and thus to behave as ε. Furthermore, the PSTEP-ACT transition rule for action handling permits state to change in any way that complies with the corresponding action contract.

The program logic allows one to handle process algebraic models *up to bisimulation*. We write $P \cong Q$ to denote that P and Q are *bisimilar* (i.e., behaviourally equivalent). Bisimilarity is a congruence with respect to all process algebraic connectives. Moreover, we indeed have that $P\downarrow$ implies $P \cong P + \varepsilon$ for any P.

STEP-PROC-INIT
$(X := \textbf{process}\ P\ \textbf{over}\ \Pi, h, s) \rightsquigarrow (\textbf{skip}, h, s)$

STEP-PROC-FINISH
$(\textbf{finish}\ X, h, s) \rightsquigarrow (\textbf{skip}, h, s)$

STEP-ACT
$$\frac{(C, h, s) \rightsquigarrow (C', h', s')}{(\textbf{action}\ X.a\ \textbf{do}\ C, h, s) \rightsquigarrow (\textbf{action}\ X.a\ \textbf{do}\ C', h', s')}$$

STEP-ACT-FINISH
$(\textbf{action}\ X.a\ \textbf{do}\ \textbf{skip}, h, s) \rightsquigarrow (\textbf{skip}, h, s)$

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

3.2 Programs

Our approach is formalised on the following simple concurrent pointer language, where $X, Y, Z, \cdots \in \mathit{Var}$ are *(program) variables*.

Definition 2 (Expressions, Conditions, Programs).

$$\begin{aligned}
E \in \mathit{Expr} &::= n \mid X \mid E + E \mid E - E \mid \cdots \\
B \in \mathit{Cond} &::= \mathsf{true} \mid \mathsf{false} \mid \neg B \mid B \wedge B \mid E = E \mid E < E \mid \cdots \\
\Pi \in \mathit{AbstrBinder} &::= \{x_0 \mapsto E_0, \ldots, x_n \mapsto E_n\} \\
C \in \mathit{Cmd} &::= \textbf{skip} \mid X := E \mid X := [E] \mid [E] := E \mid C; C \mid C \parallel C \\
&\quad \mid X := \textbf{alloc}\ E \mid \textbf{dispose}\ E \mid \textbf{atomic}\ C \\
&\quad \mid \textbf{if}\ B\ \textbf{then}\ C\ \textbf{else}\ C \mid \textbf{while}\ B\ \textbf{do}\ C \\
&\quad \mid X := \textbf{process}\ P\ \textbf{over}\ \Pi \mid \textbf{action}\ X.a\ \textbf{do}\ C \mid \textbf{finish}\ X
\end{aligned}$$

This language is a variation of the language proposed by [32,9], extended with *specification-only* commands (displayed in blue) for handling process algebraic models in the logic. These commands are ignored during program execution.

Specification-wise, $X := \textbf{process}\ P\ \textbf{over}\ \Pi$ initialises a new process algebraic model that is represented by the process term P, with Π a finite mapping from process algebraic variables to heap locations. Π is used to connect abstract state (i.e., the state of process algebraic models) to concrete program state (i.e., heap entries) and is therefore referred to as an *abstraction binder*.

The $\textbf{finish}\ X$ command *concludes* the model that is identified by X, given that the associated process successfully terminates. By *concludes* we mean that the model's postcondition can be relied upon and used in the proof system.

Finally, $\textbf{action}\ X.a\ \textbf{do}\ C$ executes the command C in the context of the abstract model X as the action a. In particular, this specification command states that, by executing C (according to the operational semantics of programs), the action a is executed in the specified process algebraic model.

Semantics. The operational semantics of programs is expressed as a small-step reduction relation $\cdot \rightsquigarrow \cdot \subseteq \mathit{Conf} \times \mathit{Conf}$, between *(program) configurations* $\mathit{Conf} \triangleq \mathit{Cmd} \times \mathit{Heap} \times \mathit{Store}$. Program configurations $(C, h, s) \in \mathit{Conf}$ consist

of a program C, as well as a heap $h \in \mathit{Heap} \triangleq \mathit{Val} \rightharpoonup_{\mathsf{fin}} \mathit{Val}$ that models shared memory and a store $s \in \mathit{Store} \triangleq \mathit{Var} \to \mathit{Val}$ that models thread-local memory.

Figure 3 shows an excerpt of the new reduction rules for ghost commands. All other reduction rules are standard in spirit and are deferred to [46].

Most importantly, all ghost commands are specification constructs: they do not affect the program state and are essentially handled as if they were comments. However, observe that STEP-PROC-FINISH and STEP-ACT-FINISH are auxiliary transition steps that reduce a finished process or action to **skip**. These are not strictly needed, but make it more convenient to prove soundness of the logic.

3.3 Program Logic

Our program logic builds on *intuitionistic*[4] concurrent separation logic (CSL), where the assertion language is defined by the following grammar.

Definition 3 (Assertions).

$$\begin{aligned} t \in \mathit{PointsToType} &::= \mathsf{std} \mid \mathsf{proc} \mid \mathsf{act} \\ \mathcal{P}, \mathcal{Q}, \mathcal{R}, \cdots \in \mathit{Assn} &::= B \mid \forall X.\mathcal{P} \mid \exists X.\mathcal{P} \mid \mathcal{P} \vee \mathcal{Q} \mid \mathcal{P} * \mathcal{Q} \mid \mathcal{P} \mathrel{-\!\!*} \mathcal{Q} \\ &\quad \mid *_{i \in I} \mathcal{P}_i \mid E \overset{\pi}{\hookrightarrow}_t E \mid \mathsf{Proc}_\pi(X, b, P, \mathit{\Pi}) \end{aligned}$$

The assertion $\mathcal{P} * \mathcal{Q}$ is the separating conjunction of separation logic and states that $\mathcal{P}$ and $\mathcal{Q}$ hold on *disjoint* parts of the heap. This for example means that $\mathcal{P}$ and $\mathcal{Q}$ cannot both express write access to the same heap entry. The assertion $*_{i \in I} \mathcal{P}_i$ is the *iteration of* $*$ and is equivalent to $\mathcal{P}_0 * \cdots * \mathcal{P}_n$ given that $I = \{0, \ldots, n\}$. Furthermore, the $\mathrel{-\!\!*}$ connective from separation logic is known as the *magic wand* and expresses hypothetical modifications of the current state.

