applied sciences

MDPI

Article

An Abstraction Technique for Verifying Shared-Memory Concurrency †

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

1 Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland
2 Department of Theoretical Computer Science, KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden; dilian@kth.se
3 Formal Methods and Tools, University of Twente, 7500 AE Enschede, The Netherlands; m.huisman@utwente.nl
* Correspondence: wytse.oortwijn@inf.ethz.ch
† This paper is an extended version of our paper published in 21st International Conference on Verification, Model Checking, and Abstract Interpretation held in New Orleans, LA, USA, 19–21 January 2020.

Received: 30 April 2020; Accepted: 2 June 2020; Published: 5 June 2020

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

Keywords: concurrency verification; program logics; process algebra; code verification; abstraction

1. Introduction

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

Even though verification of concurrent and distributed software is a very active research field [1–6], most work in this line of research is essentially theoretical, and tends to focus primarily on contributing expressive program logics specialised in reasoning about advanced concurrency features like relaxed or weak memory, fine-grained concurrency, message passing interaction, etc. Even though expressive, it is very challenging for these logics to be integrated into SMT-based automated verifiers like for example VeriFast [7], VerCors [8] and Viper [9,10]. Instead, most of these works have to be applied in

pen-and-paper style, or at best semi-automatically in the context of an interactive theorem prover like Coq [11,12] or Isabelle/HOL [13].

This article contributes a concurrency verification technique that applies directly on the level of program code and is supported by automated verifiers. However, rather than doing the verification fully on the level of program code, our approach allows *soundly* abstracting program behaviour into *abstract models* which can be reasoned about externally, on a higher level in which irrelevant implementation details are hidden, to (indirectly) prove properties about the program behaviour. The presented verification technique (1) has been implemented in VerCors—an automated SMT-based concurrency verifier; (2) is demonstrated on various (real-world) examples, including a leader election protocol (presented in Section 5); and (3) the metatheory of the technique has been fully formalised and proven sound with the Coq proof assistant. With respect to (2); apart from the examples given in this article, more examples of our approach are given in [14], including the verification of a (reentrant) lock as well as a concurrent parallel GCD algorithm. Our technique has also been used in a real-world industrial case study [15]—on the formal verification of a safety-critical traffic tunnel control system.

This article extends our earlier VMCAI'20 article [16]. Elaborating on the contributions with respect to this earlier article; we contribute a generalisation of the theory in [16] by combining it with the techniques proposed in [17] and [18] into a single logical framework that is more general than the original. This combined unified framework is proven sound with Coq and is available online at [19].

1.1. Motivation

Reasoning about complex concurrent program behaviours is only practical if conducted at a suitable level of abstraction that hides implementation details that are irrelevant for the properties to prove. Furthermore, any real concurrent programming language with shared memory, threads and locks, has only very little algebraic behaviour. In contrast, *process algebras* offer an abstract, mathematically elegant way of expressing program behaviour. Process algebras have been used widely in the past for modelling and analysing the behaviour of concurrent programs at an adequate level of abstraction [20,21]. Our approach therefore uses a process algebra as a language for *specifying* program behaviour. Such a specification can be seen as a model, the properties of which can additionally be checked (say by interactive theorem proving, or by model checking against temporal logic formulas, which can be seen as even more abstract behavioural specifications). The main difficulty of this approach is dealing with the typical abstraction gap between program implementations and their abstract 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 [22]; we leave the preservation of liveness properties for future work.

The key idea of the approach rests in the use of concurrent separation logic (CSL) to reason not only about data races and memory safety, which is standard [23,24], but also about process-algebraic models (that is, 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 (mechanically proven) correctness results stating that any verified program is a *refinement* of its abstract, process-algebraic model.

Process-algebraic models are composed out of individual *actions* that abstract atomic behaviours of program components. Our approach allows specifying program components to follow a particular sequence/pattern of actions—a protocol. One can then reason about the interaction behaviour of different program components by reasoning about the composition of their models, for example by using a model checker for process algebra, like mCRL2 [25]. This approach of specifying the interactions of program components is different from classical Hoare logic, which is purely transformational in the sense that it considers verified (terminating) program components essentially as transformers from states satisfying the specified precondition to states satisfying the specified postcondition.

