Practical Abstractions for Automated Verification of Concurrent Program Behaviour
(Technical Report)

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

[1] University of Twente, the Netherlands
[2] KTH Royal Institute of Technology, Sweden

Abstract. Modern concurrent and distributed software is highly complex. Techniques to reason about the correct behaviour of such software are essential to ensure their reliability. To be able to reason about realistic programs, these techniques must be scalable and thus modular and compositional, as well as practical and supported by automated tools. Even though concurrency verification is a very active research field, most work is theoretical and focuses more on expressiveness rather than usability. This paper contributes a powerful and practical technique for verifying behavioural properties of concurrent and distributed programs that makes a trade-off between expressivity and usability. The uniqueness of the approach is that program behaviour is abstractly described using process algebra. Reasoning about complex concurrent program behaviours is only practical if conducted at a suitable level of abstraction that hides irrelevant implementation details. We believe that process algebra provides a language for modelling and reasoning about the behaviour of concurrent programs exactly at the right abstraction level. 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 technique is formally justified by a number of correctness results that have mechanically been proven using Coq. In addition, our technique is supported by the VerCors toolset for concurrency verification and demonstrated on a case study, covering the verification of a leader election protocol.

1 Introduction

Modern software systems are typically composed of concurrent components that communicate via shared or distributed interfaces. Such software systems are behaviourally highly complex due to the many possible concurrent interactions between their components, and therefore notoriously difficult to develop correctly. Verification techniques are needed to aid software developers to comprehend all possible system behaviours, with the goal to ensure system reliability. To be able to reason about *realistic* programs, these techniques should be scalable and thus *modular* and *compositional*, and should also be *practical* and supported by *automated tools*.

Even though verification of concurrent and distributed software is a very active research field [19,16,56,40,12,51], most work is theoretical and mainly focuses on expressivity rather than usability. Instead, this paper contributes a scalable and practical technique for verifying behavioural properties of concurrent and distributed programs that makes a trade-off between expressivity and usability. Thus, 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 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. It is known that *process algebra* provides a language for modelling and reasoning about the behaviour of concurrent programs at the right abstraction level [1]. Process algebra offers an abstract, mathematically elegant way of expressing program behaviour. In contrast, the behaviour of a *real* concurrent programming language with shared memory, threads and locks, has far less algebraic behaviour. This makes process algebra a suitable 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, which can be seen as even more abstract behavioural specifications). The main difficulty is presented by 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 uniqueness 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. In particular, this includes a machine-checked soundness proof of the proof system, stating that any verified program adheres to its behavioural specification—its abstract model.

The verification technique has been been implemented in the VerCors toolset for automated deductive verification of parallel and concurrent software [4]. Even though some of the program logics in the research line given earlier are supported by Coq embeddings, their use requires substantial user interaction. In contrast, automated deductive verifiers make it much easier for users to construct a proof and only require a few manual annotations. Finally, the approach has been applied on various case studies [43], including a leader election protocol that is included in this paper.

Contributions. We make the following main contributions:

1. A technique to reason about the behaviour of shared-memory programs by means of abstraction. The proposed technique is *scalable* and the underlying

proof system is proven *sound*. Moreover, the technique is also *practical* and is implemented in the VerCors toolset for concurrency verification.
2. *Theoretical justification* of the proposed verification approach, presented as a program logic that extends CSL [59]. This paper complements our earlier article [43] that essentially contributes tool support and gives a more practical overview of the verification approach.
3. A *Coq formalisation* of the theory, containing machine-checked proofs of all theorems presented in this paper, including soundness of the proof system.
4. A *detailed case study* covering the verification of a classical distributed algorithm: a leader election protocol. This case study demonstrates that the approach can also be applied to reason about distributed software.

Outline. The remainder of this paper is organised as follows. First, Section 2 illustrates our technique on a small example, and gives a roadmap of the formalisation. Then, Sections 3–6 cover the formalisation of our technique, which is a concurrent separation logic with machinery for handling process-algebraic programs abstractions. In particular, Section 3 defines the syntax and semantics of the process algebra language for the program abstractions, as well as the syntax and semantics of programs themselves. Section 4 defines the assertion language and its semantics. Section 5 discusses the proof rules of the program logic and Section 6 discusses their soundness. Then, Section 7 elaborates on the implementation of the verification approach in VerCors and on the Coq development of the meta-theory. Section 8 demonstrates the approach on a larger case study: the verification of a leader election protocol. Finally, Section 9 discusses related work and Section 10 concludes and gives directions for future work.

2 Approach

Let us first sketch the verification approach by illustrating it on a small verification example. Our approach allows to abstractly specify concurrent program behaviour as *process algebra terms*. Process algebra terms are composed of *atomic, indivisible actions*. The new idea is that actions are *logical descriptions of shared-memory operations*: 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 programs and abstract models are established deductively, using a concurrent separation logic that is presented later in the paper. In addition, well-known techniques for process-algebraic reasoning can be applied to *guarantee* safety properties over all possible state changes, as described by their compositions of atomic actions. The novelty of the approach is that these safety properties can then be *relied upon* in the program logic due to the established connection between the program and its abstract model.

Example program. To illustrate the approach, let us consider the following program, which is a simple variant of the classical Owicki-Gries example [45].

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

This program consists of two concurrent threads: one that atomically increments the value at the shared memory location E by four, while the other thread atomically multiplies the value of E by four. The challenge is to modularly deduce that after termination of both threads, the value at heap location E is either $(E_{old}+4)*4$ or $(E_{old}*4)+4$, with E_{old} the old value at E at the pre-state of computation. Well-known classical approaches to deal with such concurrent modules [14] include *auxiliary state* [45] and *interference abstraction* via rely-guarantee reasoning [27]. Modern program logics employ first-order constructs, e.g. *atomic Hoare triples* [13] in the context of TaDa, or higher-order techniques, such as *higher-order ghost state* [30] in the context of Iris. However, these classical approaches typically do not scale well, whereas these modern approaches are hard to integrate into (semi-)automated verification tools.

***Step 1*: Program abstraction.** Two atomic actions may be defined for this example, incr and mult, that abstract the increment and multiplication in the program. These actions logically describe the behaviour of the atomic program instructions by associating *action contracts* to them, in the following way.

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

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

The action contracts abstractly describe the atomic state changes made by the two threads. The variable x is a free, process-algebraic variable that is later linked to a concrete heap location in the program; E in our case. Moreover, we generalised the increment and multiplication of 4 to an arbitrary integer n. The incr and mult actions may be composed using standard connectives from process algebra to construct an *abstract model* of the Owicki-Gries program in the following way.

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

***Step 2*: Reasoning about abstract models.** By analysing contract-complying action sequences at the process-algebraic level, we may indirectly reason about how the contents at heap location E evolves over time on the program level. The process p defined above has an associated contract (that is specified manually), stating that all finite traces of incr(n) $\parallel$ mult(n) satisfy the Owicki-Gries post-condition. This can automatically be verified: the process can be linearised to the bisimilar process term incr(n)$\cdot$mult(n) + mult(n)$\cdot$incr(n) and subsequently analysed (e.g. algorithmically, deductively, or via weakest-precondition reasoning) to establish its postcondition. VerCors does the analysis by encoding the linearised process term as input to the Viper verifier [39]. Alternatively, the analysis might also be done via a translation to e.g. mCRL2 [21] or the Ivy verifier [46].

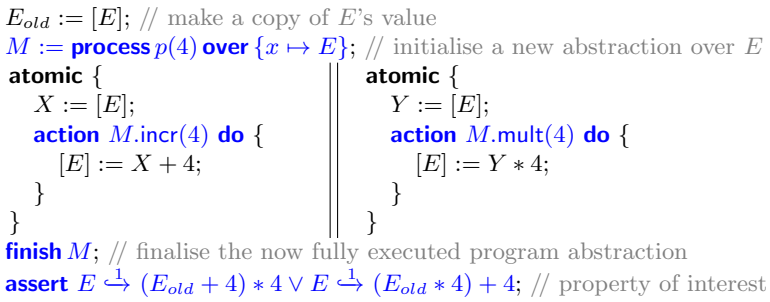

Fig. 1: Annotated version of the Owicki-Gries example.

***Step 3*: Deductively linking abstract models to programs.** The next step is to project this process-algebraic reasoning onto the program behaviour, by connecting x to E and linking incr and mult to the concrete corresponding instructions in the program. The former is done by *initialising* a new abstract model in the program logic that executes according to $p(4)$, thereby mapping x to heap location E. The latter is done by identifying *action blocks* in the code, using special program annotations. Figure 1 presents the annotated program.

The ghost instruction $M :=$ **process** $p(4)$ **over** $\{x \mapsto E\}$ initialises a new program abstraction $p(4)$ that protects all changes to the heap location E, whose content is abstracted as the process-algebraic variable x (recall that x occurs free in the contracts of p and its actions). The annotations **action** $M.a$ **do** C indicate that, by executing the program C, the action a is performed on the abstraction level. These **action** annotations are thus used to *synchronise* concrete program behaviour with the process-algebraic abstract descriptions of the program behaviour. Our proof system allows to deductively prove that the parallel program executes as prescribed by the process $p(4)$, which is defined to be incr(4) $\|$ mult(4). In particular, we verify in a *thread-modular way* that the left thread performs the incr(4) action and that the right thread performs mult(4). Moreover, the proof system only permits writing to E in the context of **action** blocks and thereby enforces that the parallel program *must* adhere to $p(4)$. After showing that the parallel program fully adheres to $p(4)$, the program logic allows to *finalise* the model M using the **finish** M ghost instruction. Finalisation in this sense means projecting the process-algebraic reasoning onto concrete program behaviour. In this case, we project the postcondition of $p(4)$ onto the heap location E, as shown by the last assertion. Here, $\hookrightarrow^{\pi}$ is the standard points-to assertion of intuitionistic separation logic, with a fractional permission π [7].

Other verification examples. More intricate verification examples can be found in our previous work [43], which gives a more practical overview on how to use program abstractions using VerCors. This work also contains a verified implementation of the Owicki-Gries example. Moreover, Section 8 contains a de-

tailed case study in which the correctness of a leader election protocol is verified using program abstractions.

2.1 Formalisation roadmap

Since the formalisation presented in Sections 3–6 may be quite involved, let us first sketch a roadmap of the formalisation and the soundness statement.

Our verification approach extends classical CSL of [41,8] with proof rules and logical machinery for the **process**, **action** and **finish** ghost instructions. Section 5 defines program judgements as sequents of the form $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$, where C is a program; $\mathcal{P}$, $\mathcal{Q}$ and $\mathcal{R}$ are assertions where $\mathcal{R}$ is the resource invariant; and Γ are *interface specifications* [42] consisting of the contracts of all process-algebraic abstractions defined for C (e.g. the pre- and postcondition of the process p). Soundness of the program logic means the following: if *(1)* a proof $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ can be derived for a program C, and *(2)* if all abstract models defined for C satisfy their contract (denoted as $\models_{\mathsf{env}} \Gamma$), then C is *safe for any number of computation steps*, starting from any state satisfying $\mathcal{P}$. Premise **(1)** of this soundness property is established via deductive reasoning, while premise **(2)** can be established algorithmically, e.g. via model checking. Computation steps—the operational semantics of programs and program abstractions—are defined in Section 3, while assertions and their models are defined in Section 4.

Our notion of "*safety for n computation steps*" (often referred to as *adequacy*, see Definition 17) is largely based on the well-known notion of fault-avoiding configuration safety from [59] but is extended with machinery to handle program abstractions. Most importantly, our definition of safety establishes a *simulation relation* between program execution and the execution of all active abstractions, for n computation steps. Such a simulation relation allows to prove that the action sequences of a verified program C, as induced by the executions of C, are a subset of the traces of the corresponding process algebraic abstractions.

To help establish this simulation relation a second operational semantics is defined in Section 6.1, referred to as the *ghost operational semantics*, that executes the program in *lock-step* with all active abstractions. The ghost semantics administers all active abstractions in *process maps*, which are defined in Section 4.1. The simulation relation sought is then a *backward simulation* between the standard operational semantics of programs and the ghost operational semantics. This backward simulation, together with **(2)** and some extra machinery defined in Section 6.2 allows us to obtain and use the postconditions of fully reduced program abstractions after finalisation using **finish**. Section 6.3 formally defines adequacy and presents the soundness statement.

3 Programs and Abstractions

This section introduces the programming language that we use to formalise the verification technique. Firstly, Section 3.1 defines the syntax and semantics of process-algebraic abstractions. Secondly, Section 3.2 discusses the syntax

Process algebra language

$$
\begin{array}{rrcl}
(Act)\ a, b, \ldots & & (ProcVar)\quad x, y, z, \ldots & (Lit)\quad m, n, \ldots \\
(ProcExpr) & e & ::= & m \mid x \mid e + e \mid e - e \mid \cdots \\
(ProcCond) & b & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg b \mid b \wedge b \mid e = e \mid e < e \mid \cdots \\
(Proc) & P, Q & ::= & \varepsilon \mid \delta \mid a \mid P \cdot Q \mid P + Q \mid P \parallel Q \mid P \between Q \mid P^*
\end{array}
$$

Fig. 2: The abstract syntax of the process algebra language.

and semantics of programs. Notably, the programming language contains the specification-only ghost commands **process**, **action** and **finish** that allow to initialise, update, and finalise abstract models inside the program logic.

3.1 Process-algebraic abstractions

Figure 2 presents the statics of the process algebra language that is used to define program abstractions. As usual, ε is the *empty process*; it has no behaviour and terminates successfully. The δ process is the *deadlocked process*; it neither progresses nor terminates. Sequential composition of process terms P and Q is written $P \cdot Q$, while $P + Q$ denotes non-deterministic choice. The process $P \parallel Q$ is the parallel composition of processes P and Q. The process $P \between Q$ is the *left-merge* of P and Q, which is similar in spirit to parallel composition, however $\between$ insists that the left-most process P proceeds first. Left-merge is an auxiliary connective commonly used to axiomatise parallel composition [38] by having $P \parallel Q = P \between Q + Q \between P$. Finally, the process P^* is the *(unary) Kleene iteration* of P: it denotes a sequence of zero or more P's.

Our verification approach uses process algebra terms in the presence of *data*, implemented via *action contracts*. Action contracts consist of pre- and postconditions that logically describe the state changes imposed by the action. These action contracts make the process-algebra language non-standard. We assume that each action has an associated contract assigned to it, and use the functions $\mathsf{pre}, \mathsf{post} : Act \to ProcCond$ for obtaining the pre- and postcondition of an action, respectively. Both these conditions are of type $ProcCond$, which is the domain of Boolean expressions over process-algebraic variables.

Operational semantics. The operational semantics of the process algebra language is expressed as a binary reduction relation $\xrightarrow{a}$ over *process configurations* (P, σ), labelled with action labels a. Figure 3 presents the reduction rules. The notion of data is implemented via *process states* $\sigma \in ProcStore \triangleq ProcVar \to Val$ that map process variables to a semantic domain Val of values.

Most reduction rules are standard in spirit [20]. The non-standard ABSTR-ACT rule for action handling permits state to change in any way that complies with the action contract. Moreover, an explicit notion of successful termination is used, which is common in process algebras with ε [2] to properly define the transition rules for sequential composition; ABSTR-SEQ-R in particular. The process

Successful termination $\boxed{P\downarrow}$

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

Small-step reduction rules $\boxed{P,\sigma \xrightarrow{a} P',\sigma'}$

ABSTR-ACT
$$\frac{[\![\mathsf{pre}(a)]\!](\sigma) \qquad [\![\mathsf{post}(a)]\!](\sigma')}{a,\sigma \xrightarrow{a} \varepsilon,\sigma'}$$

ABSTR-SEQ-L
$$\frac{P,\sigma \xrightarrow{a} P',\sigma'}{P\cdot Q,\sigma \xrightarrow{a} P'\cdot Q,\sigma'}$$

ABSTR-SEQ-R
$$\frac{P\downarrow \qquad Q,\sigma \xrightarrow{a} Q',\sigma'}{P\cdot Q,\sigma \xrightarrow{a} Q',\sigma'}$$

ABSTR-ALT-L
$$\frac{P,\sigma \xrightarrow{a} P',\sigma'}{P+Q,\sigma \xrightarrow{a} P',\sigma'}$$

ABSTR-ALT-R
$$\frac{Q,\sigma \xrightarrow{a} Q',\sigma'}{P+Q,\sigma \xrightarrow{a} Q',\sigma'}$$

ABSTR-PAR-L
$$\frac{P,\sigma \xrightarrow{a} P',\sigma'}{P \parallel Q,\sigma \xrightarrow{a} P' \parallel Q,\sigma'}$$

ABSTR-PAR-R
$$\frac{Q,\sigma \xrightarrow{a} Q',\sigma'}{P \parallel Q,\sigma \xrightarrow{a} P \parallel Q',\sigma'}$$

ABSTR-LMERGE
$$\frac{P,\sigma \xrightarrow{a} P',\sigma'}{P \between Q,\sigma \xrightarrow{a} P' \parallel Q,\sigma'}$$

ABSTR-ITER
$$\frac{P,\sigma \xrightarrow{a} P',\sigma'}{P^*,\sigma \xrightarrow{a} P'\cdot P^*,\sigma'}$$

Fig. 3: Small-step operational semantics of process algebra terms.