Apart from these standard CSL connectives, the assertion language contains three different heap ownership predicates $\overset{\pi}{\hookrightarrow}_t$, with $\pi \in (0, 1]_{\mathbb{Q}}$ a *fractional permission* in the style of Boyland [8] and t the *heap ownership type*, where:

- $E \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E'$ is the *standard heap ownership predicate* from separation logic, that provides read-only access for $0 < \pi < 1$ and write access in case $\pi = 1$.
- $E \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E'$ is the *process heap ownership predicate*, which indicates that the heap location E is bound to an active process algebraic model, but in a *read-only* manner: it only provides read-only access, even when $\pi = 1$.
- $E \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E'$ is the *action heap ownership predicate*, which indicates that the heap location E is bound by an active process algebraic model and is used in the context of an action block, in a *read/write* manner.

The distinction between different types of heap ownership is needed for the program logic to be sound, for example to disallow the deallocation of memory that is bound by a process algebraic model. Moreover, observe that $E \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E'$ predicates never provide write access to E. However, we shall later see that the

[4] This intuitively means that the program logic is able to "forget" about resources, which fits naturally with garbage collecting languages like Java and C#.

proof system allows one to upgrade $\hookrightarrow^{\pi}_{\mathsf{proc}}$ predicates to $\hookrightarrow^{\pi}_{\mathsf{act}}$ inside **action** blocks, and $\hookrightarrow^{\pi}_{\mathsf{act}}$ again provides write access if $\pi = 1$. This system of upgrading enforces that all modifications to E happen in the context of **action** $X.a$ **do** C commands, and can therefore be recorded in the model X as the action a.

Finally, the $\mathsf{Proc}_\pi(X, b, P, \Pi)$ assertion expresses ownership of the program model that is identified by X and is represented by the process P. The condition b is the postcondition of the abstract model. Furthermore, Π connects the abstract model to the concrete program, by mapping the models' process algebraic variables to heap locations in the program. And last, the fractional permission π is needed to implement the ownership system of program models. Fractional permissions are only used here to be able to reconstruct the full Proc_1 predicate.

Semantics of assertions. The interpretation of assertions is defined as a modelling relation $ph, pm, s, g \models \mathcal{P}$, where the models (ph, pm, s, g) consist of the following four components:

- A *permission heap*, $ph \in \mathit{PermHeap} \triangleq \mathit{Var} \to \mathsf{free} \mid \langle v \rangle^{\pi}_{t}$, that maps values (heap locations) to either free (unoccupied) or to occupied entries $\langle v \rangle^{\pi}_{t}$. Occupied heap cells store a value v, as well as a type t to associate heap cells to the three different kinds of heap ownership predicates used in the logic.
- A *process map*, $pm \in \mathit{ProcMap} \triangleq \mathit{Var} \to \mathsf{free} \mid \langle b, P, \Lambda \rangle^{\pi}$, defined as a total mapping from values (process identifiers) to *process map entries*. Occupied entries have the form $\langle b, P, \Lambda \rangle^{\pi}$ and model ownership of process algebraic models in the program logic. The components $\Lambda \in \mathit{ProcVar} \rightharpoonup_{\mathsf{fin}} \mathit{Val}$ in turn define the models of the abstraction binders (that were defined in Def. 2).
- Two stores, $s, g \in \mathit{Store}$, that gives an interpretation to all variables used in program and ghost code, respectively. Ghost variables do not interfere with regular program execution and are therefore separated from program variables and maintained in an extra store g, referred to as the *ghost store*.

The semantics of assertions is defined as a modelling relation $\cdot \models \cdot$ between models of the logic $\mathit{PermHeap} \times \mathit{ProcMap} \times \mathit{Store}^2$ and assertions Assn as follows:

Definition 4 (Semantics of assertions (excerpt)). *The interpretation of assertions $ph, pm, s, g \models \mathcal{P}$ is defined by structural recursion on $\mathcal{P}$ in the standard way, except for the following two cases:*

$$
\begin{aligned}
ph, pm, s, g &\models E_1 \hookrightarrow^{\pi}_{t} E_2 && \textit{iff} \quad ph(\llbracket E_1 \rrbracket(s)) = \langle \llbracket E_2 \rrbracket(s) \rangle^{\pi'}_{t} \wedge \pi \leq \pi' \\
ph, pm, s, g &\models \mathsf{Proc}_\pi(X, b, P, \Pi) && \textit{iff} \quad \exists P' \,.\, pm(g(X)) = \langle b, P \parallel P', \llbracket \Pi \rrbracket(s) \rangle^{\pi'} \\
& && \qquad \wedge \pi \leq \pi' \wedge (\pi = 1 \implies P' = \varepsilon)
\end{aligned}
$$

The full definition of the semantics of assertions can be found in [46].

Clarifying the non-standard cases, $E \hookrightarrow^{\pi}_{t} E'$ is satisfied if ph holds an entry at location E that matches with the ownership type t, with an associated fractional permission that is at least π. Process ownership assertions $\mathsf{Proc}_\pi(X, b, P, \Pi)$ are satisfied if pm holds a matching entry with a fractional permission at least π, as

$$\hookrightarrow\text{-SPLITMERGE}\quad E_1 \overset{\pi_1+\pi_2}{\hookrightarrow}_t E_2 \dashv\vdash E_1 \overset{\pi_1}{\hookrightarrow}_t E_2 * E_1 \overset{\pi_2}{\hookrightarrow}_t E_2$$

$$\text{PROC-SPLITMERGE}\quad \mathsf{Proc}_{\pi_1+\pi_2}(X, b, P_1 \parallel P_2, \Pi) \dashv\vdash \mathsf{Proc}_{\pi_1}(X, b, P_1, \Pi) * \mathsf{Proc}_{\pi_2}(X, b, P_2, \Pi)$$

$$\text{PROC-}\cong\quad \frac{P \cong Q}{\mathsf{Proc}_{\pi}(X, b, P, \Pi) \dashv\vdash \mathsf{Proc}_{\pi}(X, b, Q, \Pi)}$$

Fig. 4: Selected entailment rules of the program logic.

well as a process that has at least the behaviour of P. The denotation $[\![\Pi]\!](s)$ gives the model of the abstraction binder Π, and is defined as follows:

Definition 5 (Semantics of abstraction binders).

$$[\![\{x_0 \mapsto E_0, \ldots, x_n \mapsto E_n\}]\!](s) \triangleq \{x_0 \mapsto [\![E_0]\!](s), \ldots, x_n \mapsto [\![E_n]\!](s)\}$$

3.4 Entailment Rules

Figure 4 shows the non-standard entailment rules of the program logic. All other, standard rules can be found in [46]. The notation $\mathcal{P} \dashv\vdash \mathcal{Q}$ is a shorthand notation for $\mathcal{P} \vdash \mathcal{Q}$ and $\mathcal{Q} \vdash \mathcal{P}$, and indicates that the rule can be used in both directions. All rules have shown to be sound in the standard sense, using Coq.