A benefit of our combined approach compared to model checking is that it allows reasoning soundly about both data and control-oriented properties in a single framework. Model checkers

typically specialise in reasoning about temporal, control-oriented specifications (e.g., send actions must always be matched by a recv), and generally have limited support for handling data due to the risk of state-space explosions. Hoare logic based techniques, on the other hand, tend to specialise in reasoning about data specifications (e.g., a sorting function should yield a sorted permutation of its input), and are typically limited in their capabilities to reason about control-flow properties. Since realistic concurrent systems often deal with both data and control-flow, it is beneficial to be able to reason about both in a single framework. Additionally, our technique addresses the typical "abstraction gap" problem of model checking: is the model actually a faithful abstraction of the modelled system? We propose techniques to formally link programs to their abstract models, allowing one to prove that all program behaviours that should be captured by the abstract model are indeed soundly abstracted.

1.2. Contributions

The main contributions of this extended article are:

- A verification technique to reason about the behaviour of shared-memory concurrent programs that is modular, compositional, and proven sound. This article extends [16] by generalising its verification technique and combining it with the core ideas of [17,18]. In particular, it extends the process algebra specification language with summations, support for input parameters, and the assertional processes of [17], which shall all be introduced later, in Section 3.
- A full Coq development of the formalisation as presented in Section 3, together with a soundness proof of the approach. The Coq sources and their documentation are available at [19].
- Several examples that demonstrate this new (unified) verification approach, including a leader election protocol case study discussed in Section 5.

1.3. Outline

The remainder of this article is organised as follows. First Section 2 illustrates our technique on a small Owicki–Gries example program, before Section 3 gives theoretical justification of the verification technique. In particular, Section 3.1 introduces the process algebra specification language, after which Section 3.2 introduces the programming language on which the approach is formalised on. Section 3.3 defines and discusses the syntax and semantics of the assertion language, which is a concurrent separation logic with special constructs to to handle process-algebraic models. Section 3.4 discusses the proof system and Section 3.4 its soundness. Section 4 gives details on how the verification technique is implemented in the concurrency verified VerCors, and briefly elaborates on the Coq development. Section 5 demonstrates the approach on a larger case study: the verification of a classical leader election protocol. Finally, Section 6 discusses related work and Section 7 concludes.

2. Approach

Before going into the formal details of the approach, let us first illustrate it on a simple example. Our approach allows abstractly specifying concurrent program behaviour as process-algebraic models. Processes are composed of atomic, indivisible *actions*. In our approach the actions are *logical descriptions of shared-memory modifications*: they describe what changes the program is allowed to make to a specified region of shared memory—the program heap. These actions are then *linked* to the concrete instructions in the program code that perform the memory updates. These links between program components and their abstract models are established deductively, using a concurrent separation logic that is presented later. Well-known techniques for process-algebraic reasoning can then be applied to guarantee safety properties over all possible state changes, as described by their compositions of actions. The novelty of the approach is that these safety properties can then be *relied upon* in the program logic due to the established formal connection between program components and their process-algebraic models.

2.1. Example Program

Consider the following example program, which is a simple variant of the classical concurrent Owicki–Gries program [26].

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

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

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

Well-known existing classical approaches and techniques to deal with such concurrent programs [27] include auxiliary state [26] and interference abstraction via rely-guarantee reasoning [28]. Modern program logics employ more intricate constructs, like atomic Hoare triples [5] in the context of TaDa, or higher-order ghost state [29] in the context of Iris. However, the mentioned classical approaches typically do not scale well, whereas such modern, theoretical approaches are hard to integrate into (semi-)automated SMT-based verifiers like for example VeriFast or VerCors.

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

The approach consists of the following three steps:

Step 1. Define a process-algebraic model $\mathsf{OG} = (\mathsf{incr}(4) \parallel \mathsf{mult}(4)) \cdot ?(b_{post})$ that is composed out of two actions, incr and mult, that abstract the two atomic sub-programs;

Step 2. Verify that the OG process indeed satisfies the Owicki–Gries postcondition, b_{post}; and

Step 3. Deductively verify that OG is a correct behavioural specification of the program's execution flow. That is, verify that every atomic state change that is executed by a run of the program has a corresponding action in OG.