ε always successfully terminates, while δ never terminates. Iteration P^* may terminate successfully as it may choose not to start iterating and thereby behave as the empty process ε.

The program logic allows to handle process-algebraic abstractions *up to bisimulation* and provides constructs to replace abstractions by bisimilar ones.

Definition 1 (Bisimulation). *A binary relation $\mathscr{R} \subseteq Proc \times Proc$ over process terms is a* bisimulation relation *if, whenever $P\,\mathscr{R}\,Q$, then:*

(1) $P\downarrow$ if and only if $Q\downarrow$.
(2) If $P,\sigma \xrightarrow{a} P',\sigma'$, then there exists a Q' s.t. $Q,\sigma \xrightarrow{a} Q',\sigma'$ and $P'\,\mathscr{R}\,Q'$.
(3) If $Q,\sigma \xrightarrow{a} Q',\sigma'$, then there exists a P' s.t. $P,\sigma \xrightarrow{a} P',\sigma'$ and $P'\,\mathscr{R}\,Q'$.

Bisimulation expresses that two processes exhibit the same behaviour in the sense that their action sequences describe the same state changes. Two processes P, Q are defined to be *bisimilar* or *bisimulation equivalent*, written $P \cong Q$, if and only if there exists a bisimulation relation $\mathscr{R}$ such that $P\,\mathscr{R}\,Q$. Any bisimulation relation constitutes an equivalence relation. As usual, bisimilarity is a congruence for all process algebraic connectives.

Figure 4 presents an axiomatisation of the process algebra language, based on [21] but extended with axioms for ε and iteration, as proposed in [37]. Although standard, the axiom system is presented to give an indication on how process terms may be used in the program logic: program abstractions may be rewritten using these rules. The axioms have been proven sound with respect to bisimulation and are therefore presented directly as $\cong$-equalities.

Sequential connectives

$$P + Q \cong Q + P \quad (\text{A1})$$
$$P + (Q + R) \cong (P + Q) + R \quad (\text{A2})$$
$$P + P \cong P \quad (\text{A3})$$
$$(P + Q) \cdot R \cong P \cdot R + Q \cdot R \quad (\text{A4})$$
$$P \cdot (Q \cdot R) \cong (P \cdot Q) \cdot R \quad (\text{A5})$$
$$P + \delta \cong P \quad (\text{A6})$$
$$\delta \cdot P \cong \delta \quad (\text{A7})$$
$$P \cdot \varepsilon \cong P \quad (\text{A8})$$
$$\varepsilon \cdot P \cong P \quad (\text{A9})$$

Kleene iteration

$$P^* \cong P \cdot P^* + \varepsilon \quad (\text{KS1})$$
$$(P + Q)^* \cong P^* \cdot (Q \cdot P^*)^* \quad (\text{KS2})$$
$$\delta^* \cong \varepsilon \quad (\text{KS3})$$
$$\varepsilon^* \cong \varepsilon \quad (\text{KS4})$$

Parallel connectives

$$P \parallel Q \cong Q \parallel P \quad (\text{M1})$$
$$P \parallel (Q \parallel R) \cong (P \parallel Q) \parallel R \quad (\text{M2})$$
$$P \parallel Q \cong P \between Q + Q \between P \quad (\text{M3})$$
$$\varepsilon \parallel P \cong P \quad (\text{M4})$$
$$P \parallel \delta \cong P \cdot \delta \quad (\text{M5})$$
$$\delta \between P \cong \delta \quad (\text{ML1})$$
$$\varepsilon \between \delta \cong \delta \quad (\text{ML2})$$
$$\varepsilon \between (a \cdot P) \cong \delta \quad (\text{ML3})$$
$$(a \cdot P) \between Q \cong a \cdot (P \parallel Q) \quad (\text{ML4})$$
$$\varepsilon \between \varepsilon \cong \varepsilon \quad (\text{ML5})$$
$$\varepsilon \between (P + Q) \cong \varepsilon \between P + \varepsilon \between Q \quad (\text{ML6})$$
$$(P + Q) \between R \cong P \between R + Q \between R \quad (\text{ML7})$$
$$(P \between Q) \between R \cong P \between (Q \parallel R) \quad (\text{ML8})$$
$$P \between \delta \cong P \cdot \delta \quad (\text{ML9})$$

Fig. 4: Standard equivalences of process algebra terms.

3.2 Programs

We demonstrate our verification approach on a simple concurrent pointer language, whose syntax is presented in Figure 5. This language is a variation of the language proposed by [41,8], extended with *specification-only* commands for handling program abstractions in the logic (displayed in blue). We assume that program abstractions are externally declared, so that commands C are used in contexts **let** $p_0 := P_0, \ldots, p_n := P_n$ **in** C of process declarations $p_i := P_i$, where all $p_i \in \mathit{ProcLabel}$ are process labels. The notation $\mathsf{body}(p)$ is used in the sequel to refer to the process P that is declared in this way at process label p.

As usual, the commands $X := [E]$ and $[E] := E'$ read from and write to the heap at location E, respectively. $X :=$ **alloc** E allocates a free heap location and writes the value E to it. The command **dispose** E deallocates the heap location at E. The program **par** $C_1\, C_2$ is the statically-scoped parallel composition, expressing concurrent execution of C_1 and C_2[3]. The command **atomic** C executes C atomically without interference of other threads, while **inatom** C represents a *partially executed* atomic program—one that is active but still has to execute C. Partially executed programs are an artefact of program execution and are not written by users of the language.

[3] In pseudocode we sometimes write $C_1 \parallel C_2$ instead. The notation **par** $C_1\, C_2$ is used in the formalisation to avoid confusion with parallel composition $P \parallel Q$ of processes.

Programming language

$$
\begin{array}{rrcl}
(\mathit{Var}) & X, Y, \ldots & & \\
(\mathit{Expr}) & E & ::= & n \mid X \mid E + E \mid E - E \mid \cdots \\
(\mathit{Cond}) & B & ::= & \mathsf{True} \mid \mathsf{False} \mid \neg B \mid B \wedge B \mid E = E \mid E < E \mid \cdots \\
(\mathit{AbstrBinder}) & \Pi & ::= & \{x_0 \mapsto E_0, \ldots, x_n \mapsto E_n\} \\
(\mathit{Cmd}) & C & ::= & \mathbf{skip} \mid X := E \mid X := [E] \mid [E] := E \mid C; C \mid X := \mathbf{alloc}\, E \\
 & & & \mid \mathbf{dispose}\, E \mid \mathbf{if}\, B\, \mathbf{then}\, C\, \mathbf{else}\, C \mid \mathbf{while}\, B\, \mathbf{do}\, C \mid \mathbf{par}\, C\, C \\
 & & & \mid \mathbf{atomic}\, C \mid \mathbf{inatom}\, C \mid X := \mathbf{process}\, p\, \mathbf{over}\, \Pi \mid \mathbf{action}\, X.a\, \mathbf{do}\, C \\
 & & & \mid \mathbf{inact}\, C \mid \mathbf{finish}\, X
\end{array}
$$

Fig. 5: Syntax of the program language, with the ghost commands coloured blue.

The commands displayed in blue are specification-only language constructs for handling process-algebraic abstractions, which are ignored during regular program execution. Specification-wise, $X := \mathbf{process}\, p\, \mathbf{over}\, \Pi$ initialises a new abstract model, represented by the process term $\mathsf{body}(p)$ that is declared at p, where Π is a finite mapping from process-algebraic variables to heap locations. Π is used to bind/connect *abstract state*—the state of process-algebraic abstractions—to *concrete state*—heap locations in the program—and is therefore referred to as an *abstraction binder*. The command $\mathbf{finish}\, X$ finalises the abstraction that is identified by X. The $\mathbf{action}\, X.a\, \mathbf{do}\, C$ command executes the program C in the context of a program abstraction identified by X, as the process-algebraic action a. In particular, this command states that, by executing C (according to the operational semantics of the program), the action a is executed in the specified process-algebraic abstraction. And last, the command $\mathbf{inact}\, C$ denotes a *partially executed* action program, one that still has to execute C. Partial action programs are used only internally for technical reasons, and signify that the program C is currently being executed in the context of an action. Likewise to **inatom**, partial action programs can not be written by users.

Definition 2 (User programs). *A program C is a* user program *if C does not contain* $\mathbf{inatom}\, C'$ *nor* $\mathbf{inact}\, C'$ *as a subprogram, for any C'.*

Moreover, the action segment C of programs of the form $\mathbf{action}\ \_\ \mathbf{do}\ C$ and $\mathbf{inact}\, C$ may only contain a subcategory of commands, excluding atomic commands and ghost code, in particular nested action blocks. The latter is needed since actions must be atomically observable by environmental threads. Throughout this paper we will therefore only consider *well-formed* programs.

Definition 3 (Well-formed programs). *A command is defined to be* basic *if it does not contain any atomic subprograms (i.e.* **atomic** *or* **inatom***) or specification-specific language constructs (i.e.* **process**, **action**, **inact**, *or* **finish***).*

A command C is defined to be well-formed*, denoted* $\mathsf{wf}(C)$*, if, for any command* $\mathbf{action}\ \_\ \mathbf{do}\ C'$ *or* $\mathbf{inact}\, C'$ *that occurs in C, the subprogram C' is basic.*

Small-step operational semantics $\boxed{(C, h, s) \rightsquigarrow (C', h', s')}$

$$
\begin{aligned}
(\textbf{skip}; C, h, s) &\rightsquigarrow (C, h, s) \\
(C_1; C_2, h, s) &\rightsquigarrow (C_1'; C_2, h', s') \quad \text{if } (C_1, h, s) \rightsquigarrow (C_1', h', s') \\
(X := E, h, s) &\rightsquigarrow (h, s[X \mapsto [\![E]\!](s)]) \\
(X := [E], h, s) &\rightsquigarrow (\textbf{skip}, h, s[X \mapsto h([\![E]\!](s))]) \\
([E_1] := E_2, h, s) &\rightsquigarrow (\textbf{skip}, h[[\![E_1]\!](s) \mapsto [\![E_2]\!](s)], s) \quad \text{if } [\![E_1]\!](s) \in \mathsf{dom}(h) \\
(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, h, s) &\rightsquigarrow (C_1, h, s) \quad \text{if } [\![B]\!](s) \\
(\textbf{if } B \textbf{ then } C_1 \textbf{ else } C_2, h, s) &\rightsquigarrow (C_2, h, s) \quad \text{if } \neg[\![B]\!](s) \\
(\textbf{while } B \textbf{ do } C, h, s) &\rightsquigarrow (\textbf{if } B \textbf{ then } (C; \textbf{while } B \textbf{ do } C) \textbf{ else skip}, h, s) \\
(X := \textbf{alloc } E, h, s) &\rightsquigarrow (\textbf{skip}, h[\ell \mapsto [\![E]\!](s)], s[X \mapsto \ell]) \quad \text{where } \ell \notin \mathsf{dom}(h) \\
(\textbf{dispose } E, h, s) &\rightsquigarrow (\textbf{skip}, h - [\![E]\!](s), s) \\
(\textbf{atomic } C, h, s) &\rightsquigarrow (\textbf{inatom } C, h, s) \\
(\textbf{inatom } C, h, s) &\rightsquigarrow (\textbf{inatom } C', h', s') \quad \text{if } (C, h, s) \rightsquigarrow (C', h', s') \\
(\textbf{inatom skip}, h, s) &\rightsquigarrow (\textbf{skip}, h, s) \\
(\textbf{par } C_1\ C_2, h, s) &\rightsquigarrow (\textbf{par } C_1'\ C_2, h', s') \quad \text{if } \neg\mathsf{locked}\, C_2 \text{ and } (C_1, h, s) \rightsquigarrow (C_1', h', s') \\
(\textbf{par } C_1\ C_2, h, s) &\rightsquigarrow (\textbf{par } C_1\ C_2', h', s') \quad \text{if } \neg\mathsf{locked}\, C_1 \text{ and } (C_2, h, s) \rightsquigarrow (C_2', h', s') \\
(\textbf{par skip skip}, h, s) &\rightsquigarrow (\textbf{skip}, h, s) \\
(X := \textbf{process } p \textbf{ over } \Pi, h, s) &\rightsquigarrow (\textbf{skip}, h, s) \\
(\textbf{finish } X, h, s) &\rightsquigarrow (\textbf{skip}, h, s) \\
(\textbf{action } X.a \textbf{ do } C, h, s) &\rightsquigarrow (\textbf{inact } C, h, s) \\
(\textbf{inact } C, h, s) &\rightsquigarrow (\textbf{inact } C', h', s') \quad \text{if } (C, h, s) \rightsquigarrow (C', h', s') \\
(\textbf{inact skip}, h, s) &\rightsquigarrow (\textbf{skip}, h, s)
\end{aligned}
$$

Fig. 6: Transition rules of the operational semantics of programs. The predicate $\mathsf{locked}\, C$ determines whether the program C holds the global lock, i.e. whether C executes an atomic (sub)program.

Operational semantics. Figure 6 shows the transition rules of the small-step operational semantics of our language, expressed as a reduction relation $\rightsquigarrow$ between program configurations. A *program configuration* (C, h, s) is a triple consisting of: *(i)* a program C; *(ii)* a heap $h \in \mathit{Heap} \triangleq \mathit{Loc} \rightharpoonup_{\mathsf{fin}} \mathit{Val}$ that models shared memory, with $\mathit{Loc} \subseteq \mathit{Val}$ an infinite semantic domain of heap locations; and *(iii)* a store $s \in \mathit{Store} \triangleq \mathit{Var} \to \mathit{Val}$ that models thread-local memory. A configuration (C, h, s) is defined to be *final* if $C = \textbf{skip}$.

Notably, atomic programs are executed using a small-step reduction strategy. To prevent thread interference while executing atomic programs, the parallel composition only allows a program to make a computation step if the other program does not hold the global lock. A program C is defined to be *(globally) locked* if C executes an atomic program, i.e. if C has $\textbf{inatom}\ \_$ as a subprogram; this is formally defined in terms of the $\mathsf{locked} : \mathit{Cmd} \to \mathit{Prop}$ predicate.

Faulting configurations $\boxed{\lightning(C,h,s)}$

$$
\begin{array}{rl}
\lightning(X := [E], h, s) & \text{if } [\![E]\!](s) \notin \mathsf{dom}(h) \\
\lightning([E_1] := E_2, h, s) & \text{if } [\![E_1]\!](s) \notin \mathsf{dom}(h) \\
\lightning(\textbf{dispose}\, E, h, s) & \text{if } [\![E]\!](s) \notin \mathsf{dom}(h) \\
\lightning(C_1; C_2, h, s) & \text{if } \lightning(C_1, h, s) \\
\lightning(\textbf{par}\, C_1\, C_2, h, s) & \text{if } \lightning(C_1, h, s) \text{ and } \neg\mathsf{locked}\, C_2 \\
\lightning(\textbf{par}\, C_1\, C_2, h, s) & \text{if } \lightning(C_2, h, s) \text{ and } \neg\mathsf{locked}\, C_1 \\
\lightning(\textbf{par}\, C_1\, C_2, h, s) & \text{if } \mathsf{locked}\, C_1 \text{ and } \mathsf{locked}\, C_2 \\
\lightning(\textbf{par}\, C_1\, C_2, h, s) & \text{if } \neg(\mathsf{locked}\, C_1 \vee \mathsf{locked}\, C_2) \text{ and } \mathsf{acc}(C_1, s) \cap \mathsf{mod}(C_2, s) \neq \emptyset \\
\lightning(\textbf{par}\, C_1\, C_2, h, s) & \text{if } \neg(\mathsf{locked}\, C_1 \vee \mathsf{locked}\, C_2) \text{ and } \mathsf{acc}(C_2, s) \cap \mathsf{mod}(C_1, s) \neq \emptyset \\
\lightning(\textbf{inatom}\, C, h, s) & \text{if } \lightning(C, h, s) \\
\lightning(\textbf{inact}\, C, h, s) & \text{if } \lightning(C, h, s)
\end{array}
$$

Fig. 7: The fault semantics of programs.

Definition 4 (Locked programs).

$$
\mathsf{locked}\, C \triangleq \begin{cases}
\mathsf{True} & \textit{if } C = \textbf{inatom}\, C' \\
\mathsf{locked}\, C_1 & \textit{if } C = C_1; C_2 \\
\mathsf{locked}\, C_1 \vee \mathsf{locked}\, C_2 & \textit{if } C = \textbf{par}\, C_1\, C_2 \\
\mathsf{locked}\, C' & \textit{if } C = \textbf{inact}\, C' \\
\mathsf{False} & \textit{otherwise}
\end{cases}
$$

Moreover, process-related commands do not affect the program state and are essentially handled as if they were comments. Notice however that **action** _ **do** C programs are first reduced to **inact** C before C is being executed. This makes it more convenient to later establish a simulation equivalence between the program and ghost operational semantics.