Clarifying the entailment rules, $\hookrightarrow$-SPLITMERGE expresses that heap ownership predicates $\overset{\pi}{\hookrightarrow}_t$ of any type t may be *split* (in the left-to-right direction) and be *merged* (right-to-left) along π. This allows one to distribute heap ownership among the different threads in the program. Likewise, PROC-SPLITMERGE allows one to split and merge process ownership along parallel compositions inside abstract models, to distribute them over different threads. More specifically, by splitting a predicate $\mathsf{Proc}_{\pi_1+\pi_2}(X, b, P_1 \parallel P_2, \Pi)$ into two, both parts can be distributed over different concurrent threads, so that thread i can establish that it executes as prescribed by its part $\mathsf{Proc}_{\pi_i}(X, b, P_i, \Pi)$ of the abstraction. Afterwards, when the threads join again, the remaining partial abstractions can be merged back into a single predicate. This system thus provides a compositional way of verifying that programs meet their abstract models.

Finally, PROC-$\cong$ allows one to replace program abstractions by bisimilar ones. This rule is used to rewrite processes in a canonic form used by some other rules.

3.5 Program Judgments

Judgments of programs are defined as sequents of the form $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$, where $\mathcal{R}$ is a *resource invariant* [9], and Γ is a *process environment*:

HT-PROCINIT

$$\frac{\begin{array}{c}\mathsf{fv}(b_1) \subseteq \mathsf{dom}(\Pi) = \{x_0, \ldots, x_n\} \\ I = \{0, \ldots, n\} \qquad X \notin \mathsf{fv}(\mathcal{R}, E_0, \ldots, E_n) \qquad B = b_1[x_i/E_i]_{\forall i \in I}\end{array}}{\Gamma, \{b_1\}\, P\, \{b_2\}; \mathcal{R} \vdash \begin{array}{c}\left\{ *_{i \in I} \Pi(x_i) \hookrightarrow^{1}_{\mathsf{std}} E_i * B \right\} \\ X := \textbf{process}\ P\ \textbf{over}\ \Pi \\ \left\{ \begin{array}{c} *_{i \in I} \Pi(x_i) \hookrightarrow^{1}_{\mathsf{proc}} E_i * B * \\ \mathsf{Proc}_1(X, b_2, P, \Pi) \end{array} \right\}\end{array}}$$

HT-PROCUPDATE

$$\frac{\begin{array}{c}\mathsf{fv}(a) = \{x_0, \ldots, x_n\} \subseteq \mathsf{dom}(\Pi) \qquad I = \{0, \ldots, n\} \\ B_1 = \mathsf{pre}(a)[x_i/E_i]_{\forall i \in I} \qquad B_2 = \mathsf{post}(a)[x_i/E_i]_{\forall i \in I} \\ \Gamma; \mathcal{R} \vdash \{ *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{act}} E_i * B_1 * \mathcal{P} \}\, C\, \{ *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{act}} E_i' * B_2 * \mathcal{Q} \}\end{array}}{\Gamma; \mathcal{R} \vdash \begin{array}{c}\left\{ \begin{array}{c} *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i * B_1 * \\ \mathsf{Proc}_\pi(X, b, a \cdot P + Q, \Pi) * \mathcal{P} \end{array} \right\} \\ \textbf{action}\ X.a\ \textbf{do}\ C \\ \left\{ \begin{array}{c} *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i' * B_2 * \\ \mathsf{Proc}_\pi(X, b, P, \Pi) * \mathcal{Q} \end{array} \right\}\end{array}}$$

HT-PROCFINISH

$$\frac{\mathsf{fv}(b) \subseteq \mathsf{dom}(\Pi) = \{x_0, \ldots, x_n\} \qquad I = \{0, \ldots, n\} \qquad B = b[x_i/E_i]_{\forall i \in I} \qquad P{\downarrow}}{\Gamma; \mathcal{R} \vdash \left\{ \begin{array}{c} *_{i \in I} \Pi(x_i) \hookrightarrow^{1}_{\mathsf{proc}} E_i * \\ \mathsf{Proc}_1(X, b, P, \Pi) \end{array} \right\} \textbf{finish}\ X \left\{ *_{i \in I} \Pi(x_i) \hookrightarrow^{1}_{\mathsf{std}} E_i * B \right\}}$$

Fig. 5: The non-standard Hoare proof rules related to abstract models.

Definition 6 (Process environment).

$$\Gamma ::= \emptyset \mid \Gamma, \{b\}\, P\, \{b\}$$

Process environments are defined in the style of *interface specifications* [33], and are essentially a series of Hoare-triples $\{b_1\}\, P\, \{b_2\}$ for processes P, that constitute the top-level contracts of the programs' abstract models.

The intuitive meaning of a program judgment $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ is that, starting from any state satisfying $\mathcal{P} * \mathcal{R}$, the invariant $\mathcal{R}$ is maintained throughout execution of C, and any final state upon termination of C will satisfy $\mathcal{Q} * \mathcal{R}$. Moreover, the proof derivation of C may use any abstract model that is in Γ.

Figure 5 presents the proof rules that handle process algebraic abstractions. All other proof rules are deferred to [46] due to space constraints.

The HT-PROCINIT rule handles initialisation of an abstract model P over a set of heap locations as specified by Π. Standard points-to predicates with write-permission are required for any heap location that is to be bound by P, and these are converted to $\hookrightarrow^{1}_{\mathsf{proc}}$. Moreover, HT-PROCINIT requires that the precondition of P holds, which is constructed from b_1 by replacing all process variables by the symbolic values at the corresponding heap locations. A Proc_1 predicate with full permission is ensured, containing the postcondition b_2 of the abstract model.

The HT-PROCUPDATE rule handles updates to program abstractions, by performing an action a in the context of an **action** $X.a$ **do** C program, provided that C respects the contract of a. As a precondition, a predicate of the form $\mathsf{Proc}_\pi(X, b, a \cdot P + Q, \Pi)$ is required for some π. The process component of this predicate must be of the form $a \cdot P + Q$ to allow performing the a action. After performing a, this process component will be reduced to P, thereby discarding Q as the choice is made not to follow execution as prescribed by Q. In order to get process components into the required format $a \cdot P + Q$, the PROC-$\cong$ rule can be used to rewrite process components up to bisimilarity. Furthermore, $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates are required for any heap location that is bound by Π. These points-to predicates are needed to resolve the pre- and postcondition of a.

Finally, HT-PROCFINISH handles finalisation of program models that successfully terminate. A predicate $\mathsf{Proc}_1(X, b, P, \Pi)$ with *full* permission is required, which means that no other thread can have any fragment of the model. This predicate is exchanged for the postcondition of the abstraction. This postcondition can be established, since (i) the contracts of processes in Γ are assumed as their validity is checked externally, and b is a postcondition of one of these contracts; (ii) the abstraction has been initialised in a state satisfying the precondition of that contract; and (iii) the leftover process P is able to successfully terminate. Lastly, all $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$ predicates are converted back to $\overset{1}{\hookrightarrow}_{\mathsf{std}}$ to indicate that the associated heap locations are no longer bound by the abstraction.