The following paragraphs give more detail on these three steps.

2.1.1. Step 1: Specifying Program Behaviour

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

```
guard true;                          guard true;
effect x = \old(x) + n;              effect x = \old(x) * n;
action incr(int n);                  action mult(int n);
```

Any action contract consists of a *guard* and an *effect*. The guard of any action specifies the condition under which the action is allowed to be executed. In the above example, the guard of both incr and mult is specified to be true, meaning that both these actions may unconditionally be performed. The *effect* clause of any action specifies the way the action is allowed to change the (program) state. Observe that incr and mult are indeed abstractions of the two atomic sub-programs, and that the effect clauses of these actions are abstract specifications of how the program updates the heap. (Note that one could think of guards and effect of actions as pre- and postconditions, respectively. However, they are not strictly the same (hence the slightly different terminology). For the sake of process-algebraic analysis all action contracts can be assumed to hold, while on the program level one has to prove that sets of

instructions that correspond to the action satisfy the action contract, as will be explained in a moment.) Note that both these abstract specifications contain a free variable x, which is a *process-algebraic variable* that is later linked to a concrete heap location in the program (this will be $[E]$). Moreover, the increment and multiplication of 4 has now been generalised to an arbitrary integer n.

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

```
requires true;
process OG(int n) := (incr(n) || mult(n)) · ?(x = (\old(x) + n) * n ∨ x = \old(x) * n + n);
```

Notice that the OG process has the form $(\mathsf{incr}(n) \parallel \mathsf{mult}(n)) \cdot ?(b_{post})$ with b_{post} the Owicki–Gries postcondition. Here $\cdot$ denotes sequential composition, and $?(b_{post})$ is an *assertion process*. These assertions are the main subject of process-algebraic reasoning: we verify that all asserted properties are never violated. Here we specify that $?(b_{post})$ holds after executing $\mathsf{incr}(n)$ and $\mathsf{mult}(n)$ in any order.

The OG process also has a precondition that could potentially impose restrictions on the values of n. But for this Owicki–Gries example we do not have any such restrictions. Note that postconditions (that is, `ensures` clauses) are encoded as assertional processes, like done above.

2.1.2. Step 2: Process-Algebraic Reasoning

The next step is to verify that OG satisfies all properties b that are encoded as assertions $?(b)$, which can be reduced to standard process-algebraic analysis. Intuitively we say that OG is *verified* if, starting from any state satisfying OG's `requires` clause, the process can never reach an asserted property b that does not hold. We shall later give a more formal definition of what it means for a process to be verified with respect to its precondition, in Section 3.4.2.

The standard approach to analysing OG would be to first linearise it to the bisimilar process $\mathsf{incr}(n) \cdot \mathsf{mult}(n) \cdot ?(b_{post}) + \mathsf{mult}(n) \cdot \mathsf{incr}(n) \cdot ?(b_{post})$, where $+$ denotes non-deterministic choice and with b_{post} again the Owicki–Gries postcondition, and then to reason about all branches of this linearised process. With "reasoning about all branches" we intuitively mean establishing that all assertions encountered during any execution of a process are a logical consequence of the series of `effects` preceding the assertion. A formal definition is provided later in Section 3.1. VerCors currently does the analysis by encoding the linearised process as input to the Viper verifier [10]. VerCors can indeed automatically prove that OG satisfies the asserted property.

2.1.3. Step 3: Deductively Linking Processes to Programs

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

Figure 1 shows the required program annotations. First, x is connected to $[E]$ by initialising a new model M on line 2 that executes according to $\mathsf{OG}(4)$. The actions incr and mult are then linked to the corresponding sub-programs on lines 5–7 and 11–13 by identifying *action blocks* in the code, using special program annotations. We use these `action` annotations to verify in a thread-modular way that the left thread performs the $\mathsf{incr}(4)$ action (on lines 5–7) and that the right thread performs $\mathsf{mult}(4)$ (lines 11–13). As a result, when the program reaches the `query` annotation on line 15, only the $?(b_{post})$ process is left on the process level—the $\mathsf{incr}(4) \parallel \mathsf{mult}(4)$ part has already been executed alongside the program. Since the Owicki–Gries postcondition b_{post} is already proven externally, by other means, in the previous step, the program logic may *rely on its validity*. But since we tracked the contents at heap location $[E]$ on the process level as the variable x, one may indirectly conclude that the heap at location $[E]$ has evolved as described by OG. In other words, using program annotations we prove that the program is a *refinement* of OG, meaning that we get the asserted property in the logic, on line 17.