Faulting configurations. Figure 7 presents a *faulting semantics* [49] of programs, which is defined in terms of a relation $\lightning \subseteq \mathit{Cmd} \times \mathit{Heap} \times \mathit{Store}$ over program configurations. The notation $\lightning(C, h, s)$ abbreviates $(C, h, s) \in \lightning$. The operations acc and mod determine the set of heap locations that are *accessed* and *written to* by the given program, respectively, in the next computation step.

Program configurations in $\lightning$ exhibit a fault, as they: *(i)* access unallocated memory, or *(ii)* are deadlocked, or *(iii)* allow to perform a data-race. A data-race is being performed when two threads access the same heap location, where at least one of the two accesses is a write. The soundness argument of the program logic uses *(i)* and *(iii)* to establish memory safety and race freedom, respectively, for any verified program.

To show that the program semantics is properly defined, we prove that the operational semantics is progressive for all non-faulting program configurations.

Theorem 1 (Progress of $\rightsquigarrow$). *For any program configuration $(C, h, s) \notin \lightning$, either $C = \mathbf{skip}$ or there exists a program configuration (C', h', s') such that $(C, h, s) \rightsquigarrow (C', h', s')$.*

4 Program Logic

This section discusses the syntax and interpretation of assertions in the proof system. We first define and discuss the assertion language. Then, Section 4.1 introduces *permission heaps* and *process maps*, which are the composable structures that form the foundation of the models of the program logic. These two structures have many properties in common, and to discuss these properties we express both structures as *separation algebras*. Finally, Section 4.2 defines the semantic interpretation of assertions.

Assertions of the program logic are defined by the following grammar.

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

Apart from the standard CSL connectives, the assertion language contains three different heap ownership predicates $\overset{\pi}{\hookrightarrow}_t$, with t the *heap ownership type*, as well as an ownership predicate Proc_π for program abstractions. As usual, the notation $E \overset{\pi}{\hookrightarrow}_t -$ is used as shorthand for $\exists X. E \overset{\pi}{\hookrightarrow}_t X$, where $X \notin \mathsf{fv}(E)$.

Depending on the ownership type t, the three different points-to assertions express different access rights to the associated heap location. In more detail:

- $E \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E'$ is the *standard heap ownership predicate* from intuitionistic separation logic that gives read-access for $0 < \pi < 1$ and write-access in case $\pi = 1$. Additionally, it expresses that the heap contains the value E' at location E. Moreover, the annotation std indicates that the associated heap location is *not* bound to any abstract model. We say that a heap location $\ell \in \mathit{Loc}$ is *bound by* or *subject to* a program abstraction, if there is an active program abstraction with a binder Π that contains a mapping to ℓ.
- $E \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E'$ is the *process heap ownership predicate*, which indicates that the heap location E is bound to an active abstraction, but in a *read-only* manner. $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ assertions grant read-access exclusively, even in case $\pi = 1$.
- $E \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E'$ is the *action heap ownership predicate*, which indicates that the heap location E is bound by an active program abstraction and is used in the context of an action block in a *read/write* manner.

Action points-to assertions essentially give the same access rights as $\overset{\pi}{\hookrightarrow}_{\mathsf{std}}$ in the proof system. Nevertheless, we need both these predicate types, since a distinction is needed between handling bound and unbound heap locations.

The program logic must for example not allow to deallocate memory that is bound to a program abstraction, as this would be unsound. Moreover, the proof system allows to upgrade $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates to $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ inside action blocks, and $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ again provides write access when $\pi = 1$. Therefore, $E \overset{1}{\hookrightarrow}_{\mathsf{proc}} E'$ predicates grant the *capability to regain write access to* E, inside action blocks. This system of upgrading enforces that all modifications to E happen in the context of $\textbf{action}\, X.a\, \textbf{do}\, C$ commands, so that the modifications are protected and can be recorded by the program abstraction X, as the action a.

Finally, the $\mathsf{Proc}_\pi(X, p, P, \Pi)$ assertion expresses *ownership* of a program abstraction that is identified by X, where the abstraction is represented by the process algebra term P. Ownership in this sense means that the thread has knowledge of the existence of the abstraction, as well as the right to execute as prescribed by the abstraction. The label p identifies the declaration of the process-algebraic abstraction, as specified in the $X := \textbf{process}\, p\, \textbf{over}\, \Pi$ ghost command that was used to initialise the abstract model in the proof system. Furthermore, Π connects the abstract model to the concrete program by mapping the process-algebraic variables in the abstraction to heap locations in the program, as discussed before. And last, the fractional permission π is needed to implement the ownership system of program abstractions. Fractional permissions are only used here to be able to reconstruct the full Proc_1 predicate. We shall later see that Proc_π predicates can be split and merged along π and parallel compositions inside P, and consumed in the proof system by action programs.

4.1 Models of the program logic

Before discussing the semantics of assertions in Section 4.2, this section first introduces *permission heaps* and *process maps*, which form the basis for the models of our concurrent separation logic. Permission heaps extend program heaps to capture the three different types t of heap ownership assertions. Process maps capture state and ownership of process-algebraic abstractions. Both these structures can be conveniently defined in terms of a *separation algebra*, to highlight their common properties.

Definition 5 (Separation algebra). *A* separation algebra *consists of a carrier set* $\mathcal{X}$*, together with:*

- *A distinguished unit element* $\mathbb{1} \in \mathcal{X}$*;*
- *A predicate* $\mathsf{valid} : \mathcal{X} \to \mathsf{Prop}$*;*
- *Three binary relations* $\equiv, \bot, \preceq : \mathcal{X} \to \mathcal{X} \to \mathsf{Prop}$*;*
- *A binary operation* $\uplus : \mathcal{X} \to \mathcal{X} \to \mathcal{X}$*;*

Satisfying the following eight laws:

1. $\equiv$ *is an* equivalence relation *and a* congruence *for* valid*,* $\bot$*,* $\preceq$*, and* $\uplus$*.*
2. $\mathsf{valid}\, \mathbb{1}$ *and* $\mathbb{1} \uplus x \equiv x$*.*
3. $x \uplus (y \uplus z) \equiv (x \uplus y) \uplus z$ *and* $x \uplus y \equiv y \uplus x$*.*
4. *If* $\mathsf{valid}\, x$*, then* $\mathbb{1} \perp x$*.*

5. *If* $x \perp y$ *and* $(x \uplus y) \perp z$*, then* $y \perp z$ *and* $x \perp (y \uplus z)$.
6. *If* $x \perp y$*, then* $y \perp x$*,* $\mathsf{valid}\, x$ *and* $\mathsf{valid}\, (x \uplus y)$.
7. $\preceq$ *is a* partial order.
8. *If* $x \perp y$*, then* $x \preceq x \uplus y$.

Definition 5 is an adaptation of the definition of (simple) separation algebras of [32] but does not include a difference relation $\setminus$. Furthermore, equality in the laws given above is defined up to a specified equivalence relation $\equiv$ rather than standard Leibniz equality. This allows to define a separation algebra for e.g. process maps, whose operations depend on equality up to bisimilarity.

Fractional permissions. Permission heaps and process maps both use Boyland's fractional permissions $\pi \in (0,1]_{\mathbb{Q}}$ [7] to associate ownership to shared memory and program abstractions, respectively, where $0 < \pi < 1$ denotes read access and $\pi = 1$ denotes both read and write access. To handle fractional permissions we define basic *validity* and *disjointness* as follows.

$$\mathsf{valid}_{\mathbb{Q}}\, \pi \triangleq 0 < \pi \leq 1 \qquad \pi_1 \perp_{\mathbb{Q}} \pi_2 \triangleq 0 < \pi_1 \wedge 0 < \pi_2 \wedge \pi_1 + \pi_2 \leq 1$$

Here, $\mathsf{valid}_{\mathbb{Q}} : \mathbb{Q} \to \mathit{Prop}$ determines whether the given rational number is within the range $(0,1]_{\mathbb{Q}}$. We refer to rationals π as *fractional permissions* if $\mathsf{valid}_{\mathbb{Q}}\, \pi$. The binary relation $\perp_{\mathbb{Q}} \colon \mathbb{Q} \to \mathbb{Q} \to \mathit{Prop}$ determines *disjointness* of two rationals. Disjoint rational numbers do not overlap, in the sense that both operands are fractional permissions, as well as their addition.

Lemma 1. *The following standard laws are satisfied.*

1. *If* $\pi_1 \perp_{\mathbb{Q}} \pi_2$*, then* $\pi_2 \perp_{\mathbb{Q}} \pi_1$*,* $\mathsf{valid}_{\mathbb{Q}}\, \pi_1$*, and* $\mathsf{valid}_{\mathbb{Q}}\, (\pi_1 + \pi_2)$.
2. *If* $\pi_1 \perp_{\mathbb{Q}} \pi_2$ *and* $(\pi_1 + \pi_2) \perp_{\mathbb{Q}} \pi_3$*, then* $\pi_2 \perp_{\mathbb{Q}} \pi_3$ *and* $\pi_1 \perp_{\mathbb{Q}} (\pi_2 + \pi_3)$.

Permission heaps. Figure 8 defines permission heaps, as well as four basic operations on permission heaps and their cells. Permission heaps are defined as total functions from heap locations to *permission heap cells*, which in turn are inductively defined to be one of the following five elements:

- free models an *unoccupied* heap cell. Having an explicit notion of heap cells being unoccupied gives desirable algebraic properties: free would act as unit in popular algebraic structures for modelling abstract state, including separation algebras [9,17].
- $\langle v \rangle^{\pi}_{\mathsf{std}}$ are *standard heap cells*, which model standard heap ownership ($\overset{\pi}{\hookrightarrow}_{\mathsf{std}}$) in the program logic. Standard heap cells store a value $v \in \mathit{Val}$ and are augmented with a rational $\pi \in \mathbb{Q}$ to administer its access rights in the proof system.
- $\langle v \rangle^{\pi}_{\mathsf{proc}}$ are *process heap cells*, which model process heap ownership in the program logic in the same style as $\langle v \rangle^{\pi}_{\mathsf{std}}$.

Permission heaps

$$ph \in \mathit{PermHeap} \triangleq \mathit{Loc} \to \mathit{PermHeapCell} \quad \text{(Permission heaps)}$$
$$hc \in \mathit{PermHeapCell} ::= \mathsf{free} \mid \langle v \rangle^{\pi}_{\mathsf{std}} \mid \langle v \rangle^{\pi}_{\mathsf{proc}} \mid \langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}} \mid \mathsf{inv} \quad \text{(Permission heap cells)}$$
$$\mathsf{valid}_{\mathsf{ph}}\, ph \triangleq \forall \ell \,.\, \mathsf{valid}_{\mathsf{hc}}\, ph(\ell) \quad \text{(Permission heap validity)}$$
$$ph_1 \perp_{\mathsf{ph}} ph_2 \triangleq \forall \ell \,.\, ph_1(\ell) \perp_{\mathsf{hc}} ph_2(\ell) \quad \text{(Permission heap disjointness)}$$
$$ph_1 \uplus_{\mathsf{ph}} ph_2 \triangleq \lambda \ell \,.\, ph_1(\ell) \uplus_{\mathsf{hc}} ph_2(\ell) \quad \text{(Disjoint union of permission heaps)}$$
$$ph_1 \preceq_{\mathsf{ph}} ph_2 \triangleq \forall \ell \,.\, ph_1(\ell) \preceq_{\mathsf{hc}} ph_2(\ell) \quad \text{(Permission subheaps)}$$

Permission heap cell validity

$$\mathsf{valid}_{\mathsf{hc}}\, \mathsf{free} \triangleq \mathsf{True}$$
$$\mathsf{valid}_{\mathsf{hc}}\, \langle v \rangle^{\pi}_{\mathsf{std}} \triangleq \mathsf{valid}_{\mathbb{Q}}\, \pi$$
$$\mathsf{valid}_{\mathsf{hc}}\, \langle v \rangle^{\pi}_{\mathsf{proc}} \triangleq \mathsf{valid}_{\mathbb{Q}}\, \pi$$
$$\mathsf{valid}_{\mathsf{hc}}\, \langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}} \triangleq \mathsf{valid}_{\mathbb{Q}}\, \pi$$
$$\mathsf{valid}_{\mathsf{hc}}\, \mathsf{inv} \triangleq \mathsf{False}$$

Permission heap cell disjoinness

$$\mathsf{free} \perp_{\mathsf{hc}} hc \triangleq \mathsf{valid}_{\mathsf{hc}}\, hc$$
$$hc \perp_{\mathsf{hc}} \mathsf{free} \triangleq \mathsf{valid}_{\mathsf{hc}}\, hc$$
$$\langle v \rangle^{\pi_1}_{\mathsf{std}} \perp_{\mathsf{hc}} \langle v \rangle^{\pi_2}_{\mathsf{std}} \triangleq \pi_1 \perp_{\mathbb{Q}} \pi_2$$
$$\langle v \rangle^{\pi_1}_{\mathsf{proc}} \perp_{\mathsf{hc}} \langle v \rangle^{\pi_2}_{\mathsf{proc}} \triangleq \pi_1 \perp_{\mathbb{Q}} \pi_2$$
$$\langle v_1, v_2 \rangle^{\pi_1}_{\mathsf{act}} \perp_{\mathsf{hc}} \langle v_1, v_2 \rangle^{\pi_2}_{\mathsf{act}} \triangleq \pi_1 \perp_{\mathbb{Q}} \pi_2$$
$$hc_1 \perp_{\mathsf{hc}} hc_2 \triangleq \mathsf{False}, \text{ otherwise}$$

Permission heap cell addition

$$\mathsf{free} \uplus_{\mathsf{hc}} hc \triangleq hc$$
$$hc \uplus_{\mathsf{hc}} \mathsf{free} \triangleq hc$$
$$\langle v \rangle^{\pi_1}_{\mathsf{std}} \uplus_{\mathsf{hc}} \langle v \rangle^{\pi_2}_{\mathsf{std}} \triangleq \langle v \rangle^{\pi_1+\pi_2}_{\mathsf{std}}$$
$$\langle v \rangle^{\pi_1}_{\mathsf{proc}} \uplus_{\mathsf{hc}} \langle v \rangle^{\pi_2}_{\mathsf{proc}} \triangleq \langle v \rangle^{\pi_1+\pi_2}_{\mathsf{proc}}$$
$$\langle v_1, v_2 \rangle^{\pi_1}_{\mathsf{act}} \uplus_{\mathsf{hc}} \langle v_1, v_2 \rangle^{\pi_2}_{\mathsf{act}} \triangleq \langle v_1, v_2 \rangle^{\pi_1+\pi_2}_{\mathsf{act}}$$
$$hc_1 \uplus_{\mathsf{hc}} hc_2 \triangleq \mathsf{inv}, \text{ otherwise}$$

Permission heap cell inclusion

$$\mathsf{free} \preceq_{\mathsf{hc}} hc \triangleq \mathsf{True}$$
$$\langle v \rangle^{\pi_1}_{\mathsf{std}} \preceq_{\mathsf{hc}} \langle v \rangle^{\pi_2}_{\mathsf{std}} \triangleq \pi_1 \leq \pi_2$$
$$\langle v \rangle^{\pi_1}_{\mathsf{proc}} \preceq_{\mathsf{hc}} \langle v \rangle^{\pi_2}_{\mathsf{proc}} \triangleq \pi_1 \leq \pi_2$$
$$\langle v_1, v_2 \rangle^{\pi_1}_{\mathsf{act}} \preceq_{\mathsf{hc}} \langle v_1, v_2 \rangle^{\pi_2}_{\mathsf{act}} \triangleq \pi_1 \leq \pi_2$$
$$\mathsf{inv} \preceq_{\mathsf{hc}} \mathsf{inv} \triangleq \mathsf{True}$$
$$hc_1 \preceq_{\mathsf{hc}} hc_2 \triangleq \mathsf{False}, \text{ otherwise}$$

Fig. 8: Definitions of basic operations on permission heaps- and heap cells.

- $\langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}}$ are *action heap cells*, which model action heap ownership ($\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$). In particular, action heap cells model shared memory that is subject to a program abstraction as well as modified in the context of an action block. Two values are stored: v_1 represents the current value of the heap cell (likewise to v in $\langle v \rangle^{\pi}_{\mathsf{std}}$ and $\langle v \rangle^{\pi}_{\mathsf{proc}}$), whereas v_2 is a *snapshot value*: a copy of the heap cell's value, made when entering the action block. Snapshot values are maintained for technical reasons only and are needed in the soundness proof of the proof system. This is further discussed in Section 6.2.
- inv models an *invalid* or *corrupted* heap cell. Invalid heap cells represent the erroneous result of composing two incompatible heap cells.