3.6 Soundness

The soundness proof of the program logic has been fully mechanised using the Coq proof assistant, as a deep embedding that is inspired by [53]. The overall Coq implementation comprises roughly 15.000 lines of code. Proving soundness was non-trivial and required substantial auxiliary definitions. The Coq development and its documentation can be found at [46].

The soundness theorem relates program judgments to the operational semantics of programs, and amounts to the following: if a proof $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ can be derived for any program C, and if the contracts in Γ of all abstract models of C are satisfied, then C *executes safely for any number of computation steps*. To concretise this, we first define the semantics of program judgments.

Definition 7 (Semantics of program judgments).

$$\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \triangleq \models \Gamma \implies \forall n, ph, pm, s, g\,. \\ ph, pm, s, g \models \mathcal{P} \implies \mathsf{safe}^n_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$$

The entailment $\models \Gamma$ intuitively means that, for any Hoare triple $\{b_1\}\, P\, \{b_2\}$ in Γ and for any σ such that $[\![b_1]\!](\sigma)$, we have that any run $(P, \sigma) \longrightarrow^* (P', \sigma')$ that terminates (i.e., $P'{\downarrow}$) ends up with a store σ' for which $[\![b_2]\!](\sigma')$ holds.

The predicate safe^n_Γ defines execution safety for n computation steps, meaning that the program is: data-race free, memory safe, complies with its pre- and postconditions, and refines its process algebraic models, for n computation

steps. This definition extends the well-known inductive definition of configuration safety of Vafeiadis [53] by adding machinery to handle process algebraic models. The most important extension is a *simulation argument* between program execution (with respect to $\rightsquigarrow$) and the execution of all active models (with respect to $\xrightarrow{a}$). However, as the reduction steps of these two semantics do not directly correspond one-to-one, this simulation is established via an intermediate, instrumented semantics. This intermediate semantics is defined in terms of $\rightsquigarrow_{\mathsf{ghost}}$ transitions that define the *lock-step execution* of program transitions $\rightsquigarrow$ and the transitions $\xrightarrow{a}$ of their abstractions. Our definition of "*executing safely for n execution steps*" includes that all $\rightsquigarrow$ steps can be simulated by $\rightsquigarrow_{\mathsf{ghost}}$ steps and vice versa, for n execution steps. Thus, the end-result is a refinement between programs and their abstract models.

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

The underlying idea of the above definition, i.e., having a continuation-passing style definition for program judgments, has first been applied in [2] and has further been generalised in [16] and [17]. Moreover, the idea of defining (program) execution safety in terms of an inductive predicate originates from [3]. These two concepts have been reconciled in [53] into a formalisation for the classical CSL of Brookes [9], that has been encoded and mechanically been proven in both Isabelle and Coq. Our definition builds on the latter, by having a refinement between programs and abstractions encoded in safe.

4 Implementation

The verification approach has been implemented in the VerCors verifier, which specialises in automated verification of parallel and concurrent programs written in high-level languages, like (subsets of) Java and C [6]. VerCors applies a correctness-preserving translation of the input program into a sequential imperative language, and delegates the generation of verification conditions to the Viper verifier [29] and their verification ultimately to Z3.

Tool support for our technique has been implemented in VerCors for languages with fork/join concurrency and statically-scoped parallel constructs [34]. This is done defining an axiomatic domain for processes in Viper, consisting of constructors for all process-algebraic connectives, supported by standard process-algebraic axioms. The Proc_π assertions are encoded as predicates over these process types. The three different ownership types $\overset{\pi}{\hookrightarrow}_t$ are encoded by defining extra fields that maintain the ownership status t for each global reference.

To analyse process-algebraic models, VerCors first linearises all processes and then encodes the linear processes and their contracts into Viper. The linearisation algorithm is based on a rewrite system that uses a subset of the standard process-algebraic axioms as rewrite rules [51] to eliminate parallel connectives.

The VerCors implementation of the abstraction approach is much richer than the simple language of Section 3 that is used to formalise the approach on. Notably, the abstraction language in VerCors supports general recursion instead

of Kleene iteration, and allows parameterising process and action declarations by data. VerCors also has support for several axiomatic data types that enrich the expressivity of reasoning with abstractions, like (multi)sets and sequences.

5 Case Study

Finally, we demonstrate our verification approach on a well-known version of the leader election protocol [35] that is based on shared memory. Most importantly, this case study shows how our approach bridges the typical abstraction gap between process algebraic models and program implementations. In particular, it shows how a high-level process algebraic model of a leader election protocol, together with a contract for this model (checked with mCRL2 for various inputs), is formally connected to an actual program implementation of the protocol.

The protocol is performed by N concurrent workers that are organised in a ring, so that worker i only sends to worker $i+1$ and only receives from worker $i-1$, modulo N. The goal is to determine a leader among these workers. To find a leader, the election procedure assumes that each worker i receives a unique integer value to start with, and then operates in N rounds. In every round (i) each worker sends the highest value it encountered so far to its right neighbour, (ii) receives a value from its left neighbour, and (iii) remembers the highest of the two. The result after N rounds is that all workers know the highest unique value in the network, allowing its original owner to announce itself as leader.

The case study has been verified with VerCors using the presented approach. All workers communicate via two standard non-blocking operations for message passing: $\texttt{mp\_send}(r, msg)$ for sending a message msg to the worker with rank r[5], and $msg := \texttt{mp\_recv}(r)$ for receiving a message from worker r. The election protocol is implemented on top of this message passing system.