Finally, the `finish` annotation on line 16 indicates that the model has been fully reduced at that point, and thus may be disposed of. This is for technical reasons; the program logic will do some bookkeeping while dealing with process-algebraic abstractions, and `finish` will cause this bookkeeping to be cleaned up. This is later discussed in greater detail, in Section 3.4.2.

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

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

3. Formalisation

We now give theoretical justification of the verification approach and explains the underlying logical machinery. First, Sections 3.1 and 3.2 briefly discuss the syntax and semantics of process algebraic models and programs, respectively. Then Section 3.3 presents the program logic as a concurrent separation logic with assertions that allow to specify program behaviour as a process algebraic model. Section 3.4 formally introduces and discusses the proof rules. Finally, Section 3.5 discusses soundness of the approach. All these components have been fully formalised in Coq, including the soundness proof of the logic. Section 4 elaborates on the Coq development of the meta-theory, as well as on tool support, developed for the VerCors concurrency verifier.

3.1. Process-Algebraic Models

Program abstractions are defined using the following ACP-style [30] process-algebraic specification language, where $x, y, z, \cdots \in \mathit{ProcVar}$ are *process-algebraic variables*; $v, w, \cdots \in \mathit{Val}$ are *values* from an infinite domain *Val*; and $a, \cdots \in \mathit{Act}$ are *(process-algebraic) actions*.

Definition 1 (Processes)**.**

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

Clarifying the different connectives and constructs, ε is the *empty process*, which has no behaviour. The δ process is the *deadlocked process* which neither progresses nor terminates. Processes of the form $a(e)$ are *actions*, which model the basic, observable (shared-memory) system behaviours. Actions are parameterised by data, in the form of expressions e. The process $P \cdot Q$ is the *sequential composition* of P and Q, whereas $P + Q$ is their non-deterministic *choice*. The *parallel composition* of processes P and Q is written $P \parallel Q$. 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. The left-merge is an auxiliary connective commonly used to axiomatise parallel composition [31], by having $P \parallel Q = P \between Q + Q \between P$. The process $\Sigma_x P$ is the infinite summation $P[x/v_0] + P[x/v_1] + \cdots$ over all values $v_0, v_1, \ldots \in \mathit{Val}$. Any summation $\Sigma_x P$ is a *binder* for the summation variable x. In the remainder we assume without loss of generality that all variables bound by summation are unique (since any such

variables can be renamed to unique ones if this is not yet the case). Sometimes $\Sigma_{x_0,\ldots,x_n} P$ is written to abbreviate $\Sigma_{x_0} \cdots \Sigma_{x_n} P$. The conditional (guarded) process $b : P$ behaves as P if the Boolean condition b holds, and otherwise behaves as δ. Finally, P^* is the *repetition*, or *iteration* of P, and denotes a sequence of zero or more P's. The infinite iteration of P is derived to be $P^\omega \triangleq P^* \cdot \delta$. Finally, $?(b)$ is the *assertive process*, which is very similar to guarded processes: $?(b)$ is behaviourally equivalent to δ in case b does not hold. However, assertive processes have a special role in our approach: they are the main subject of process-algebraic analysis, as they encode the properties b to verify, as logical assertions. Moreover, they are a key component in connecting process-algebraic reasoning with deductive reasoning, as their properties can be relied upon in the deductive proofs of programs via the `query` b ghost command.