Permission heaps ph are *valid* if the permissions of all ph's heap cells are valid, where free is always valid and inv is never valid. Two permission heaps are *disjoint* (by $\perp_{\mathsf{ph}}$) if all their heap cells are pairwise compatible and their underlying fractional permissions are disjoint. *Disjoint union* $\uplus_{\mathsf{ph}}$ is only defined

for compatible heap cells, and otherwise gives the corrupted entry inv. The free heap cell is neutral with respect to $\uplus_{\mathsf{hc}}$, while inv is absorbing. Finally, $ph_1 \preceq_{\mathsf{ph}} ph_2$ expresses that ph_1 is a *subheap* of ph_2, meaning that all ph_1's cells are pointwise compatible with ph_2's cells and their permissions adhere to $\leq_{\mathbb{Q}}$.

Lemma 2 (Heap separation algebra). *The operations on permission heaps and permission heap cells defined in Figure 8 form a* separation algebra (of permission heaps and of permission heap cells, respectively).

Finally, we define a heap cell hc to be *full*, denoted as $\mathsf{full}_{\mathsf{p}}\, hc$, if hc is an occupied heap cell with an associated fractional permission that is 1.

Definition 6 (Full permission heap cells).

$$\frac{}{\mathsf{full}_{\mathsf{p}}\, \langle v \rangle^{1}_{\mathsf{std}}} \qquad \frac{}{\mathsf{full}_{\mathsf{p}}\, \langle v \rangle^{1}_{\mathsf{proc}}} \qquad \frac{}{\mathsf{full}_{\mathsf{p}}\, \langle v_1, v_2 \rangle^{1}_{\mathsf{act}}}$$

Process maps. Figure 9 defines process maps and their basic operations. Process maps are defined as total mappings from an infinite domain of *process identifiers* $\rho \in \mathit{ProcID} \subseteq \mathit{Val}$ to *process map cells*.

Likewise to permission heap cells, free models an *unoccupied* process map entry and inv an *invalid* entry. Occupied entries $\langle p, P, \Lambda \rangle^{\pi}$ model ownership of program abstractions (i.e. Proc_{π} assertions) in the program logic, where the process term P is the abstract model. The label p identifies the process declaration that was used to initialise the abstraction, like in Proc_{π}. Finally, Λ binds the abstraction P to the program by mapping its process variables directly to heap locations. The Λ binders are used as models for the Π components inside Proc_{π} predicates. For this purpose, we define the following evaluation function $[\![\cdot]\!] : \mathit{AbstrBinder} \to \mathit{Store} \to \mathit{Binder}$.

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

The $\cong_{\mathsf{pm}}$ relation is an equivalence relation that expresses *equality up to bisimulation*; two process maps are defined to be *bisimilar* if they are related by $\cong_{\mathsf{pm}}$ (and likewise for process map cells and $\cong_{\mathsf{mc}}$). Validity and disjointness of process maps are defined by lifting validity and disjointness of fractional permissions, in the same style as permission heaps. The disjoint union of two compatible non-empty process cells is the addition of their fractional permissions, together with the parallel composition of their process terms. The union of incompatible process cells—entries where the process label p or mapping Λ do not match—produces a corrupted result inv.

Lemma 3 (Process map separation algebra). *The operations on process maps and process map cells defined in Figure 9 form a* separation algebra (of process maps and of process map cells, respectively).

Process maps

$$pm \in \mathit{ProcMap} \triangleq \mathit{ProcID} \to \mathit{ProcMapCell} \qquad \text{(Process maps)}$$
$$\Lambda \in \mathit{Binder} \triangleq \mathit{ProcVar} \rightharpoonup_{\mathsf{fin}} \mathit{Loc} \qquad \text{(Abstraction binders)}$$
$$mc \in \mathit{ProcMapCell} ::= \mathsf{free} \mid \langle p, P, \Lambda \rangle^{\pi} \mid \mathsf{inv} \qquad \text{(Process map cells)}$$
$$pm_1 \cong_{\mathsf{pm}} pm_2 \triangleq \forall \rho \,.\, pm_1(\rho) \cong_{\mathsf{mc}} pm_2(\rho) \qquad \text{(Process map equality)}$$
$$\mathsf{valid}_{\mathsf{pm}}\, pm \triangleq \forall \rho \,.\, \mathsf{valid}_{\mathsf{mc}}\, pm(\rho) \qquad \text{(Process map validity)}$$
$$pm_1 \perp_{\mathsf{pm}} pm_2 \triangleq \forall \rho \,.\, pm_1(\rho) \perp_{\mathsf{mc}} pm_2(\rho) \qquad \text{(Process map disjointness)}$$
$$pm_1 \uplus_{\mathsf{pm}} pm_2 \triangleq \lambda \rho \,.\, pm_1(\rho) \uplus_{\mathsf{mc}} pm_2(\rho) \qquad \text{(Disjoint union of process maps)}$$
$$pm_1 \preceq_{\mathsf{pm}} pm_2 \triangleq \forall \rho \,.\, pm_1(\rho) \preceq_{\mathsf{mc}} pm_2(\rho), \text{ where} \qquad \text{(Process submap)}$$
$$mc_1 \preceq_{\mathsf{mc}} mc_2 \triangleq \exists pmc \,.\, mc \perp_{\mathsf{mc}} mc_1 \wedge mc_1 \uplus_{\mathsf{mc}} mc \cong_{\mathsf{mc}} mc_2 \qquad \text{(Proc. cell inclusion)}$$

Process cell equality

$$\mathsf{free} \cong_{\mathsf{mc}} \mathsf{free} \triangleq \mathsf{True}$$
$$\langle p, P_1, \Lambda \rangle^{\pi} \cong_{\mathsf{mc}} \langle p, P_2, \Lambda \rangle^{\pi} \triangleq P_1 \cong P_2$$
$$\mathsf{inv} \cong_{\mathsf{mc}} \mathsf{inv} \triangleq \mathsf{True}$$
$$mc_1 \cong_{\mathsf{mc}} mc_2 \triangleq \mathsf{False}, \text{ otherwise}$$

Process cell validity

$$\mathsf{valid}_{\mathsf{mc}}\, \mathsf{free} \triangleq \mathsf{True}$$
$$\mathsf{valid}_{\mathsf{mc}}\, \langle p, P, \Lambda \rangle^{\pi} \triangleq \mathsf{valid}_{\mathbb{Q}}\, \pi$$
$$\mathsf{valid}_{\mathsf{mc}}\, \mathsf{inv} \triangleq \mathsf{False}$$

Process cell disjointness

$$\mathsf{free} \perp_{\mathsf{mc}} mc \triangleq \mathsf{valid}_{\mathsf{mc}}\, mc$$
$$mc \perp_{\mathsf{mc}} \mathsf{free} \triangleq \mathsf{valid}_{\mathsf{mc}}\, mc$$
$$\langle p, P_1, \Lambda \rangle^{\pi_1} \perp_{\mathsf{mc}} \langle p, P_2, \Lambda \rangle^{\pi_2} \triangleq \pi_1 \perp_{\mathbb{Q}} \pi_2$$
$$mc_1 \perp_{\mathsf{mc}} mc_2 \triangleq \mathsf{False}, \text{ otherwise}$$

Process cell addition

$$\mathsf{free} \uplus_{\mathsf{mc}} mc \triangleq mc$$
$$mc \uplus_{\mathsf{mc}} \mathsf{free} \triangleq mc$$
$$\langle p, P_1, \Lambda \rangle^{\pi_1} \uplus_{\mathsf{mc}} \langle p, P_2, \Lambda \rangle^{\pi_2} \triangleq \langle p, P_1 \parallel P_2, \Lambda \rangle^{\pi_1 + \pi_2}$$
$$mc_1 \uplus_{\mathsf{mc}} mc_2 \triangleq \mathsf{inv}, \text{ otherwise}$$

Fig. 9: Definitions of the basic operations of process map cells.

4.2 Semantics of assertions

Figure 10 defines the interpretation of assertions, given as a satisfaction relation $ph, pm, s, g \models \mathcal{P}$, stating that the assertion $\mathcal{P}$ is satisfied by the model (ph, pm, s, g). Notably, the models contain two stores $s, g \in \mathit{Store}$. The extra store g, referred to as the *ghost store*, gives an interpretation to all variables used in ghost code. Such *ghost variables* do not interfere with regular program execution, and hence they are separated from program variables in the evaluation of assertions.

Heap ownership assertions $E \overset{\pi}{\hookrightarrow}_t E'$ are satisfied if the permission heap 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 $\mathsf{Proc}_{\pi}(X, p, P, \Pi)$ are satisfied if the process map holds a matching entry at position $g(X)$ with: a fractional permission at least π, and a process that is bisimilar to $P \parallel Q$, for some process Q. The latter enforces that the process map entry must at least include all the behaviours of the process P.

Semantics of assertions $\boxed{ph, pm, s, g \models \mathcal{P}}$

$$
\begin{aligned}
ph, pm, s, g \models B &\iff [\![B]\!](s) \\
ph, pm, s, g \models \forall X.\mathcal{P} &\iff \forall v \,.\, ph, pm, s[X \mapsto v], g[X \mapsto v] \models \mathcal{P} \\
ph, pm, s, g \models \exists X.\mathcal{P} &\iff \exists v \,.\, ph, pm, s[X \mapsto v], g[X \mapsto v] \models \mathcal{P} \\
ph, pm, s, g \models \mathcal{P} \vee \mathcal{Q} &\iff ph, pm, s, g \models \mathcal{P} \vee ph, pm, s, g \models \mathcal{Q} \\
ph, pm, s, g \models \mathcal{P} * \mathcal{Q} &\iff \exists ph_1, ph_2 \,.\, ph_1 \perp_{\mathsf{ph}} ph2 \wedge ph = ph_1 \uplus_{\mathsf{ph}} ph_2 \wedge \\
&\qquad \exists pm_1, pm_2 \,.\, pm_1 \perp_{\mathsf{pm}} pm2 \wedge pm \cong_{\mathsf{pm}} pm_1 \uplus_{\mathsf{pm}} pm_2 \wedge \\
&\qquad\quad ph_1, pm_1, s, g \models \mathcal{P} \wedge ph_2, pm_2, s, g \models \mathcal{Q} \\
ph, pm, s, g \models \mathcal{P} \mathrel{-\!\!*} \mathcal{Q} &\iff \forall ph', pm' . (ph \perp_{\mathsf{ph}} ph' \wedge pm \perp_{\mathsf{pm}} pm' \wedge ph', pm', s, g \models \mathcal{P}) \\
&\qquad \implies ph \uplus_{\mathsf{ph}} ph', pm \uplus_{\mathsf{pm}} pm', s, g \models \mathcal{Q} \\
ph, pm, s, g \models *_{i \in I} \mathcal{P}_i &\iff ph, pm, s, g \models \mathcal{P}_{i_0} * \cdots * \mathcal{P}_{i_n} \text{ for } I = \{i_0, \ldots, i_n\} \\
ph, pm, s, g \models E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E_2 &\iff \langle [\![E_2]\!](s) \rangle^{\pi}_{\mathsf{std}} \preceq_{\mathsf{hc}} ph([\![E_1]\!](s)) \\
ph, pm, s, g \models E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E_2 &\iff \langle [\![E_2]\!](s) \rangle^{\pi}_{\mathsf{proc}} \preceq_{\mathsf{hc}} ph([\![E_1]\!](s)) \\
ph, pm, s, g \models E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E_2 &\iff \exists v \,.\, \langle [\![E_2]\!](s), v \rangle^{\pi}_{\mathsf{act}} \preceq_{\mathsf{hc}} ph([\![E_1]\!](s)) \\
ph, pm, s, g \models \mathsf{Proc}_{\pi}(X, p, P, \pi) &\iff \langle p, P, [\![\Pi]\!](s) \rangle^{\pi} \preceq_{\mathsf{mc}} pm(g(X))
\end{aligned}
$$

Fig. 10: Semantic interpretation of assertions.

Lemma 4. *The* $\models$ *relation satisfies the following basic properties.*

(1) Monotonicity: *if* $ph_1 \preceq_{\mathsf{ph}} ph_2$ *and* $pm_1 \preceq_{\mathsf{pm}} pm_2$*, then* $ph_1, pm_1, s, g \models \mathcal{P}$ *implies* $ph_2, pm_2, s, g \models \mathcal{P}$*, provided that* $\mathsf{valid}_{\mathsf{ph}}\, ph_2$ *and* $\mathsf{valid}_{\mathsf{pm}}\, pm_2$*.*

(2) Closed under bisimulation: *if* $pm_1 \cong_{\mathsf{pm}} pm_2$*, then* $ph, pm_1, s, g \models \mathcal{P}$ *implies* $ph, pm_2, s, g \models \mathcal{P}$*.*

Property *(1)* in the above lemma states that adding resources does not invalidate the satisfiability of an assertion, which is a key property of intuitionistic separation logic. Property *(2)* is essential for allowing program abstractions to be replaced by bisimilar ones inside the program logic.

Let $[\![\mathcal{P}]\!] \triangleq \{(ph, pm, s, g) \mid ph, pm, s, g \models \mathcal{P}\}$ be the set of all models of the assertion $\mathcal{P}$. As usual, an assertion $\mathcal{P}$ is defined to *(semantically) entail* $\mathcal{Q}$, notation $\mathcal{P} \models \mathcal{Q}$, if every model of $\mathcal{P}$ is also a model of $\mathcal{Q}$.

Definition 7 (Semantic entailment). $\mathcal{P} \models \mathcal{Q} \triangleq [\![\mathcal{P}]\!] \subseteq [\![\mathcal{Q}]\!]$.

5 Proof System

This section discusses the structural rules (Section 5.1) and the proof rules (Section 5.2) of our program logic, which essentially extends the CSL of [59] by adding permission accounting [7,6] and machinery for handling program abstractions.

5.1 Logical entailment

Figure 11 shows the structural rules for the program logic. The notation $\mathcal{P} \dashv\vdash \mathcal{Q}$ is shorthand for $\mathcal{P} \vdash \mathcal{Q}$ and $\mathcal{Q} \vdash \mathcal{P}$ and thereby indicates that the rule can be used in both directions. All entailment rules are sound in the standard sense, namely that $\mathcal{P} \vdash \mathcal{Q}$ implies $\mathcal{P} \models \mathcal{Q}$.

The rule $\hookrightarrow$-SPLITMERGE expresses that heap ownership predicates $\overset{\pi}{\hookrightarrow}_t$ of any type t may be *split* (left-to-right direction) and *merged* (right-to-left direction) along π. Note however that multiple points-to predicates for the same heap location may only co-exist if they have the same ownership type, as indicated by the following rule.

$$\frac{t_1 \neq t_2}{E \overset{\pi_1}{\hookrightarrow}_{t_1} E' * E \overset{\pi_2}{\hookrightarrow}_{t_2} E' \vdash \mathsf{False}}$$

The PROC-$\cong$ rule allows to replace program abstractions by bisimilar ones. Finally, PROC-SPLITMERGE allows to distribute parallel processes over parallel threads. By splitting a predicate $\mathsf{Proc}_{\pi_1+\pi_2}(X, p, P_1 \parallel P_2, \Pi)$ into two, both parts can be distributed over different concurrent threads in the program logic, so that thread i can establish that it executes as prescribed by its part $\mathsf{Proc}_{\pi_i}(X, p, P_i, \Pi)$ of the abstract model. Afterwards, when the threads join again, the remaining partial abstractions can be merged back into a single predicate. This system provides a thread-modular way of verifying that programs meet their abstraction. The logical machinery of this is discussed in Section 5.2.

5.2 Proof rules

Figure 12 presents the standard CSL proof rules, whereas Figure 13 presents the extended proof rules of the program logic. Program judgements are sequents of the form $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$, with $\mathcal{R}$ a resource invariant and Γ an environment in the style of *interface specifications* of [42] that is defined as follows:

Definition 8 (Process environments).

$$(\mathit{ProcEnv}) \quad \Gamma \;::=\; \emptyset \mid \Gamma, \{b\}\, p\, \{b\}$$

These environments, which we refer to as *process environments*, describe assumptions as Hoare-triples $\{b_1\}\, p\, \{b_2\}$ for process terms, where p identifies the process declaration. These Hoare triples constitute the contracts of the process-algebraic abstractions that are defined for a program.

The rule ALLOC for heap allocation generates a new points-to ownership predicate of type std, indicating that the allocated heap location is not subject to any program abstraction. Heap deallocation (DISPOSE) requires a full *standard* ownernship predicate for the associated heap location, thereby making sure that the deallocation does not break any bindings of active program abstractions.