The main challenge of this case study is to define a message passing system on the process algebra level that matches this implementation. To design such a system we follow the ideas of [35]; by defining two actions, $\mathsf{send}(r, msg)$ and $\mathsf{recv}(r, msg)$, that abstractly describe the behaviour of the concrete implementations in `mp_send` and `mp_recv`, respectively. Moreover, *process algebraic summation* $\Sigma_{x \in D} P$ is used to quantify over the possible messages that `mp_recv` might receive. The summation operator $\Sigma_{x \in D} P$ quantifies over a set $D = \{d_0, \ldots, d_n\}$ of data and is defined as the (finite) sequence $P[x/d_0] + \cdots + P[x/d_n]$ of non-deterministic choices. The following two rules illustrate how the abstract send and recv actions are connected to `mp_send` and `mp_recv` (observe that both these actions are parameterised by data[6]).

$$\{\mathsf{send}(r, msg) \cdot P\}\ \texttt{mp\_send}(r, msg)\ \{P\}$$
$$\{\Sigma_{x \in Msg} \mathsf{recv}(r, x) \cdot P\}\ msg := \texttt{mp\_recv}(r)\ \{P[x/msg]\}$$

[5] The identifiers of workers are typically called *ranks* in message passing terminology.
[6] Recall that the VerCors implementation of our abstraction technique is much richer than the simple language of Section 3 that is used to formalise the approach on.

```
1  seq⟨seq⟨Msg⟩⟩ chan; // communication channels between workers
2  int lead; // rank of the worker that is announced as leader
3
4  /* Action for sending messages. */
5  requires 0 ≤ rank < |chan|;
6  ensures chan[rank] = \old(chan[rank]) + {msg};
7  ensures ∀r' : int . (0 ≤ r' < |chan| ∧ r' ≠ rank) ⇒ chan[r'] = \old(chan[r']);
8  action send(int rank, Msg msg);
9
10 /* Action for receiving messages. */
11 requires 0 ≤ rank < |chan|;
12 ensures {msg} + chan[rank] = \old(chan[rank]);
13 ensures ∀r' : int . (0 ≤ r' < |chan| ∧ r' ≠ rank) ⇒ chan[r'] = \old(chan[r']);
14 action recv(int rank, Msg msg);
15
16 /* Action for announcing a leader. */
17 requires 0 ≤ rank < |chan|;
18 ensures lead = rank;
19 action announce(int rank);
20
21 /* Local behavioural specification for each worker. */
22 requires 0 ≤ n ≤ |chan| ∧ 0 ≤ rank < |chan|;
23 process Elect(int rank, Msg v0, Msg v, int n) ≜
24   if 0 < n then send((rank + 1) % |chan|, v) ·
25     Σ_{v'∈Msg} recv(rank, v') · Elect(rank, v0, max(v, v'), n − 1)
26   else (if v = v0 then announce(rank) else ε);
27
28 /* Global behavioural specification of the election protocol. */
29 requires |vs| = |chan|;
30 requires ∀i, j : int . (0 ≤ i < |vs| ∧ 0 ≤ j < |vs| ∧ vs[i] = vs[j]) ⇒ i = j;
31 ensures |vs| = |chan| ∧ 0 ≤ lead < |vs|;
32 ensures ∀i : int . (0 ≤ i < |vs|) ⇒ vs[i] ≤ vs[lead];
33 process ParElect(seq⟨Msg⟩ vs) ≜
34   Elect(0, vs[0], vs[0], |vs|) ‖ ··· ‖ Elect(|vs|−1, vs[|vs|−1], vs[|vs|−1], |vs|);
```

Fig. 6: Behavioural specification of the leader election protocol.

Finally, we construct a process-algebraic model of the election protocol using send and recv, and verify that the implementation adheres to this model. This model has been analysed with mCRL2 for various inputs (since mCRL2 is essentially finite-state) to establish the global property of announcing the correct leader. The deductive proof of the program can then rely on this property.

5.1 Behavioural Specification

Our main goal is proving that the implementation determines the correct leader upon termination. To prove this, we first define a *behavioural specification* of the election protocol that hides all irrelevant implementation details, and prove

the correctness property on this specification. Process algebra provides a proper abstraction level that suits our needs well, as the behaviour of leader election can concisely be specified in terms of sequences of sends and receives.

Figure 6 presents the process algebraic specification. In particular, ParElect specifies the *global* behaviour whereas Elect specifies the *thread-local* behaviour. The ParElect process encodes the parallel composition of all eligible participants. ParElect takes a sequence vs of initial values as argument, whose length equals the total number of workers by its precondition. ParElect's postcondition states that $lead$ must be a valid rank after termination and that $vs[lead]$ be the highest initial worker value. It follows that worker $lead$ is the correctly chosen leader.

The Elect process takes four arguments, which are: the rank of the worker, the initial unique value v_0 of that worker, the current highest value v encountered by that worker, and finally the number n of remaining rounds. The rounds are implemented via general recursion. In each round all workers send their current highest value v to their right neighbour (on line 24), receive a value v' in return from their left neighbour (line 25), and continue with the highest of the two. The extra announce action is declared and used to announce the leader after n rounds. The postcondition of announce is that $lead$ stores the leader's rank.

The contracts of send and recv describe the behaviour of standard non-blocking message passing. Communication on the specification level is implemented via *message queues*. Message queues are defined as sequences of messages that are taken from a finite domain Msg. Since workers are organised in a ring in this case, every worker can do with only a single queue and the global communication channel architecture can be defined as a sequence of message queues: $chan$ in the figure. The action contract of send(r, msg) expresses enqueuing the message msg onto the message queue $chan[r]$ of the worker with rank r. The postcondition of send is that msg has been enqueued onto $chan[r]$ and that the queues $chan[r']$ for any $r' \neq r$ have not been altered. Likewise, the contract of recv(r, msg) expresses dequeuing msg from $chan[r]$. The expression $\backslash\mathbf{old}(e)$ indicates that e is to be evaluated with respect to the pre-state of computation.

5.2 Protocol Implementation

Figure 7 presents the annotated implementation of the election protocol[7]. The `elect` method contains the code that is executed by every worker. The contract of `elect`(X, $rank$, v_0, v) states that the method body adheres to the behavioural description Elect($rank$, v_0, v, N) of the election protocol. Each worker performing `elect` enters a **for**-loop that iterates N times, whose loop invariant states that, at iteration i, the remaining program behaves as prescribed by the process Elect($rank$, v_0, v, $i-1$). The invocations to `mp_send` and `mp_recv` on