3.1.1. Action Contracts

The presented verification approach uses processes in the presence of data, which is implemented via *action contracts*. Action contracts consist of pre- and postconditions which we refer to as *guards* and *effects*, respectively, that logically describe the state changes that are imposed by the corresponding action. In the remainder of this article, each action is assumed to have an action contract assigned to it. Instead of defining syntax for writing these contracts, the following two functions are assumed for obtaining the pre- and postcondition of an action (from *Act*) and its data parameter (from *ProcExpr*), respectively.

$$\mathsf{guard} : \mathit{Act} \to \mathit{ProcExpr} \to \mathit{ProcCond} \qquad \mathsf{effect} : \mathit{Act} \to \mathit{ProcExpr} \to \mathit{ProcCond}$$

Both these conditions are of type *ProcCond*, which is the domain of Boolean expressions over process-algebraic variables. Note that, since actions are parameterised by data (see Definition 1), both guard and effect take a second argument to account for the input parameter, which is of type *ProcExpr*—the type of arithmetic expressions over process-algebraic variables.

Here $\mathit{Act} \to \mathit{ProcExpr} \to \mathit{ProcCond}$ should be read as $\mathit{Act} \to (\mathit{ProcExpr} \to \mathit{ProcCond})$ and interpreted as a function sequence (in the sense of currying). That is, it is the set of functions mapping *Act* to the set of functions mapping *ProcExpr* to *ProcCond*.

3.1.2. Free Variables and Substitution

A function $\mathsf{fv_e} : \mathit{ProcExpr} \to 2^{\mathit{ProcVar}}$ is used to determine the set of free process-algebraic variables in expressions as usual, and likewise for $\mathsf{fv_b}(b)$ and $\mathsf{fv_P}(P)$ for Boolean expressions b and processes P. We often omit the subscripts and simply write $\mathsf{fv}(\cdot)$ whenever the context allows it. The definitions of $\mathsf{fv_e}$, $\mathsf{fv_b}$ and $\mathsf{fv_P}$ are mostly standard and thus deferred to [19]. Noteworthy however are:

$$\mathsf{fv_P}(a(e)) \triangleq \mathsf{fv_b}(\mathsf{guard}\ a\ e) \cup \mathsf{fv_b}(\mathsf{effect}\ a\ e) \qquad \mathsf{fv_P}(\Sigma_x\ P) \triangleq \mathsf{fv_P}(P) \setminus \{x\} \qquad \mathsf{fv_P}(?(b)) \triangleq \mathsf{fv_b}(b)$$

Substitution is written $e'[x/e]$ (and likewise for Boolean expressions and processes) and has a standard definition: replacing any occurrence of x inside e' by the expression e. Noteworthy is that substitutions inside action processes $a(e)$ do not affect the action contracts: $a(e')[x/e] \triangleq a(e'[x/e])$.

3.1.3. Operational Semantics

The denotational semantics of process-algebraic expressions $[\![\cdot]\!]_\mathsf{e} : \mathit{ProcExpr} \to \mathit{ProcStore} \to \mathit{Val}$ and conditions $[\![\cdot]\!]_\mathsf{b} : \mathit{ProcCond} \to \mathit{ProcStore} \to \mathit{Bool}$ is defined in the standard way, as total functions that evaluate to *Val* and *Bool*, resp. The set $\sigma \in \mathit{ProcStore} \triangleq \mathit{ProcVar} \to \mathit{Val}$ is the domain of *process stores*, which are used to give an interpretation to all process-algebraic variables. The overloaded notations $[\![e]\!]\sigma$ and $[\![b]\!]\sigma$ are used instead of $[\![e]\!]_\mathsf{e}\sigma$ and $[\![b]\!]_\mathsf{b}\sigma$ wherever the context allows it. Moreover, $[\![e]\!]$ is sometimes written instead of $[\![e]\!]_\mathsf{e}\sigma$ when e is closed (i.e., when $\mathsf{fv}(e) = \emptyset$), and likewise for $[\![b]\!]$.

The operational semantics of the process algebra language is expressed as a labelled binary small-step reduction relation $\xrightarrow{\alpha} \subseteq \mathit{ProcConf} \times \mathit{ProcLabel} \times \mathit{ProcConf}$ over *process configurations*,

defined as $ProcConf \triangleq Proc \times ProcStore$—pairs of processes and process stores. The labels α of the reduction rules are defined as follows: $\alpha \in ProcLabel ::= a(v) \mid \mathit{assn}$. Transitions labelled $a(v)$ are reductions of actions, whereas assn indicates reductions of assertions.