The PROCINIT rule handles initialisation of an abstract model P over a set of heap locations as specified by the Π mapping. This rule requires *standard* points-to predicates with write-permission for any heap location that is to be bound

Logical entailment $\boxed{\mathcal{P} \vdash \mathcal{Q}}$

$$\frac{}{\mathcal{P} * \mathcal{Q} \vdash \mathcal{P}}\ *\text{-WEAK} \qquad \frac{}{\mathcal{P} * (\mathcal{Q} * \mathcal{R}) \vdash (\mathcal{P} * \mathcal{Q}) * \mathcal{R}}\ *\text{-ASSOC} \qquad \frac{}{\mathcal{P} * \mathcal{Q} \vdash \mathcal{Q} * \mathcal{P}}\ *\text{-COMM} \qquad \frac{\mathcal{P} \vdash \mathcal{P}' \qquad \mathcal{Q} \vdash \mathcal{Q}'}{\mathcal{P} * \mathcal{Q} \vdash \mathcal{P}' * \mathcal{Q}'}\ *\text{-MONO}$$

$$\frac{}{\mathcal{P} \vdash \mathsf{True}}\ \mathsf{True}\text{-INTRO} \qquad \frac{}{\mathsf{False} \vdash \mathcal{P}}\ \mathsf{False}\text{-ELIM} \qquad \frac{\mathcal{P} * \mathcal{Q} \vdash \mathcal{R}}{\mathcal{P} \vdash \mathcal{Q} \mathrel{-\!\!*} \mathcal{R}}\ \mathrel{-\!\!*}\text{-INTRO} \qquad \frac{\mathcal{P} \vdash \mathcal{Q} \mathrel{-\!\!*} \mathcal{Q}'}{\mathcal{P} * \mathcal{Q} \vdash \mathcal{Q}'}\ \mathrel{-\!\!*}\text{-ELIM} \qquad \frac{\forall n\,.\,(\mathcal{P} \vdash \mathcal{Q}[X/n])}{\mathcal{P} \vdash \forall X.\mathcal{Q}}\ \forall\text{-INTRO}$$

$$\frac{\mathcal{P} \vdash \forall X.\mathcal{Q}}{\mathcal{P} \vdash \mathcal{Q}[X/n]}\ \forall\text{-ELIM} \qquad \frac{\mathcal{P} \vdash \mathcal{Q}[X/n]}{\mathcal{P} \vdash \exists X.\mathcal{Q}}\ \exists\text{-INTRO} \qquad \frac{\mathcal{P} \vdash \exists X.\mathcal{Q}}{\exists n\,.\,(\mathcal{P} \vdash \mathcal{Q}[X/n])}\ \exists\text{-ELIM} \qquad \frac{\mathcal{P} \vdash \mathcal{Q}_1}{\mathcal{P} \vdash \mathcal{Q}_1 \vee \mathcal{Q}_2}\ \vee\text{-ELIMLEFT}$$

$$\frac{\mathcal{P} \vdash \mathcal{Q}_2}{\mathcal{P} \vdash \mathcal{Q}_1 \vee \mathcal{Q}_2}\ \vee\text{-ELIMRIGHT} \qquad \frac{\pi_1 \perp_{\mathbb{Q}} \pi_2}{E_1 \overset{\pi_1+\pi_2}{\hookrightarrow}_t E_2 \dashv\vdash E_1 \overset{\pi_1}{\hookrightarrow}_t E_2 * E_1 \overset{\pi_2}{\hookrightarrow}_t E_2}\ \hookrightarrow\text{-SPLITMERGE}$$

$$\frac{}{*_{i \in I_1 \uplus I_2} \mathcal{P}_i \dashv\vdash (*_{i \in I_1} \mathcal{P}_i) * (*_{i \in I_2} \mathcal{P}_i)}\ \text{ITER-SPLITMERGE} \qquad \frac{P \cong Q}{\mathsf{Proc}_\pi(X, p, P, \Pi) \dashv\vdash \mathsf{Proc}_\pi(X, p, Q, \Pi)}\ \text{PROC-}\cong$$

$$\frac{\pi_1 \perp_{\mathbb{Q}} \pi_2}{\mathsf{Proc}_{\pi_1+\pi_2}(X, p, P_1 \parallel P_2, \Pi) \dashv\vdash \mathsf{Proc}_{\pi_1}(X, p, P_1, \Pi) * \mathsf{Proc}_{\pi_2}(X, p, P_2, \Pi)}\ \text{PROC-SPLITMERGE}$$

Fig. 11: Entailment rules of the program logic.

by P, and these are converted to $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$. Moreover, PROCINIT requires that the precondition B of P holds, which is constructed from b_1 by replacing all process variables by the values at the corresponding heap locations as specified by Π. A Proc_1 predicate with full permission is ensured, containing the label p of the declared process, so that the postcondition b_2 can later be retrieved from Γ.

The rule READ states that heap reading is allowed with any type of heap ownership, whereas heap writing (WRITE) is only allowed with a points-to predicate of type std or act; the $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ assertion exclusively grants read-access to the association location. We will in a moment see that the rule PROCUPDATE for action blocks can upgrade $E \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E'$ predicates to $E \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E'$ to regain write access to the heap location at E. This system of upgrading enforces that all modifications to E are captured by the program abstraction the heap location is subject to, inside an action block.

The PROCUPDATE rule handles updates to program abstractions, by performing an action a in the context of an **action** $X.a$ **do** C program. This rule imposes four preconditions on handling **action** programs. First, a predicate of the form $\mathsf{Proc}_\pi(X, p, a \cdot P + Q, \Pi)$ is required for some π. In particular, the process term must be of the form $a \cdot P + Q$ and therewith allow to perform the a action. After

Standard proof rules $\boxed{\Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,C\,\{\mathcal{Q}\}}$

$$\text{SKIP}\ \frac{}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,\mathbf{skip}\,\{\mathcal{P}\}} \qquad \text{SEQ}\ \frac{\Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,C_1\,\{\mathcal{P}'\} \qquad \Gamma;\mathcal{R}\vdash\{\mathcal{P}'\}\,C_2\,\{\mathcal{Q}\}}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,C_1;C_2\,\{\mathcal{Q}\}}$$

$$\text{ITE}\ \frac{\Gamma;\mathcal{R}\vdash\{\mathcal{P}*B\}\,C_1\,\{\mathcal{Q}\} \qquad \Gamma;\mathcal{R}\vdash\{\mathcal{P}*\neg B\}\,C_2\,\{\mathcal{Q}\}}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,\mathbf{if}\,B\,\mathbf{then}\,C_1\,\mathbf{else}\,C_2\,\{\mathcal{Q}\}}$$

$$\text{WHILE}\ \frac{\Gamma;\mathcal{R}\vdash\{\mathcal{P}*B\}\,C\,\{\mathcal{P}\}}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,\mathbf{while}\,B\,\mathbf{do}\,C\,\{\mathcal{P}*\neg B\}} \qquad \text{ASSIGN}\ \frac{X\notin\mathsf{fv}(\mathcal{R})}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}[X/E]\}\,X:=E\,\{\mathcal{P}\}}$$

$$\text{PAR}\ \frac{\mathsf{fv}(\mathcal{R},\mathcal{P}_1,C_1)\cap\mathsf{mod}(C_2)=\emptyset \qquad \mathsf{fv}(\mathcal{R},\mathcal{P}_2,C_2)\cap\mathsf{mod}(C_1)=\emptyset \qquad \Gamma;\mathcal{R}\vdash\{\mathcal{P}_1\}\,C_1\,\{\mathcal{Q}_1\} \qquad \Gamma;\mathcal{R}\vdash\{\mathcal{P}_2\}\,C_2\,\{\mathcal{Q}_2\}}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}_1*\mathcal{P}_2\}\,\mathbf{par}\,C_1\,C_2\,\{\mathcal{Q}_1*\mathcal{Q}_2\}}$$

$$\text{SHARE}\ \frac{\Gamma;\mathcal{R}*\mathcal{R}'\vdash\{\mathcal{P}\}\,C\,\{\mathcal{Q}\}}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}*\mathcal{R}'\}\,C\,\{\mathcal{Q}*\mathcal{R}'\}} \qquad \text{ATOMIC}\ \frac{\Gamma;\mathsf{True}\vdash\{\mathcal{P}*\mathcal{R}\}\,C\,\{\mathcal{Q}*\mathcal{R}\}}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,\mathbf{atomic}\,C\,\{\mathcal{Q}\}}$$

$$\text{CONSEQ}\ \frac{\mathcal{P}\vdash\mathcal{P}' \qquad \Gamma;\mathcal{R}\vdash\{\mathcal{P}'\}\,C\,\{\mathcal{Q}'\} \qquad \mathcal{Q}'\vdash\mathcal{Q}}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,C\,\{\mathcal{Q}\}}$$

$$\text{DISJ}\ \frac{\Gamma;\mathcal{R}\vdash\{\mathcal{P}_1\}\,C\,\{\mathcal{Q}_1\} \qquad \Gamma;\mathcal{R}\vdash\{\mathcal{P}_2\}\,C\,\{\mathcal{Q}_2\}}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}_1\vee\mathcal{P}_2\}\,C\,\{\mathcal{Q}_1\vee\mathcal{Q}_2\}}$$

$$\text{FRAME}\ \frac{\mathsf{fv}(\mathcal{F})\cap\mathsf{mod}(C)=\emptyset \qquad \Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,C\,\{\mathcal{Q}\}}{\Gamma;\mathcal{R}\vdash\{\mathcal{P}*\mathcal{F}\}\,C\,\{\mathcal{Q}*\mathcal{F}\}} \qquad \text{EX}\ \frac{\Gamma;\mathcal{R}\vdash\{\mathcal{P}\}\,C\,\{\mathcal{Q}\} \qquad X\notin\mathsf{fv}(C)}{\Gamma;\mathcal{R}\vdash\{\exists X.\mathcal{P}\}\,C\,\{\exists X.\mathcal{Q}\}}$$

Fig. 12: Standard proof rules of the program logic.

performing a the process term will be reduced to P and Q will be discarded, as the choice is made not to follow execution as prescribed by Q. To get processes in the required format, one may apply PROC-$\cong$ together with the standard axioms of process algebras. For example, processes of the form $a\cdot P$ can always be rewritten to $a\cdot P+\delta$ to obtain the required choice. Second, $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates are required for any heap location that is bound by Π. These points-to predicates are needed to resolve the guard and effect of a. Third, the guard of a must hold as a precondition. And last, the remaining resource $\mathcal{P}$ should hold.

Among the premises of PROCUPDATE is a proof derivation for the sub-program C, in which all required $\overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}}$ predicates are essentially upgraded to $\overset{\pi_i}{\hookrightarrow}_{\mathsf{act}}$ and thereby regain write access when $\pi_i = 1$. However, in case $\pi_i < 1$ the upgrade

Extended proof rules $\boxed{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}}$

READ
$$\frac{X \notin \mathsf{fv}(\mathcal{R}, E, E')}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}[X/E'] * E \overset{\pi}{\hookrightarrow}_t E'\}\, X := [E]\, \{\mathcal{P} * E \overset{\pi}{\hookrightarrow}_t E'\}}$$

WRITE
$$\frac{t \neq \mathsf{proc}}{\Gamma; \mathcal{R} \vdash \{E_1 \overset{1}{\hookrightarrow}_t -\}\, [E_1] := E_2\, \{E_1 \overset{1}{\hookrightarrow}_t E_2\}}$$

ALLOC
$$\frac{X \notin \mathsf{fv}(\mathcal{R}, E)}{\Gamma; \mathcal{R} \vdash \{\mathsf{True}\}\, X := \mathbf{alloc}\, E\, \{X \overset{1}{\hookrightarrow}_{\mathsf{std}} E\}}$$

DISPOSE
$$\frac{}{\Gamma; \mathcal{R} \vdash \{E \overset{1}{\hookrightarrow}_{\mathsf{std}} -\}\, \mathbf{dispose}\, E\, \{\mathsf{True}\}}$$

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

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

PROCFINISH
$$\frac{\mathsf{fv}(b) \subseteq \mathsf{dom}(\Pi) = \{x_0, \ldots, x_n\} \qquad I = \{0, \ldots, n\} \qquad B = b_2[x_i/E_i]_{\forall i \in I} \qquad P\!\downarrow}{\Gamma, \{b_1\}\, p\, \{b_2\}; \mathcal{R} \vdash \left\{\begin{array}{c} *_{i \in I} \Pi(x_i) \overset{1}{\hookrightarrow}_{\mathsf{proc}} E_i \\ * \mathsf{Proc}_1(X, p, P, \Pi) \end{array}\right\} \mathbf{finish}\, X \left\{*_{i \in I} \Pi(x_i) \overset{1}{\hookrightarrow}_{\mathsf{std}} E_i * B\right\}}$$

Fig. 13: The non-standard proof rules related to program abstractions.

does not give any additional privileges, since $\overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}}$ provides read-access just the same. We found that these unnecessary conversions complicate the soundness proof. To avoid unnecessary upgrades, we convert all affected $\overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}}$ predicates to auxiliary $\overset{\pi_i}{\hookrightarrow}_{\mathsf{proc|act}}$ predicates instead to simplify the correctness proof.

Definition 9 (Process-action ownership).

$$E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{proc|act}} E_2 \triangleq \begin{cases} E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E_2 & \textit{if } \pi = 1 \\ E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E_2 & \textit{otherwise} \end{cases}$$

The PROCUPDATE rule ensures a process ownership predicate that holds the resulting process P after execution of a. In addition, updates to the heap are ensured that comply with the postconditions of the proof derivation of C.

Finally, the PROCFINISH rule handles finalisation of abstractions that have fully been executed. A predicate $\mathsf{Proc}_1(X, p, P, \Pi)$ with *full* permission is required (thereby implying that no other thread can have a fragment of the abstraction), where P must be able to successfully terminate. Successful termination in this sense means that P is bisimilar to $\varepsilon + Q$ for some process Q, and thus has the choice to have no further behaviour. The Proc_1 predicate is exchanged for the postcondition B of the abstraction, again constructed by replacing all process variables in b_2 by concrete values obtained via points-to assertions. The postcondition B can be established at this point, since: *(i)* the contracts of processes in Γ are assumed, as their validity is checked externally, and b_2 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 process has been reduced to a point of successful termination. Hence all the classical assumptions of Hoare-triple reasoning are fulfilled. 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 subject to the abstraction.

6 Soundness

We now discuss soundness of the program logic. The soundness proof has been mechanised using Coq. Proving soundness was non-trivial and required substantial auxiliary definitions. This section discusses these auxiliary definitions and explains how they are used to construct the soundness argument. For further proof details we refer to the Coq development.

The soundness theorem relates program judgements with the operational semantics of programs and boils down to the following: if *(i)* a proof $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ can be derived and *(ii)* the contracts in Γ of all abstract models of C are satisfied, then C *executes safely for any number of computation steps*. Execution safety in this sense also includes that C does not fault for any number of execution steps, with respect to the fault semantics $\lightning$ defined in Section 3.

Our definition of execution safety extends the well-known inductive definition of configuration safety of Vafeiadis [59] by adding machinery to handle process-algebraic abstractions. The most important extension is a *simulation argument* between concrete program executions (with respect to $\rightsquigarrow$) and abstract program executions 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 referred to as the *ghost operational semantics*. This intermedate semantics is defined in Section 6.1 in terms of *ghost transitions* $\rightsquigarrow_{\mathsf{g}}$ that essentially 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 $\rightsquigarrow_{\mathsf{g}}$ steps, and vice versa, for n execution steps. Thus, the end-result is a *refinement* between programs and their abstractions.

In addition to establishing such refinements, our definition of execution safety must also allow the postconditions of abstractions to be used inside the program

logic, particularly in the context of the PROCFINISH proof rule. To account for these, the definition of execution safety uses two extra ingredients, both of which are defined in Section 6.2. The first ingredient is the notion of *process execution safety*, from which the semantics of process Hoare triples and of process environments Γ are defined. Informally, execution safety of a process Hoare triple $\{b_1\}\, p\, \{b_2\}$ states that all finite traces of $\mathsf{body}(p)$ starting from a state satisfying b_1, terminate in a state that satisfies b_2. The second ingredient is an invariant, stating that all *active* program abstractions *preserve their execution safety* for n execution steps, with respect to the current state of the program. Maintaining this invariant allows to obtain and use the postconditions of fully reduced abstractions when executing a **finish** _ ghost command.

Finally, Section 6.3 formally defines process execution safety—the semantic meaning of program judgements—and presents the soundness statement.

6.1 Ghost operational semantics

To establish the refinements between programs and their abstractions, an intermediate semantics is used that administers the states of all active program abstractions. This intermediate semantics is referred to in the sequel as the *ghost operational semantics*. The ghost semantics is expressed as a transition relation $\rightsquigarrow_{\mathrm{g}}$ between *ghost configurations* of the form (C, h, pm, s, g). Ghost configurations extend on program configurations by two extra components, namely:

- A process map pm that is used to administer the state of all active program abstractions; and
- An extra store $g \in \mathit{Store}$, referred to as a *ghost store*, as it is used to map variable names to process identifiers in the context of ghost instructions.

The ghost operational semantics uses two stores instead of one, to keep the administration of program data and ghost data strictly separated. By doing so, it is easier to establish that the variables referred to in ghost code do not interfere with regular program execution, and vice versa.