[7] It should be noted that the presentation is slightly different from the version that is verified by VerCors, to better connect to the theory discussed in the earlier sections to the case study. Notably, VerCors uses Implicit Dynamic Frames [27] as the underlying logical framework, which is equivalent to separation logic [39] but handles ownership slightly differently. The details of this are deferred to [6,21].

```
1  global seq⟨seq⟨Msg⟩⟩ C; // implementation of communication channels
2  global int N; // total number of workers
3  global int L; // rank of the leader to be announced
4
5  lock_invariant L ↪¹proc − ∗ ∃c : seq⟨seq⟨Msg⟩⟩ . C ↪¹proc c ∗ N ↪½proc |c|;
6
7  given p, P, Q, Π, π, π′;
8  context {chan ↦ C} ∈ Π ∗ ∃n . N ↪π proc n ∗ 0 ≤ rank < n;
9  requires Proc_π′(X, p, send(rank, msg) · P + Q, Π);
10 ensures Proc_π′(X, p, P, Π);
11 void mp_send(ref X, int rank, Msg msg) { /∗ omitted ∗/ }
12
13 given p, P, Q, Π, π, π′;
14 context {chan ↦ C} ∈ Π ∗ ∃n . N ↪π proc n ∗ 0 ≤ rank < n;
15 requires Proc_π′(X, p, Σ_{m∈Msg} recv(rank, m) · P + Q, Π);
16 ensures Proc_π′(X, p, P[m/\result], Π);
17 Msg mp_recv(ref X, int rank) { /∗ omitted ∗/ }
18
19 given n, p, Π, π, π′;
20 context {lead ↦ L, chan ↦ C} ∈ Π ∗ N ↪π proc n ∗ 0 ≤ rank < n;
21 requires Proc_π′(X, p, Elect(rank, v0, v, n), Π);
22 ensures Proc_π′(X, p, ε, Π);
23 void elect(ref X, int rank, Msg v0, Msg v) {
24   loop_invariant 0 ≤ i ≤ n;
25   loop_invariant Proc_π′(X, p, Elect(rank, v0, v, n − i), Π);
26   for (int i := 0 to N) {
27     mp_send(X, (rank + 1) % N, v) with {
28       P := Σ_{x∈Msg} recv(rank, x) · Elect(rank, v0, max(v, x), n − i − 1),
29       Q := ε, p := p, Π := Π, π := π, π′ := π′
30     };
31     Msg v′ := mp_recv(X, rank) with {
32       P := Elect(rank, v0, max(v, v′), n − i − 1),
33       Q := ε, p := p, Π := Π, π := π, π′ := π′
34     };
35     v := max(v, v′);
36   }
37   if (v = v0) {
38     atomic { action X.announce(rank) do L := rank; }
39   }
40 }
```

Fig. 7: The annotated implementation of the leader election protocol. Annotations of the form **context** $\mathcal{P}$ are shorthand for **requires** $\mathcal{P}$; **ensures** $\mathcal{P}$.

lines 27 and 31 are annotated with **with** clauses that resolve the assignments required by the **given** clauses in the contracts of `mp_send` and `mp_recv`. The **given** $\overline{\eta}$ annotation expresses that the parameter list $\overline{\eta}$ are extra ghost arguments for the sake of specification. After N rounds all workers with $v = v_0$ announce

```
41 given p, Π;
42 context N ↪[1/2]→_proc |vs| ∗ 0 < |vs|;
43 requires Proc_1(X, p, ParElect(vs), Π);
44 ensures Proc_1(X, p, ε, Π);
45 void parelect(ref X, seq⟨Msg⟩ vs) {
46   context 0 ≤ rank < |vs|;
47   requires Proc_{1/|vs|}(X, p, Elect(rank, vs[rank], vs[rank], |vs|), Π′);
48   ensures Proc_{1/|vs|}(X, p, ε, Π);
49   par (int rank := 0 to N) {
50     elect(X, vs[rank], vs[rank]) with {
51       n := N, p := p, Π := Π, π := 1/(4|vs|), π′ := 1/|vs|
52     };
53   }
54 }
55
56 context N ↪[1]_std − ∗ C ↪[1]_std − ∗ L ↪[1]_std −;
57 requires ∀i, j : int . (0 ≤ i < |vs| ∧ 0 ≤ j < |vs| ∧ vs[i] = vs[j]) ⇒ i = j;
58 ensures 0 ≤ \result < |vs|;
59 ensures ∀i : int . (0 ≤ i < |vs|) ⇒ vs[i] ≤ vs[\result];
60 int main(seq⟨Msg⟩ vs) {
61   N := |vs|, C := initialiseChannels(N);
62   X := process ParElect(vs) over {chan ↦ C, lead ↦ L};
63   commitLock(); // initialise the lock invariant
64   parelect(X, vs) with {p := ParElect(vs), Π := {chan ↦ C, lead ↦ L}};
65   uncommitLock(); // reclaim the lock invariant
66   finish X; // obtain the global correctness property from the abstraction
67   return L; // return rank of leader
68 }
```

Fig. 8: Bootstrap procedures of the leader election protocol.

themselves as leader. However, since the initial values are chosen to be unique there can only be one such worker. Finally, we can verify that at the post-state of `elect` the abstract model has been fully executed and thus reduced to ε.

The `mp_send`(X, *rank*, *msg*) method implements the operation of enqueuing *msg* onto the message queue of worker *rank*. Its implementation has been omitted for brevity. The contract of `mp_send` expresses that the enqueuing operation is encapsulated as a send(*rank*, *msg*) action that is prescribed by an abstract model identified by X. The `mp_recv`(X, *rank*) function implements the operation of dequeuing and returns the first message of the message queue of worker *rank*. The receive is prescribed as the recv action on the abstraction level, where the potential received message is ranged over by the summation on line 15.

Figure 8 presents bootstrapping code for the implementation of message passing. The `main` function initialises the communication channels whereas `parelect` spawns all worker threads. `main`(vs) additionally initialises and finalises the abstraction ParElect(vs) on the specification level (on line 62 and 66, respectively), whose analysis allows one to establish the postconditions of `main`. The function `parelect`(X, vs) implements the abstract model ParElect(vs) by

spawning N workers that all execute the `elect` program. The contract associated to the parallel block (lines 46–48) is called an *iteration contract* and assigns pre- and postconditions to every parallel instance. For more details on iteration contracts we refer to [5]. Most importantly, the iteration contract of each parallel worker states (on line 47) that the worker behaves as specified by Elect. Thus, we deductively verify in a *thread-modular way* that the program implements its behavioural specification. Lastly, all the required ownership for the global fields and the Proc_1 predicate is split and distributed among the individual workers via the iteration contract and the **with** clause on lines 50–52.

6 Related Work

Significant progress has been made on the theory of concurrent software verification over the last years [13,11,48,49,50,30,12,41]. This line of research proposes advanced program logics that all provide some notion of expressing and restricting thread interference of various complexity, via *protocols* [24]: formal descriptions of how shared-memory is allowed to evolve over time. In our approach protocols have the form of process algebraic abstractions.

The original work on CSL [32] allows specifying simple thread interference in shared-memory programs via resource invariants and critical regions. Later, RGSep [54] merges CSL with rely-guarantee reasoning to enable describing more fine-grained inter-thread interference by identifying atomic concurrent actions. Many modern program logics build on these principles and propose even more advanced ways of verifying shared-memory concurrency. For example, TaDa [41] and CaReSL [50] express thread interference protocols through state-transition systems. iCAP [48] and Iris [25] propose a more unified approach by accepting user-defined monoids to express protocols on shared state, together with invariants restricting these protocols. Iris provides reasoning support for proving language properties in Coq, where our focus is on proving programs correct.

In the distributed setting, Disel [44] allows specifying protocols for distributed systems. Disel builds on dependent type theory and is implemented as a shallow embedding in Coq. Even though their approach is more expressive than ours, it can only semi-automatically be applied in the context of Coq. Villard et al. [55] present a program logic for message passing concurrency, where threads may communicate over channels using native send/receive primitives. This program logic allows specifying protocols via *contracts*, which are state-machines in the style of Session Types [18], to describe channel behaviour. Our technique is more general, as the approach of Villard et al. is tailored specifically to basic shared-memory message passing. Actor Services [47] is a program logic with assertions to express the consequences of asynchronous message transfers between actors. However, the meta-theory of Actor Services has not been proven sound.

Most of the related work given so far is essentially theoretical and mainly focuses on expressiveness and generality. Our approach is a trade-off between expressivity and usability. It allows specifying process algebraic protocols over a general class of concurrent systems, while also allowing the approach to be

implemented in automated verifiers for concurrency like VerCors. Related concurrency verifiers are SmallfootRG [10], VeriFast [19], CIVL [45], Threader [14] and Viper [22,29]; the latter tool is used as the main back-end of VerCors. SmallfootRG is a memory-safety verifier based on RGSep. VeriFast is a rich toolset for verifying (multi-threaded) Java and C programs using separation logic. Notably, Penninckx et al. [40] extend VeriFast with a Petri-net extension to reason about the I/O behaviour of programs. This Petri-net approach is similar to ours, however our technique supports reasoning about abstract models and allows reasoning about more than just I/O behaviour. The CIVL framework can reason about race-freedom and functional correctness of MPI programs written in C [57,28]. The reasoning is done via bounded model checking combined with symbolic execution. Threader is an automated verifier for multi-threaded C, based on model checking and counterexample-guided abstraction refinement.

Apart from the proposed technique, VerCors also allows using process algebraic abstractions as *histories* [7,56]. Also related in this respect are the time-stamped histories of [43], which records atomic state changes in concurrent programs as a history, which are, likewise to our approach, handled as resources in the logic. However, history recording is only suitable for terminating programs.

Finally, there is a lot of more general work on proving linearisability [15,52,26], which essentially allows reasoning about fine-grained concurrency by using sequential verification techniques. Our technique, as well as the history-based technique of [7], uses process algebraic linearisation to do so.

7 Conclusion

To reason effectively about realistic concurrent and distributed software, we have presented a verification technique that performs the reasoning at a suitable level of abstraction that hides irrelevant implementation details, is scalable to realistic programs by being modular and compositional, and is practical by being supported by automated tools. The approach is expressive enough to allow reasoning about realistic software as is demonstrated by the case study as well as by [36], and can be implemented as part of an automated deductive program verifier (viz. VerCors). The proof system underlying our technique has mechanically been proven sound using Coq. Our technique is therefore supported by a strong combination of theoretical justification and practical usability.

We consider the presented technique as just the beginning of a comprehensive verification framework that aims to capture many different concurrent and distributed programming paradigms. To illustrate, we recently adapted the presented approach to the distributed case, by allowing process algebraic models to describe message passing behaviour of distributed programs [37].

We are currently further investigating the use of mCRL2 and Ivy to reason algorithmically about program abstractions, e.g., [31]. Moreover, we are planning to investigate the preservation of liveness properties in addition to safety.

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. A. Aldini, M. Bernardo, and F. Corradini. *A Process Algebraic Approach to Software Architecture Design*. Springer Science & Business Media, 2010. doi:10.1007/978-1-84800-223-4.
2. A. Appel and S. Blazy. Separation Logic for Small-Step CMINOR. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 5–21. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-74591-4_3.
3. A. Appel, P. Melliès, C. Richards, and J. Vouillon. A Very Modal Model of a Modern, Major, General Type System. In *POPL*, volume 42, pages 109–122. ACM, 2007. doi:10.1145/1190216.1190235.
4. J. Baeten. *Process Algebra with Explicit Termination*. Eindhoven University of Technology, Department of Mathematics and Computing Science, 2000.
5. S. Blom, S. Darabi, and M. Huisman. Verification of Loop Parallelisations. In A. Egyed and I. Schaefer, editors, *FASE*, volume 9033 of *LNCS*, pages 202–217. Springer, 2015. doi:10.1007/978-3-662-46675-9_14.
6. S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In N. Polikarpova and S. Schneider, editors, *iFM*, volume 10510 of *LNCS*, pages 102–110. Springer, 2017. doi:10.1007/978-3-319-66845-1_7.
7. S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. History-Based Verification of Functional Behaviour of Concurrent Programs. In R. Calinescu and B. Rumpe, editors, *SEFM*, volume 9276 of *LNCS*, pages 84–98. Springer, 2015. doi:10.1007/978-3-319-22969-0_6.
8. J. Boyland. Checking Interference with Fractional Permissions. In R. Cousot, editor, *Static Analysis (SAS)*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003. doi:10.1007/3-540-44898-5_4.
9. S. Brookes. A Semantics for Concurrent Separation Logic. *Theoretical Computer Science*, 375(1–3):227–270, 2007. doi:10.1016/j.tcs.2006.12.034.
10. C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular Safety Checking for Fine-Grained Concurrency. In H. Nielson and G. Filé, editors, *Static Analysis (SAS)*, pages 233–248. Springer, 2007. doi:10.1007/978-3-540-74061-2_15.
11. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *LNCS*, pages 504–528, 2010. doi:10.1007/978-3-642-14107-2_24.
12. X. Feng. Local Rely-Guarantee Reasoning. In *POPL*, volume 44, pages 315–327. ACM, 2009. doi:10.1145/1480881.1480922.
13. X. Feng, R. Ferreira, and Z. Shao. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In R. De Nicola, editor, *ESOP*, pages 173–188. Springer, 2007. doi:10.1007/978-3-540-71316-6_13.
14. A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A Constraint-Based Verifier for Multi-threaded Programs. In G. Gopalakrishnan and S. Qadeer, editors, *CAV*, pages 412–417. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-22110-1_32.
15. M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *TOPLAS*, 12(3):463–492, 1990. doi:10.1145/78969.78972.
16. A. Hobor. *Oracle Semantics*. PhD thesis, Princeton University, 2008.
17. A. Hobor, A. Appel, and F. Nardelli. Oracle Semantics for Concurrent Separation Logic. In S. Drossopoulou, editor, *ESOP*, pages 353–367. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-78739-6_27.