Ghost transitions essentially describe the *lock-step execution* of concrete programs ($\rightsquigarrow$ steps) and their program abstractions ($\xrightarrow{a}$ steps). An excerpt of the transition rules of the ghost semantics is presented in Figure 14. This excerpt only contains the transition rules related to program abstraction; all other transition rules are essentially the same as those of $\rightsquigarrow$, with the two extra configuration components simply carried over and left unchanged. Furthermore, note that also here the blue colourings are merely visual cues, indicating the parts of the program structure that only exist for specificational purposes. These colourings do not have any semantical meaning whatsoever.

To clarify the ghost transition rules, GHOST-PROCINIT instantiates a new program abstraction and stores it in a free entry in pm. Finalisation of program abstractions is handled by GHOST-PROCFINISH, under the condition that the process-algebraic abstraction is able to terminate successfully. The remaining three ghost transition rules handle the execution of action blocks. Before

Ghost operational semantics $\boxed{(C, h, pm, s, g) \rightsquigarrow_{\mathsf{g}} (C', h', pm', s', g')}$

GHOST-PROCINIT

$$\frac{pm(\rho) = \mathsf{free}}{(X := \textbf{process}\, p\, \textbf{over}\, \Pi, h, pm, s, g) \rightsquigarrow_{\mathsf{g}} (\textbf{skip}, h, pm[\rho \mapsto \langle p, \mathsf{body}(p), [\![\Pi]\!](s)\rangle^{1}], s, g)}$$

GHOST-PROCFINISH

$$\frac{g(X) = \rho \qquad pm(\rho) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^{\pi} \qquad P{\downarrow}}{(\textbf{finish}\, X, h, pm, s, g) \rightsquigarrow_{\mathsf{g}} (s, h, pm - \rho, s, g)}$$

GHOST-ACTINIT

$$\frac{}{(\textbf{action}\, X.a\, \textbf{do}\, C, h, pm, s, g) \rightsquigarrow_{\mathsf{g}} (\textbf{inact}\, (a, g(X), h)\, C, h, pm, s, g)}$$

GHOST-ACTSTEP

$$\frac{(C, h, pm, s, g) \rightsquigarrow_{\mathsf{g}} (C', h', pm', s', g')}{(\textbf{inact}\, m\, C, h, pm, s, g) \rightsquigarrow_{\mathsf{g}} (\textbf{inact}\, m\, C', h', pm', s', g')}$$

GHOST-ACTEND

$$\frac{pm(\rho) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^{\pi} \qquad P, \|\Lambda\|(h_{old}) \xrightarrow{a} P', \|\Lambda\|(h)}{(\textbf{inact}\, (a, \rho, h_{old})\, \textbf{skip}, h, pm, s, g) \rightsquigarrow_{\mathsf{g}} (\textbf{skip}, h, pm[\rho \mapsto \langle p, P', \Lambda\rangle^{\pi}], s, g)}$$

Fig. 14: Excerpt of the transition rules of the ghost operational semantics.

discussing these, first observe that the ghost semantics maintains an extra component m in $\textbf{inact}\, m\, C$ commands. This component contains *(ghost) metadata* regarding the program abstraction in whose context the action program C is being executed. Concretely, ghost metadata is defined as a triple $m \in \mathit{Act} \times \mathit{ProcID} \times \mathit{Heap}$ consisting of:

1. The label of the action that is being executed;
2. The identifier of the corresponding program abstraction in the process map;
3. A copy of the heap, made when the program started to execute the action block; that is, when the **action** program was reduced to **inact**.

The GHOST-ACTINIT transition rule starts to execute an **action** block by reducing it to an **inact** program, thereby assembling and attaching the required ghost metadata. In particular, a copy of the heap is made at this point, so that the GHOST-ACTEND transition rule for finalising **inact** programs is able to access the old contents of the heap. This is needed to allow the abstraction to make a matching $\xrightarrow{a}$ step; in particular to calculate the pre-state of such a step. To see how this works, first recall that the process-algebraic state of program abstractions are linked to concrete program state—entries in the heap—via the Λ binders that are maintained in process maps. Therefore, to be able to make an $\xrightarrow{a}$ step, the GHOST-ACTEND rule first needs to construct process-algebraic state out of the state of the program. This is done using the auxiliary function $\|\cdot\| : \mathit{Binder} \to \mathit{Heap} \to \mathit{ProcStore}$ that is referred to as the *abstract state reification function*, which has the following definition.

Ghost fault semantics $\boxed{\lightning_{\mathsf{g}}(C, h, pm, s, g)}$

$$
\begin{array}{rl}
\lightning_{\mathsf{g}}(X := \textbf{process}\, p\, \textbf{over}\, \Pi, h, pm, s, g) & \text{if there is no } \rho \text{ such that } pm(\rho) = \mathsf{free}. \\
\lightning_{\mathsf{g}}(\textbf{finish}\, X, h, pm, s, g) & \text{if } pm(g(X)) \in \{\mathsf{free}, \mathsf{inv}\}. \\
\lightning_{\mathsf{g}}(\textbf{finish}\, X, h, pm, s, g) & \text{if } pm(g(X)) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^{\pi} \text{ with } P \not\downarrow. \\
\lightning_{\mathsf{g}}(\textbf{inact}\, (a, \rho, h_{old})\, \textbf{skip}, h, pm, s, g) & \text{if } pm(\rho) \in \{\mathsf{free}, \mathsf{inv}\}. \\
\lightning_{\mathsf{g}}(\textbf{inact}\, (a, \rho, h_{old})\, \textbf{skip}, h, pm, s, g) & \text{if } pm(\rho) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^{\pi} \text{ and} \\
 & \quad \exists x \in \mathsf{dom}(\Lambda)\,.\, \Lambda(x) \notin \mathsf{dom}(h) \cap \mathsf{dom}(h'). \\
\lightning_{\mathsf{g}}(\textbf{inact}\, (a, \rho, h_{old})\, \textbf{skip}, h, pm, s, g) & \text{if } pm(\rho) \cong_{\mathsf{mc}} \langle p, P, \Lambda\rangle^{\pi} \text{ and there is no } P' \\
 & \quad \text{such that } P, \|\Lambda\|(h_{old}) \xrightarrow{a} P', \|\Lambda\|(h). \\
\lightning_{\mathsf{g}}(\textbf{inact}\, m\, C, h, pm, s, g) & \text{if } \lightning_{\mathsf{g}}(C, h, pm, s, g).
\end{array}
$$

Fig. 15: Excerpt of the ghost fault semantics.

Definition 10 (Abstract state reification).

$$
\|\Lambda\|(h) \triangleq \lambda x : \mathit{Proc}\,\mathit{Var}\,.\, \begin{cases} h(\Lambda(x)) & \text{if } x \in \mathsf{dom}(\Lambda) \text{ and } \Lambda(x) \in \mathsf{dom}(h) \\ \text{unspecified} & \text{otherwise} \end{cases}
$$

Faulting ghost configurations. Figure 15 presents an excerpt of the *ghost fault semantics*, defined in terms of a relation $\lightning_{\mathsf{g}}$ over ghost configurations, likewise to $\lightning$. Only the faulting configurations related to ghost code are shown; the other cases are similar to the ones presented in Figure 7.

Process initialisation may fault if there is no free entry in the process map. Process finalisation aborts if the referenced entry in the process map is: *(i)* either unoccupied or invalid, or *(ii)* contains a process-algebraic abstraction that is unable to successfully terminate. Finally, computation within action blocks $\textbf{inact}\, m\, C$ may fault if: *(i)* m does not refer to an abstraction, or *(ii)* the abstraction relies on process variables that have an incorrect binding, or *(iii)* the process is not able to make a matching step, or *(iv)* C is able to fault.

The ghost semantics enjoys the same progress property as the standard operational semantics as stated by Theorem 1.

Theorem 2 (Progress of $\rightsquigarrow_{\mathsf{g}}$). *For any ghost configuration* $(C, h, pm, s, g) \notin \lightning_{\mathsf{g}}$*, either* $C = \textbf{skip}$ *or there exists a ghost configuration* (C', h', pm', s', g') *such that* $(C, h, pm, s, g) \rightsquigarrow_{\mathsf{g}} (C', h', pm', s', g')$.

Moreover, it is quite straightforward to establish a *forward simulation* between $\rightsquigarrow$ and $\rightsquigarrow_{\mathsf{g}}$. A matching backward simulation is ensured by the soundness argument of the program logic, as is customary for establishing refinements [15].

Lemma 5 (Forward simulation). *The standard operational semantics and the fault semantics of programs are* embedded *in the ghost operational semantics and ghost fault semantics, respectively:*

1. *If* $(C, h, pm, s, g) \rightsquigarrow_{\mathsf{g}} (C', h', pm', s', g')$*, then* $C, h, s \rightsquigarrow C', h', s'$*.*
2. *If* $\lightning(C, h, s)$*, then* $\lightning_{\mathsf{g}}(C, h, pm, s, g)$ *for any process map* pm *and store* g*.*

Lemma 5 also shows that the ghost fault semantics extends the fault semantics of programs. The soundness argument later establishes that verified programs do not fault with respect to $\rightsquigarrow_{\mathsf{g}}$, and hence they also do not fault with respect to $\rightsquigarrow$.

6.2 Process execution safety

In addition to establishing refinements between programs and their abstractions, our notion of program execution safety (defined later, in Section 6.3) also needs logical mechanisms that allow the postconditions of finalised program abstractions to be used inside the program logic. These mechanisms are essential for establishing soundness of the PROCFINISH rule. We now discusses these mechanisms, consisting of the following two components:

- A notion of *process execution safety*, from which a semantical notion of correctness of process Hoare triples and of process environments can be defined.
- Machinery for expressing and maintaining an *invariant*, stating that all active program abstractions *preserve their execution safety* (that they established from the previous point, when they were initialised) throughout program execution, with respect to (reification of) the current program state.

To better discuss the use of such an invariant, let us first discuss the first item in the above, by defining process execution safety.

Definition 11 (Process execution safety). Execution safety *of a process term* P *with respect to a process store* σ *and a postcondition* b*, is defined in terms of a predicate* $\checkmark(P, \sigma, b)$*. This predicate is coinductively defined such that, if* $\checkmark(P, \sigma, b)$ *holds, then:*

(1) If $P\downarrow$*, then* $[\![b]\!](\sigma)$*, and*
(2) For any a*,* P' *and* σ'*, if* $P, \sigma \xrightarrow{a} P', \sigma'$*, then* $\checkmark(P', \sigma', b)$*.*

Thus, a process configuration (P, σ) *executes safely* with respect to a postcondition b, if for any configuration (P', σ') that can be reached from (P, σ) by zero or more $\xrightarrow{a}$ reductions, it holds that $[\![b]\!](\sigma')$ whenever $P'\downarrow$. It follows that the $\checkmark$ predicate is closed under bisimilarity.

Process execution safety is used to define partial correctness of process Hoare triples, and of process environments, in the following way.

Definition 12 (Semantics of process Hoare triples).

$$\models_{\mathsf{proc}} \{b_1\}\, p\, \{b_2\} \triangleq \forall \sigma \in \mathit{ProcStore}\,.\, [\![b_1]\!](\sigma) \implies \checkmark(\mathsf{body}(p), \sigma, b_2)$$

Definition 13 (Semantics of process environments).

$$\frac{}{\models_{\mathsf{env}} \emptyset} \qquad \frac{\models_{\mathsf{env}} \Gamma \qquad \models_{\mathsf{proc}} \{b_1\}\, p\, \{b_2\}}{\models_{\mathsf{env}} \Gamma, \{b_1\}\, p\, \{b_2\}}$$

The invariant mentioned in the preamble will express that all active program abstractions retain their execution safety throughout program execution, with respect to ✓. Since active program abstractions are administered in process maps, we lift the notion of program execution safety to *process map safety*, expressed as judgements of the form $\Gamma; h \models_{\mathsf{pm}} pm$. Intuitively, a process map pm is *safe* if all process-algebraic abstractions stored in pm execute safely with respect to ✓, together with their postconditions in Γ. The heap h represents the current program state, and is reified into process-algebraic state using $\|\cdot\|(h)$.

Definition 14 (Process map safety).

$$\Gamma; h \models_{\mathsf{pm}} pm \triangleq \forall \rho \in \mathit{ProcID} \,.\, \Gamma; h \models_{\mathsf{pc}} pm(\rho)$$

where $\Gamma; h \models_{\mathsf{pc}} pc$ *is defined by case distinction on* pc*, so that*

$$\frac{}{\Gamma; h \models_{\mathsf{pc}} \mathsf{free}} \qquad \frac{\checkmark(P, \|\Lambda\|(h), b_2)}{\{b_1\}\, p\, \{b_2\}, \Gamma; h \models_{\mathsf{pc}} \langle p, P, \Lambda\rangle^{\pi}}$$

Free process cells are always safe, whereas invalid entries inv are never safe. Moreover, the judgements $\models_{\mathsf{pm}}$ and $\models_{\mathsf{pc}}$ are both closed under bisimilarity.

The next section formally defines *program execution safety*. To prove soundness of the PROCFINISH rule, program execution safety maintains the aforementioned invariant that $\Gamma; h \models_{\mathsf{pm}} pm$ always holds throughout program execution, where h and pm are constructed from the current state at every execution step. This allows to prove soundness of the PROCFINISH proof rule, as it requires the concerned process term to be able to successfully terminate, and thus by Definition 11 must satisfy the postcondition of the abstraction, via Definition 13.

However, one has to be careful on how to exactly state this invariant, to allow it to be re-established after every computation step. In most cases re-establishing the invariant is straightforward. For example, $\Gamma; h \models_{\mathsf{pm}} pm$ can be re-established after initialising a new program abstraction using the PROCINIT proof rule, by Definition 12 and the structure of that proof rule. The invariant can also trivially be re-established after finalising an abstraction using PROCFINISH, as the abstraction is then no longer active and thereby removed from pm. However, computation steps that involve heap writing (i.e. handling of $[E] := E'$ programs) may be problematic. To see the potential problem, consider the following code snippet.

```
1 requires x > 0;
2 ensures x = 0;
3 action reset;
```

```
4 {Proc_π(X, p, reset · P, {x ↦ E}) ∗ ···}
5 action X.reset do {
6     [E] := −2; // the problem is here
7     [E] := 0;
8 }
9 {Proc_π(X, p, P, {x ↦ E}) ∗ ···}
```

Suppose that $\Gamma; h \models_{\mathsf{pm}} pm$ holds on line 5. After computing line 6, the heap h holds the value -2 at location $[\![E]\!](s)$. Moreover, the process map pm has

not been changed, because the action program (lines 5–8) has not fully been executed yet. Nevertheless, $\Gamma; h[[\![E]\!](s) \mapsto -2] \models_{\mathsf{pm}} pm$ may now be violated, as the reset action can no longer be performed, since $x = -2$ after reification, while reset's precondition requires x to be positive.

The root of the problem, is that the invariant should not necessarily have to hold during intermediate steps while executing **action** programs, but only at the pre- and poststate of the action program. Program execution safety will solve this by making a *snapshot of the heap* every time an action program is being started on (likewise to GHOST-ACTINIT), and expressing the invariant over these snapshot heaps. Snapshots are recorded at the level of permission heaps, which already have the required structure to do this: action heap cells $\langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}}$ allow to store *snapshot values* v_2 alongside "concrete" values v_1. These snapshot values are used to construct *snapshot heaps*, with help of the following operation.

Definition 15 (Snapshot heap). *The* snapshot *of a permission heap is defined in terms of a total function* $\lfloor \cdot \rfloor_{\mathsf{snapshot}} : \mathit{PermHeap} \to \mathit{Heap}$*, so that*

$$\lfloor ph \rfloor_{\mathsf{snapshot}} \triangleq \lambda \ell : \mathit{Loc} \,.\, \lfloor ph(\ell) \rfloor_{\mathsf{snapshot}}$$

where $\lfloor hc \rfloor_{\mathsf{snapshot}}$ *is defined by case distinction on* hc*, so that*

$$\begin{aligned} \lfloor \langle v \rangle^{\pi}_{\mathsf{proc}} \rfloor_{\mathsf{snapshot}} &\triangleq v \\ \lfloor \langle -, v \rangle^{\pi}_{\mathsf{act}} \rfloor_{\mathsf{snapshot}} &\triangleq v \\ \lfloor hc \rfloor_{\mathsf{snapshot}} &\triangleq \mathsf{undef}, \textit{ in all other cases} \end{aligned}$$

The snapshot $\lfloor ph \rfloor_{\mathsf{snapshot}}$ of a permission heap ph only contains heap cells bound by abstract models, and is constructed by taking the snapshot values of all ph's action heap cells. As we shall see in Section 6.3, the final invariant maintained by program execution safety will be $\Gamma; \lfloor ph \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm$, where ph and pm are taken from the models of the program logic and represent the current state of the program. This invariant, combined with establishing a refinement between the program and its abstractions, provide sufficient means for proving soundness of the program logic.

6.3 Adequacy

This section defines *program execution safety* and uses this definition to define the semantic meaning of program judgements and thereby to formulate the soundness theorem. Program execution safety extends on the well-known notion of configuration safety, as proposed in [59], by adding permission accounting, process-algebraic state, and the machinery introduced in Sections 6.1 and 6.2.