18. K. Honda, V. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In C. Hankin, editor, *ESOP*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.
19. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NFM*, pages 41–55. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-20398-5_4.
20. C. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *TOPLAS*, 5(4):596–619, 1983. doi:10.1145/69575.69577.
21. S. Joosten, W. Oortwijn, M. Safari, and M. Huisman. An Exercise in Verifying Sequential Programs with VerCors. In A. Summers, editor, *FTfJP*, pages 40–45. ACM, 2018. doi:10.1145/3236454.3236479.
22. U. Juhasz, I. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. Technical report, ETH Zürich, 2014.
23. R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-Order Ghost State. In *ICFP*, volume 51, pages 256–269. ACM, 2016. doi:10.1145/2951913.2951943.
24. R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*, volume 50, pages 637–650. ACM, 2015. doi:10.1145/2676726.2676980.
25. R. Krebbers, R. Jung, A. Bizjak, J. Jourdan, D. Dreyer, and L. Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In H. Yang, editor, *ESOP*, volume 10201 of *LNCS*, pages 696–723. Springer, 2017. doi:10.1007/978-3-662-54434-1_26.
26. S. Krishna, D. Shasha, and T. Wies. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *POPL*, 2:1–31, 2017. doi:10.1145/3158125.
27. K.R.M. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *FOSAD*, volume 5705 of *LNCS*, pages 195–222, 2009. doi:10.1007/978-3-642-03829-7_7.
28. Z. Luo, M. Zheng, and S. Siegel. Verification of MPI programs using CIVL. In *EuroMPI*. ACM, 2017. doi:10.1145/3127024.3127032.
29. P. Müller, M. Schwerhoff, and A. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K.R.M. Leino, editors, *VMCAI*, pages 41–62. Springer, 2016. doi:10.1007/978-3-662-49122-5_2.
30. A. Nanevski, R. Ley-Wild, I. Sergey, and G. Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In Z. Shao, editor, *ESOP*, pages 290–310. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54833-8_16.
31. T. Neele, T. Willemse, and J.F. Groote. Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting. In K. Bae and P. Ölveczky, editors, *FACS*, pages 216–236. Springer, 2018. doi:10.1007/978-3-030-02146-7_11.
32. P. O'Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007. doi:10.1016/j.tcs.2006.12.035.
33. P. O'Hearn, H. Yang, and J. Reynolds. Separation and Information Hiding. In *POPL*, volume 39, pages 268–280. ACM, 2004. doi:10.1145/964001.964024.
34. W. Oortwijn, S. Blom, D. Gurov, M. Huisman, and M. Zaharieva-Stojanovski. An Abstraction Technique for Describing Concurrent Program Behaviour. In A. Paske-

vich and T. Wies, editors, *VSTTE*, volume 10712 of *LNCS*, pages 191–209, 2017. doi:10.1007/978-3-319-72308-2_12.

35. W. Oortwijn, S. Blom, and M. Huisman. Future-based Static Analysis of Message Passing Programs. In *Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES)*, pages 65–72. Open Publishing Association, 2016. doi:10.4204/EPTCS.211.7.
36. W. Oortwijn and M. Huisman. Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System. In W. Ahrendt and S. L. Tapia Tarifa, editors, *iFM*, LNCS. Springer, 2019. To appear.
37. W. Oortwijn and M. Huisman. Practical Abstractions for Automated Verification of Message Passing Concurrency. In W. Ahrendt and S. L. Tapia Tarifa, editors, *iFM*, LNCS. Springer, 2019. To appear.
38. S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica*, 6:319–340, 1975. doi:10.1007/BF00268134.
39. M. Parkinson and A. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. In G. Barthe, editor, *ESOP*, pages 439–458. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-19718-5_23.
40. W. Penninckx, B. Jacobs, and F. Piessens. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In J. Vitek, editor, *ESOP*, pages 158–182. Springer Berlin Heidelberg, 2015. doi:10.1007/978-3-662-46669-8_7.
41. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In R. Jones, editor, *ECOOP*, LNCS, pages 207–231. Springer, 2014. doi:10.1007/978-3-662-44202-9_9.
42. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Steps in Modular Specifications for Concurrent Modules. In *MFPS*, pages 3–18, 2015. doi:10.1016/j.entcs.2015.12.002.
43. I. Sergey, A. Nanevski, and A. Banerjee. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In J. Vitek, editor, *ESOP*, volume 9032 of *LNCS*, pages 333–358. Springer, 2015. doi:10.1007/978-3-662-46669-8_14.
44. I. Sergey, J. Wilcox, and Z. Tatlock. Programming and Proving with Distributed Protocols. *POPL*, 2:1–30, 2017. doi:10.1145/3158116.
45. S. Siegel, M. Zheng, Z. Luo, T. Zirkel, A. Marianiello, J. Edenhofner, M. Dwyer, and M. Rogers. CIVL: The Concurrency Intermediate Verification Language. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 61. ACM, 2015. doi:10.1145/2807591.2807635.
46. Supplementary Material. The supplementary material for this paper, consisting of a technical report, the Coq formalisation and the case study, can be found online at https://github.com/wytseoortwijn/VMCAI20-SharedMemAbstr.
47. A. Summers and P. Müller. Actor Services – Modular Verification of Message Passing Programs. In P. Thiemann, editor, *ESOP*, pages 699–726. Springer, 2016. doi:10.1007/978-3-662-49498-1_27.
48. K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In Z. Shao, editor, *ESOP*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014. doi:10.1007/978-3-642-54833-8_9.
49. K. Svendsen, L. Birkedal, and M. Parkinson. Modular Reasoning about Separation of Concurrent Data Structures. In M. Felleisen and P. Gardner, editors, *ESOP*, pages 169–188. Springer, 2013. doi:10.1007/978-3-642-37036-6_11.

50. A. Turon, D. Dreyer, and L. Birkedal. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-Order Concurrency. In *ICFP*, pages 377–390. ACM, 2013. doi:10.1145/2500365.2500600.
51. Y. Usenko. *Linearization in μCRL*. Technische Universiteit Eindhoven, 2002.
52. V. Vafeiadis. Automatically Proving Linearizability. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, pages 450–464. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-14295-6_40.
53. V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, volume 276 of *ENTCS*, pages 335–351, 2011. doi:10.1016/j.entcs.2011.09.029.
54. V. Vafeiadis and M. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In L. Caires and V. Vasconcelos, editors, *CONCUR*, pages 256–271. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-74407-8_18.
55. J. Villard, É. Lozes, and C. Calcagno. Proving Copyless Message Passing. In Z. Hu, editor, *APLAS*, pages 194–209. Springer, 2009. doi:10.1007/978-3-642-10672-9_15.
56. M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying Functional Behaviour of Concurrent Programs*. PhD thesis, University of Twente, 2015. doi:10.3990/1.9789036539241.
57. M. Zheng, M. Rogers, Z. Luo, M. Dwyer, and S. Siegel. CIVL: Formal Verification of Parallel Programs. In *ASE*, pages 830–835. IEEE, 2015. doi:10.1109/ASE.2015.99.