First, in order to help connect the models of the program logic to concrete program state, we define a *concretisation function* for permission heaps.

Definition 16 (Concretisation). Concretisation *of permission heaps is defined as a total function* $\lfloor \cdot \rfloor_{\mathsf{concr}} : \mathit{PermHeap} \to \mathit{Heap}$*, so that*

$$\lfloor ph \rfloor_{\mathsf{concr}} \triangleq \lambda \ell : \mathit{Loc} \,.\, \lfloor ph(\ell) \rfloor_{\mathsf{concr}}$$

where $\lfloor hc \rfloor_{\mathsf{concr}}$ *is defined by case distinction on* hc*, so that*

$$\begin{aligned}
\lfloor \langle v \rangle^{\pi}_{\mathsf{std}} \rfloor_{\mathsf{concr}} &\triangleq v \\
\lfloor \langle v \rangle^{\pi}_{\mathsf{proc}} \rfloor_{\mathsf{concr}} &\triangleq v \\
\lfloor \langle v, - \rangle^{\pi}_{\mathsf{act}} \rfloor_{\mathsf{concr}} &\triangleq v \\
\lfloor hc \rfloor_{\mathsf{concr}} &\triangleq \mathsf{undef}, \textit{ in all other cases}
\end{aligned}$$

Heap concretisation constructs concrete program heaps out of a permission heaps by simply discarding all internal structure regarding abstract models. Only the information relevant for regular program execution is retained. $\lfloor \cdot \rfloor_{\mathsf{snapshot}}$ essentially does the same, but only retains heap cells bound to program abstractions and prefers to take snapshot values whenever possible.

We now have all the ingredients needed to express adequacy. *Safety of program execution* is defined in terms of a predicate $\mathsf{safe}^n_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$, stating that the program C is *safe* for n computation steps with respect to a permission heap ph, a process map pm, two stores s and g, a resource invariant $\mathcal{R}$, and a postcondition $\mathcal{Q}$.

Definition 17 (Program execution safety). *The* $\mathsf{safe}^0_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$ *predicate always holds, whereas* $\mathsf{safe}^{n+1}_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$ *holds if and only if the following five conditions hold.*

1. *If* $C = \mathbf{skip}$*, then* $ph, pm, s, g \models \mathcal{Q}$*.*
2. *For every* ph_F *and* pm_F *such that* $ph \perp_{\mathsf{hc}} ph_F$ *and* $pm \perp_{\mathsf{mc}} pm_F$*, it holds that* $(C, \lfloor ph \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}}, pm \uplus_{\mathsf{pm}} pm_F, s, g) \notin \lightning_{\mathsf{g}}$*.*
3. *For any* $\ell \in \mathsf{acc}(C, s)$ *it holds that* $ph(\ell) \notin \{\mathsf{free}, \mathsf{inv}\}$*.*
4. *For any* $\ell \in \mathsf{mod}(C, s)$ *it holds that* $\mathsf{full}_{\mathsf{p}}(ph(\ell))$*.*
5. *For any* ph_J*,* phF*,* pm_J*,* pm_F*,* pm_C*,* h'*,* s'*, and* C' *such that, if:*

   5a. $ph \perp_{\mathsf{ph}} ph_J$ *and* $(ph \uplus_{\mathsf{ph}} ph_J) \perp_{\mathsf{ph}} ph_F$*, and*
   5b. $pm \perp_{\mathsf{pm}} pm_J$ *and* $(pm \uplus_{\mathsf{pm}} pm_J) \perp_{\mathsf{pm}} pm_F$*, and*
   5c. $\neg\mathsf{locked}\, C$ *implies* $ph_J, pm_J, s, g \models \mathcal{R}$*, and*
   5d. $(pm \uplus_{\mathsf{pm}} pm_J \uplus_{\mathsf{pm}} pm_F) \cong_{\mathsf{pm}} pm_C$*, and*
   5e. $\Gamma, \lfloor ph \uplus_{\mathsf{ph}} ph_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm_C$*, and*
   5f. $C, \lfloor ph \uplus_{\mathsf{ph}} ph_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}}, s \rightsquigarrow C', h', s'$*;*

   then there exists ph'*,* ph'_J*,* pm'*,* pm'_J*,* pm'_C*, and* g' *such that*

   5g. $ph' \perp_{\mathsf{ph}} ph'_J$ *and* $(ph' \uplus_{\mathsf{ph}} ph'_J) \perp_{\mathsf{ph}} ph_F$*, and*
   5h. $pm' \perp_{\mathsf{pm}} pm'_J$ *and* $(pm' \uplus_{\mathsf{pm}} pm'_J) \perp_{\mathsf{pm}} pm_F$*, and*
   5i. $\lfloor ph' \uplus_{\mathsf{ph}} ph'_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}} = h'$*, and*
   5j. $(pm' \uplus_{\mathsf{pm}} pm'_J \uplus_{\mathsf{pm}} pm_F) \cong_{\mathsf{pm}} pm'_C$*, and*
   5k. $\Gamma, \lfloor ph' \uplus_{\mathsf{ph}} ph'_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm'_C$*, and*
   5l. $\neg\mathsf{locked}\, C'$ *implies* $ph'_J, pm'_J, s', g' \models \mathcal{R}$*, and*
   5m. $(C, \lfloor ph \uplus_{\mathsf{ph}} ph_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}}, pm_C, s, g) \rightsquigarrow_{\mathsf{g}} (C', h', pm'_C, s', g')$*, and*
   5n. $\mathsf{safe}^n_\Gamma(C', ph', pm', s', g', \mathcal{R}, \mathcal{Q})$*.*

To clarify, any configuration is safe for $n+1$ steps if: the postcondition is satisfied if the program C has terminated (***1***); the program C does not fault (***2***); C only accesses heap entries that are allocated (***3***); C only writes to heap locations for which full permission is available (***4***); and finally, after making a computation step the program remains safe for another n steps (***5***). Condition ***2*** implies race freedom, while conditions ***3*** and ***4*** account for memory safety.

Condition ***5*** is particularly involved. In particular, it encodes the backward simulation: if the program can do a $\rightsquigarrow$ step (***5f***), then it must be able to make a matching $\rightsquigarrow_{\mathsf{g}}$ ghost step (***5m***). Moreover, the resource invariant $\mathcal{R}$ must remain satisfied (due to ***5c*** and ***5l***) after making a computation step, whenever the program is not locked. In addition, the process maps invariably remain safe with respect to Γ and the snapshot heap due to ***5e*** and ***5k***, as discussed in the previous section. All the other (sub-)conditions are for the most part standard.

Lemma 6. *Program execution safety satisfies the following properties.*

1. safe^n_Γ *is* monotone *with respect to* n*: if* $\mathsf{safe}^n_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$ *and* $m \leq n$*, then* $\mathsf{safe}^m_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$.
2. safe^n_Γ *is* closed under $\cong_{\mathsf{pm}}$*: if* $\mathsf{safe}^n_\Gamma(C, ph, pm_1, s, g, \mathcal{R}, \mathcal{Q})$ *and* $pm_1 \cong_{\mathsf{pm}} pm_2$*, then* $\mathsf{safe}^n_\Gamma(C, ph, pm_2, s, g, \mathcal{R}, \mathcal{Q})$.

The semantics of program judgements is defined as judgements $\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$, expressing that C is safe for any number of execution steps, starting from any state satisfying the precondition $\mathcal{P}$.

Definition 18 (Semantics of program judgements). $\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ *holds if and only if:*

(1) C *is a user program; and*
(2) If $\models_{\mathsf{env}} \Gamma$*, then, for any* n*,* ph*,* pm*,* s*,* g*, if:*
  (2a) $\mathsf{valid}_{\mathsf{ph}}\, ph$ *and* $\mathsf{valid}_{\mathsf{pm}}\, pm$*, and*
  (2b) $\mathsf{wf}(C)$*, and*
  (2c) $ph, pm, s, g \models \mathcal{P}$*;*
  then $\mathsf{safe}^n_\Gamma(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$.

Theorem 3 (Soundness). $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ *implies* $\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$.

The soundness proofs of all proof rules have been mechanised using the Coq proof assistant and can be found on the public Git repository[4]. The PROCUPDATE proof rule was the most difficult to prove sound, as it includes, among other things: *(i)* showing that the abstraction can match the program with a simulating step, as well as *(ii)* handling the invariant that was discussed in Section 6.2. On top of that, the combination of *(i)* and *(ii)* requires some extra bookkeeping to ensure that the snapshot heaps stored in ghost metadata (discussed in Section 6.1) agree with the snapshot values stored in permission heaps.

[4] But for now, we have provided all supplementary material as a ZIP file.

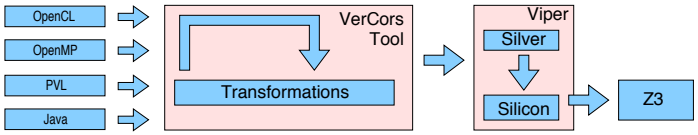

Fig. 16: Workflow of the VerCors toolset. The Viper infrastructure is maintained at ETH Zurich.

7 Implementation

The presented verification approach has been implemented in the VerCors tool set, which specialises in automated verification of parallel and concurrent programs written in high-level languages like (subsets of) Java and C [4]. VerCors can reason about programs using both heterogeneous concurrency (as in Java) and homogeneous concurrency (e.g. OpenCL), as well as compiler directives (as in OpenMP). VerCors allows (concurrent) programs to be specified with permission-based separation logic annotations. VerCors supports reasoning about data-race freedom, memory safety and functional program behaviour.

Figure 16 gives an overview of the workflow of VerCors. Several input languages are supported, including OpenCL and OpenMP for C, Java, and PVL (Prototype Verification Language). The latter is a procedural Java-like language used by the developers to prototype new verification features. VerCors is essentially a set of compiler transformations that translate input verification problems to input to the Viper verification infrastructure [39] (maintained at ETH Zurich). Viper eventually transforms the verification problem into an SMT-solving problem, handled by Z3. The main goal of VerCors is lifting existing verification technology, viz. Viper, to high-level languages and advanced concurrency features, rather than developing new verification technology.

Tool support. Tool support for our technique has been implemented in VerCors for languages with fork/join concurrency and statically-scoped parallel constructs [43] (including a verified implementation of the Owicki-Gries example of Section 2). Our technique has been implemented by defining an axiomatic domain for process types in Viper, consisting of constructors for the process-algebraic connectives and standard process-algebraic axioms to support these. The three different ownership types $\overset{\pi}{\hookrightarrow}_t$ are encoded in Viper by defining extra fields that maintain the ownership status t for each global reference. The Proc_π assertions are encoded as predicates over process types.

VerCors is able to reason about process-algebraic abstractions, by first linearising process terms and then encoding the linear processes and their contracts into Viper input. A process term is *linear* if it does not use the $\|$ and $\lfloor\!\lfloor$ connectives. The linearisation algorithm is based on a rewrite system that uses a subset of the standard process-algebraic axioms as rewrite rules [57] to eliminate parallel connectives. For example, a process term $(a_1 \cdot a_2) \parallel a_3$ can be linearised to the bisimilar process $a_1 \cdot a_2 \cdot a_3 + a_1 \cdot a_3 \cdot a_2 + a_3 \cdot a_1 \cdot a_2$. Al-

ternatively, process-algebraic abstractions may also algorithmically be analysed; we are currently investigating the use of the mCRL2 [21] toolset and the Ivy verifier [46].

Moreover, the VerCors implementation of the abstraction technique 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 process and action declarations to be parameterised by data. VerCors also has support for several axiomatic data types that enrich the expressivity of reasoning with program abstractions, like (multi)sets, bags, sequences, and option types.

Coq formalisation. The formalisation and soundness proof (Sections 3–6) of the program logic have been fully mechanised using Coq, as a deep embedding that is inspired by [59]. The overall implementation comprises roughly 15.000 lines of code. The Coq development and its documentation can be found at: https://github.com/utwente-fmt/FM19-ProgramAbstractions

8 Case Study

This section demonstrates our verification approach on a bigger case, that involves verifying the correctness of a classical distributed algorithm, namely a leader election protocol [44]. The algorithm is performed by N distributed 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 those workers. To find a leader, the election procedure assumes that each worker i receives a unique integer value v_i to start with, i.e. $v_i = v_j$ implies $i = j$, and then operates in N rounds. In every round, each worker sends the highest value it encountered so far to its right neighbour and in turn receives a value from its left neighbour, and remembers the highest of the two. The result after N rounds is that all workers know the highest unique value $\mathsf{max}\{v_0, \ldots, v_N\}$ in the network, allowing its original owner to announce itself as leader.

The case study has been verified with VerCors via an implementation in PVL. Since VerCors does not yet have native support for message passing we first implemented basic asynchronous message passing functionality in PVL. This implementation consists of two non-blocking operations for standard communication: `mp_send`(r, msg) for sending a message msg to the worker with rank r[5], and $msg :=$ `mp_recv`(r) for receiving a message msg 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 system of message passing on the abstraction level that matches the implementation, using the techniques that have been presented so far. To design such a system we follow the ideas of [44]. First we define two process-algebraic actions, $\mathsf{send}(r, msg)$ and

[5] The identifiers of workers are typically called *ranks* in message passing terminology.

$\mathsf{recv}(r, msg)$, that abstractly describe the behaviour of the concrete implementations in `mp_send` and `mp_recv`, respectively. Secondly, to properly handle message receival on the abstraction level we also need to define *process-algebraic summation*. A summation operator $\Sigma_{x \in D} P$ allows to quantify over a set of data $D = \{d_0, \ldots, d_n\}$ and has the behaviour of $P[x/d_0] + \cdots + P[x/d_n]$. We use process-algebraic summation to quantify over the possible messages that `mp_recv` might receive. The following two rules illustrate how the abstract send and recv actions are connected to `mp_send` and `mp_recv`.

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

Note that the send and recv actions are both parameterised by data. 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.

And last, we use send and recv to construct a process-algebraic model of the leader election protocol and verify that the implementation adheres to this model. This model has been analysed with mCRL2 to establish the global property that the correct leader is announced—the one with the highest initial value. From this it follows that the implementation determines the correct leader.

8.1 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 the right abstraction level, as the behaviour of leader election can concisely be specified in terms of sequences of sends and receives. Figure 17 presents the process-algebraic specification. In particular, ParElect specifies the *global* behaviour of the protocol, whereas Elect specifies the *thread-local* behaviour. Ideally the send and recv actions would be part of a native message passing library. This is planned as future work.

The ParElect process encodes the parallel composition of the eligible participants. ParElect takes a sequence vs of initial values as argument, whose length equals the total number of workers due to 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. We used mCRL2 to verify that ParElect is a correct abstract specification of the election protocol, with respect to its contract.

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 (line 20), receive a value v' in return from their left neighbour (line 21), and continue with the highest of the two.

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

Fig. 17: Behavioural specification of the leader election protocol.

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, where messages 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 $\mathsf{send}(r, msg)$ expresses enqueuing the message msg onto the message queue $chan[r]$ of the worker with rank r. In more detail, the precondition of $\mathsf{send}(r, msg)$ expresses that r must be a valid rank in the network. Note that, since every worker receives one message queue we have that $|chan|$ is equal to N—the total number of workers. 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

```
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 ↪¹_proc − ∗∗ ∃c : seq⟨seq⟨Msg⟩⟩ . C ↪¹_proc c ∗∗ N ↪^{1/2}_proc |c|;
6
7  given p, P, 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, Π);
10 ensures Proc_π′(X, p, P, Π);
11 void mp_send(ProcID X, int rank, Msg msg) { /∗ see Figure 20 ∗/ }
12
13 given p, P, 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, Π);
16 ensures Proc_π′(X, p, P[m/\result], Π);
17 Msg mp_recv(ProcID X, int rank) { /∗ see Figure 20 ∗/ }
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(ProcID X, int rank, Msg v0, Msg v) {
24   loop_invariant 0 ≤ i ≤ n;
25   loop_invariant Proc_π′(X, p, Elect(rank, v0, v, 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), i − 1),
29       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′), i − 1), p := p, Π := Π, π := π, π′ := π′
33     };
34     v := max(v, v′);
35   }
36   if (v = v0) {
37     atomic { action X.announce(rank) do L := rank; }
38   }
39 }
```

Fig. 18: Annotated implementation of the leader election protocol.

of $\mathsf{recv}(r, msg)$ 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.

8.2 Protocol implementation

Figure 18 presents the annotated implementation of the election protocol[6]. The `elect` method contains the code that is executed by every worker. The contract of $\texttt{elect}(X, rank, v_0, v)$ states that the method body adheres to the behavioural description $\mathsf{Elect}(rank, v_0, v, N)$ of the election protocol. The annotation **context** $\mathcal{P}$ is shorthand for **requires** $\mathcal{P}$; **ensures** $\mathcal{P}$. 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 $\mathsf{Elect}(rank, v_0, v, i)$. The invocations to `mp_send` and `mp_recv` on 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** $\overline{\chi}$ annotation expresses that the parameter list $\overline{\chi}$ are extra ghost arguments for the sake of specification. After N rounds all workers with $v = v_0$ announce 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.

Figure 19 presents bootstrapping code for the message passing implementation. The `main` function initialises the communication channels whereas `parelect` spawns all worker threads. $\texttt{main}(vs)$ additionally initialises and finalises the abstraction $\mathsf{ParElect}(vs)$ on the specification level (line 61 and 65, respectively), whose analysis allows to establish the postconditions of `main`. The function $\texttt{parelect}(X, vs)$ implements the abstract model $\mathsf{ParElect}(vs)$ by spawning N workers that all execute the `elect` program. The contract associated to the parallel block (lines 45–47) is called an *iteration contract* and assigns pre- and postconditions to every parallel instance. For more details on iteration contracts we refer to [3]. Most importantly, the iteration contract of each parallel worker states (on line 46) 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 49–50.

8.3 Communication primitives

Figure 20 presents the implementation of the message passing system. The $\texttt{mp\_send}(X, rank, msg)$ method implements the operation of enqueuing msg onto the message queue of worker $rank$. The contract of `mp_send` expresses that the enqueuing operation is encapsulated as a $\mathsf{send}(rank, msg)$ action that is prescribed by an abstract model identified by X. The program performs the send action by means of an action block that updates C by enqueuing msg. The result

[6] It should be noted that the presentation is slightly different from the PVL 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 [35] as the underlying logical framework, which is equivalent to Separation Logic [47] but handles ownership slightly differently. The details of this are deferred to [4,28].

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

Fig. 19: Bootstrap procedures of the leader election protocol.

is that send has been performed in the post-state of `mp_send` (line 4). In order for X to be able to perform the send action, all send's preconditions have to be satisfied. For this purpose line 2 requires that N is a reference field to some integer n that represents the total number of workers, and that $rank$ is between 0 and n.

The `mp_recv`$(X, rank)$ function implements the operation of dequeuing and returns the first message in 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 13.

The implementation of `mp_recv`(X, r) simply checks in a busy-loop whether $C[r]$ contains a message, and if so, pops the first available message of $C[r]$ as a recv action. This message will eventually be returned on line 29. The resulting abstraction after termination of `mp_recv`, as by line 14 is the trailing process P with the quantified variable m substituted for the returned message.

```
1  given p, P, Π, π, π';
2  context {chan ↦ C} ∈ Π ∗∗ ∃n . N ↪→^π_proc n ∗∗ 0 ≤ rank < n;
3  requires Proc_π'(X, p, send(rank, msg) · P, Π);
4  ensures Proc_π'(X, p, P, Π);
5  void mp_send(ProcID X, int rank, Msg msg) {
6    atomic {
7      action X.send(rank, msg) do C[rank] := C[rank] + {msg};
8    }
9  }
10
11 given p, P, Π, π, π';
12 context {chan ↦ C} ∈ Π ∗∗ ∃n . N ↪→^π_proc n ∗∗ 0 ≤ rank < n;
13 requires Proc_π'(X, p, Σ_{m∈Msg} recv(rank, m) · P, Π);
14 ensures Proc_π'(X, p, P[m/\result], Π);
15 Msg mp_recv(ProcID X, int rank) {
16   bool stop := False;
17   Msg msg;
18   loop_invariant ¬stop ⇒ Proc_π'(X, p, Σ_{m∈Msg} recv(rank, m) · P, Π);
19   loop_invariant stop ⇒ Proc_π'(X, p, P[m/msg], Π);
20   while (¬stop) {
21     atomic (this) {
22       if (0 < |C[rank]|) {
23         msg := C[rank][0];
24         action X.recv(rank, msg) do C[rank] := C[rank][1..];
25         stop := True;
26       }
27     }
28   }
29   return msg;
30 }
```

Fig. 20: A basic implementation of message passing, whose behaviour is specified in terms of the send and recv actions that were defined in Figure 17.

9 Related Work

Significant progress has been made on the theory of concurrent software verification over the last years [19,16,54,55,56,40,18,13]. This line of research proposes advanced program logics that all provide some notion of expressing and restricting thread interference of various complexity, via *protocols* [31]: 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 [41] allows to specify simple thread interference in shared-memory programs via resource invariants and critical regions. Later, RGSep [60] merges CSL with rely-guarantee (RG) 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 and elaborate ways of verifying shared-memory concurrency. For example, TaDa [13] and CaReSL [56] express thread interference protocols through state-transition systems. iCAP [54] and Iris [33] propose a more unified approach by accepting user-defined monoids to express protocols on shared state, together with invariants restricting these protocols. The Iris logic therefore goes by the slogan *"Monoids and invariants are all you need"*[31]. In our technique the invariants take the form of process- and action contracts. Iris provides reasoning support for proving language properties in Coq, where our focus is on proving programs correct.

In the distributed setting, Disel [51] allows to specify protocols for distributed systems. Disel builds dependent type theory and is implemented as a shallow embedding in Coq. Even though this approach is more expressive than ours, it can only semi-automatically be applied in the context of Coq. Villard et al. [61] present a program logic for message passing concurrency, where threads may communicate over channels using native send/receive primitives. This program logic allows to specify protocols via *contracts*, which are state-machines in the style of Session Types [24], 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 [53] 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 rather than usability. Our approach is a balanced trade-off between expressivity and usability. It allows to specify process-algebraic protocols over a general class of concurrent systems, while also allowing the approach to be implemented in automated verification toolsets for concurrency, like VerCors. Related verification toolsets for concurrency are SmallfootRG [9], VeriFast [26], CIVL [52] and Viper [29,39]; 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 formal verification of (multi-threaded) Java and C programs using separation logic. Notably, Penninckx et al. [48] extend VeriFast with Petri-net extensions 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 to reason about more than just I/O behaviour. The CIVL framework can reason about race-freedom and functional correctness of MPI programs written in C [63,36]. The reasoning is done via bounded model checking and symbolic execution.

Apart from the proposed technique, VerCors also allows process-algebraic abstractions to be used as *histories* [5,62]. This is more or less dual to our approach: instead of *reducing* abstract models, the history approach *records* all actions a encountered in $\mathbf{action}\, X.a\, \mathbf{do}\, C$ programs during computation, and thereby builds-up a history process. This process can be analysed to reason about the history of changes in the corresponding concrete shared-memory locations in the program. Our work subsumes this approach, as history recording is only suitable for terminating programs. Our approach performs the reasoning

up-front and allows to specify behavioural patterns of the functional behaviour of non-terminating programs. A locking protocol is a classical example of this, consisting of two states, "locked" and "unlocked", and expressing that clients of the protocol may only transition to "locked" while being in an "unlocked" state, and vice versa. Also related in this respect are the time-stamped histories of Sergey et al. [50] and the more general work on proving linearisability [22,58,34]. These approaches allow to reason about fine-grained concurrency by using sequential verification techniques. Our technique, as well as the history-based technique use process-algebraic linearisation to do so.

Other type theoretical approaches to reason about concurrency and distribution are Session Types [24,25,23]. These approaches typically use process calculi (e.g. the π-calculus) to describe the type of the communication protocol. Behavioural safety of programs is then checked through type checking. Our technique integrates with separation logic and supports reasoning about communication behaviour, as shown by the case study.

10 Conclusion

To reason effectively about realistic concurrent and distributed software, we have presented a verification technique that:

1. Performs the reasoning at a *suitable level of abstraction* that hides irrelevant implementation details;
2. Is *scalable* to realistic programs by being modular and compositional; and
3. Is *practical* by being supported by automated tools.

The approach is a trade-off between expressivity and usability: it is expressive enough to allow reasoning about realistic software, as demonstrated by the case study, and at the same time usable enough to be implemented as part of a deductive program verifier. In contrast, many related program logics mainly aim for expressiveness, while their usage requires substantial manual labour (either because the proof is hand-written, which is also error-prone, or because all steps need to be done interactively within a theorem prover) and therefore hardly scale to realistic programs. For **1.** we use process algebra with data, which offers a mathematically elegant way of expressing program behaviour at the right abstraction level. Process-algebraic specifications of programs can be seen as *models*, over which we can check safety properties, for example via model checking against temporal logic formulas. For **2.** we use a concurrent separation logic that handles process-algebraic models as *resources* that can be split and consumed. This allows to verify in a thread-modular way that programs behave as specified by their abstract models, and allows to project process-algebraic reasoning onto program behaviour. For **3.** the approach has been implemented in VerCors [43] and has been illustrated on various case studies. 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 for reasoning about realistic software.

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. At the moment we are exploring our approach further by applying it on a large industrial case study. We are also investigating the use of mCRL2 and Ivy to reason algorithmically about program abstractions. It would be interesting to investigate to what extend this reasoning can be done *compositionally*, e.g. by using techniques like [10,11]. Moreover, the approach currently only preserves safety properties when connecting programs to their abstract models. We are planning to investigate the preservation of liveness properties. In addition, we are investigating ways to integrate assume-guarantee reasoning more deeply, so that the program logic can get feedback from abstractions not only at the finalisation point, but also at intermediate points in the proof. For the distributed setting, we are planning to implement native support for message passing concurrency and distributed objects into the verification approach. This includes native support for communication and synchronisation on the process-algebraic level.

Acknowledgements. We thank Robbert Krebbers and Sebastiaan Joosten for technical discussions and their comments on earlier drafts of this paper. 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.
2. J.C.M. Baeten. *Process Algebra with Explicit Termination*. Eindhoven University of Technology, Department of Mathematics and Computing Science, 2000.
3. S. Blom, S. Darabi, and M. Huisman. Verification of Loop Parallelisations. In *FASE*, volume 9033 of *LNCS*, pages 202–217. Springer, 2015.
4. S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *iFM*, volume 10510 of *LNCS*, pages 102 – 110. Springer, 2017.
5. S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. History-based Verification of Functional Behaviour of Concurrent Programs. In *SEFM*, volume 9276 of *LNCS*, pages 84 – 98. Springer, 2015.
6. R. Bornat, C. Calcagno, P.W. O'Hearn, and M.J. Parkinson. Permission Accounting in Separation Logic. In *POPL*, pages 259–270, 2005.
7. J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
8. S. Brookes. A semantics for concurrent separation logic. *Theoretical Computer Science*, 375(1–3):227–270, 2007.
9. C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular Safety Checking for Fine-Grained Concurrency. In *International Static Analysis Symposium*, pages 233–248. Springer, 2007.
10. S.C. Cheung and J. Kramer. Checking Safety Properties using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 49–78, 1999.
11. J.M. Cobleigh, D. Giannakopoulou, and C.S Păsăreanu. Learning Assumptions for Compositional Verification. In *TACAS*, pages 331–346. Springer, 2003.
12. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A logic for time and data abstraction. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer, 2014.
13. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, volume 8586 of *LNCS*, pages 207–231. Springer, 2014.
14. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Steps in Modular Specifications for Concurrent Modules. In *MFPS*, EPTCS, pages 3–18, 2015. doi:10.1016/j.entcs.2015.12.002.
15. W. de Roever, K. Engelhardt, and K. Buth. *Data Refinement: Model-oriented Proof Methods and their Comparison*, volume 47. Cambridge University Press, 1998.
16. T. Dinsdale-Young, M. Dodds, P. Gardner, M.J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.
17. R. Dockins, A. Hobor, and A.W. Appel. A Fresh Look at Separation Algebras and Share Accounting. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 161–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
18. X. Feng. Local Rely-Guarantee Reasoning. In *POPL*, volume 44, pages 315–327. ACM, 2009.
19. X. Feng, R. Ferreira, and Z. Shao. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *ESOP*, pages 173–188. Springer, 2007.

20. W. Fokkink and H. Zantema. Basic Process Algebra with Iteration: Completeness of its Equational Axioms. *The Computer Journal*, 37(4):259–267, 1994.
21. J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.
22. M.P. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *TOPLAS*, 12(3):463–492, 1990.
23. K. Honda, E. Marques, F. Martins, N. Ng, V. Vasconcelos, and N. Yoshida. Verification of MPI Programs using Session Types. In *EuroMPI*, volume 7940 of *LNCS*, pages 291–293. Springer, 2012. doi:10.1007/978-3-642-33518-1_37.
24. K. Honda, V.T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP*, pages 122–138. Springer, 1998.
25. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM, 2008.
26. 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 *NFM*, 2011.
27. C.B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, pages 321–332, 1983.
28. S. Joosten, W. Oortwijn, M. Safari, and M. Huisman. An Exercise in Verifying Sequential Programs with VerCors. In A.J. Summers, editor, *20th Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2018.
29. U. Juhasz, I.T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A.J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. Technical report, ETH Zurich, 2014.
30. R. Jung, R. Krebbers, L. Birkedal, and D. Dreyer. Higher-Order Ghost State. In *ICFP*, volume 51, pages 256–269. ACM, 2016.
31. 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 *Principles of Programming Languages (POPL)*, 2015.
32. R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015.
33. R. Krebbers, R. Jung, A. Bizjak, J. Jourdan, D. Dreyer, and L. Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP*, volume 10201 of *LNCS*, pages 696–723. Springer, 2017.
34. S. Krishna, D. Shasha, and T. Wies. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *POPL*, 2017.
35. K.R.M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *FOSAD*, volume 5705 of *LNCS*, pages 195–222, 2009.
36. Z. Luo, M. Zheng, and S.F. Siegel. Verification of MPI programs using CIVL. In *EuroMPI*. ACM, 2017.
37. R. Milner. A Complete Inference System for a Class of Regular Behaviours. *Journal of Computer and System Sciences*, 28(3):439–466, 1984.
38. F. Moller. The Importance of the Left Merge Operator in Process Algebras. In *Automata, Languages and Programming*, pages 752–764, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
39. P. Müller, M. Schwerhoff, and A.J. Summers. Viper - A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*, 2016.
40. A. Nanevski, R. Ley-Wild, I. Sergey, and G.A. Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *European Symposium on Programming (ESOP)*, pages 290–310, 2014.

41. P. W. O'Hearn. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.
42. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and Information Hiding. In *Principles of Programming Languages*, pages 268–280. ACM Press, 2004.
43. W. Oortwijn, S. Blom, D. Gurov, M. Huisman, and M. Zaharieva-Stojanovski. An Abstraction Technique for Describing Concurrent Program Behaviour. In *VSTTE*, volume 10712 of *LNCS*, pages 191 – 209, 2017.
44. W. Oortwijn, S. Blom, and M. Huisman. Future-based Static Analysis of Message Passing Programs. In *PLACES*, pages 65–72, 2016.
45. S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs. *Acta Informatica Journal*, 6:319–340, 1975. doi:10.1007/BF00268134.
46. O. Padon, K.L. McMillan, A. Panda, M. Sagiv, and S. Shoham. Ivy: Safety Verification by Interactive Generalization. In *PLDI*, pages 614–630, 2016. doi:10.1145/2908080.2908118.
47. M.J. Parkinson and A.J. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. In G. Barthe, editor, *ESOP*, volume 6602 of *LNCS*, pages 439–458. Springer, 2011.
48. W. Penninckx, B. Jacobs, and F. Piessens. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *ESOP*, pages 158–182. Springer, 2015.
49. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.
50. I. Sergey, A. Nanevski, and A. Banerjee. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *ESOP*, volume 9032 of *LNCS*, pages 333–358. Springer, 2015.
51. I. Sergey, J.R. Wilcox, and Z. Tatlock. Programming and Proving with Distributed Protocols. *POPL*, 2, 2017.
52. S.F. Siegel, M. Zheng, Z. Luo, T.K. Zirkel, A.V. Marianiello, J.G. Edenhofner, M.B. Dwyer, and M.S. Rogers. CIVL: The Concurrency Intermediate Verification Language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 61. ACM, 2015.
53. A.J. Summers and P. Müller. Actor Services – Modular Verification of Message Passing Programs. In *ESOP*, pages 699–726. Springer, 2016.
54. K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014.
55. K. Svendsen, L. Birkedal, and M. Parkinson. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP*, pages 169–188. Springer, 2013.
56. 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, 2013.
57. Y.S. Usenko. *Linearization in μCRL*. Technische Universiteit Eindhoven, 2002.
58. V. Vafeiadis. Automatically Proving Linearizability. In *CAV*, pages 450–464, 2010.
59. V. Vafeiadis. Concurrent separation logic and operational semantics. In *MFPS*, volume 276 of *ENTCS*, pages 335 – 351, 2011.
60. V. Vafeiadis and M.J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.
61. J. Villard, É. Lozes, and C. Calcagno. Proving Copyless Message Passing. In *Asian Symposium on Programming Languages and Systems*, pages 194–209. Springer, 2009.

62. M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying Functional Behaviour of Concurrent Programs*. PhD thesis, University of Twente, 2015. doi:10.3990/1.9789036539241.
63. M. Zheng, M.S. Rogers, Z. Luo, M.B. Dwyer, and S.F. Siegel. CIVL: Formal Verification of Parallel Programs. In *ASE*, pages 830–835, 2015. doi:10.1109/ASE.2015.99.