Before giving the reduction rules we first define a notion of *successful termination* $P\downarrow$ of processes P. Successful termination is only defined for processes that are *well-formed*. Any process P is defined to be *well-formed* if any action parameters (the e's in $a(e)$) and conditions (the b's in $b : Q$) occurring inside P are closed.

Definition 2 (Successful termination)**.**

$$\downarrow\text{-EPSILON}\quad \varepsilon\downarrow \qquad \downarrow\text{-SEQ}\quad \frac{P\downarrow \qquad Q\downarrow}{P\cdot Q\downarrow} \qquad \downarrow\text{-ALT-L}\quad \frac{P\downarrow}{P+Q\downarrow} \qquad \downarrow\text{-ALT-R}\quad \frac{Q\downarrow}{P+Q\downarrow} \qquad \downarrow\text{-PAR}\quad \frac{P\downarrow \qquad Q\downarrow}{P \parallel Q\downarrow} \qquad \downarrow\text{-MERGE}\quad \frac{P\downarrow \qquad Q\downarrow}{P \between Q\downarrow}$$

$$\downarrow\text{-SUM}\quad \frac{P[x/v]\downarrow}{\Sigma_x P\downarrow} \qquad \downarrow\text{-COND}\quad \frac{[\![b]\!] \qquad P\downarrow}{b : P\downarrow} \qquad \downarrow\text{-ITER}\quad P^*\downarrow$$

Intuitively, any process P can terminate successfully if P has the choice to have no further behaviour. This means that ε can always successfully terminate ($\downarrow$-EPSILON), as it has no behaviour, while δ can never successfully terminate. Iteration P^* can always successfully terminate ($\downarrow$-ITER) as it may choose not to start iterating and thereby to behave as ε.

The small-step reduction rules of process configurations are given below. Likewise to the definition of successful termination, also these reduction rules require processes to be *well-formed*.

Definition 3 (Reductions of process configurations)**.**

$$\longrightarrow\text{-ACT}\quad \frac{[\![\text{guard } a\ e]\!]\sigma \qquad [\![\text{effect } a\ e]\!]\sigma'}{(a(e),\sigma) \xrightarrow{a([\![e]\!])} (\varepsilon,\sigma')} \qquad \longrightarrow\text{-ASSN}\quad \frac{[\![b]\!]\sigma}{(?(b),\sigma) \xrightarrow{\mathit{assn}} (\varepsilon,\sigma)} \qquad \longrightarrow\text{-SEQ-L}\quad \frac{(P,\sigma) \xrightarrow{\alpha} (P',\sigma')}{(P\cdot Q,\sigma) \xrightarrow{\alpha} (P'\cdot Q,\sigma')}$$

$$\longrightarrow\text{-SEQ-R}\quad \frac{P\downarrow \qquad (Q,\sigma) \xrightarrow{\alpha} (Q',\sigma')}{(P\cdot Q,\sigma) \xrightarrow{\alpha} (Q',\sigma')} \qquad \longrightarrow\text{-ALT-L}\quad \frac{(P,\sigma) \xrightarrow{\alpha} (P',\sigma')}{(P+Q,\sigma) \xrightarrow{\alpha} (P',\sigma')} \qquad \longrightarrow\text{-ALT-R}\quad \frac{(Q,\sigma) \xrightarrow{\alpha} (Q',\sigma')}{(P+Q,\sigma) \xrightarrow{\alpha} (Q',\sigma')}$$

$$\longrightarrow\text{-PAR-L}\quad \frac{(P,\sigma) \xrightarrow{\alpha} (P',\sigma')}{(P \parallel Q,\sigma) \xrightarrow{\alpha} (P' \parallel Q,\sigma')} \qquad \longrightarrow\text{-PAR-R}\quad \frac{(Q,\sigma) \xrightarrow{\alpha} (Q',\sigma')}{(P \parallel Q,\sigma) \xrightarrow{\alpha} (P \parallel Q',\sigma')} \qquad \longrightarrow\text{-LMERGE}\quad \frac{(P,\sigma) \xrightarrow{\alpha} (P',\sigma')}{(P \between Q,\sigma) \xrightarrow{\alpha} (P' \parallel Q,\sigma')}$$

$$\longrightarrow\text{-SUM}\quad \frac{(P[x/v],\sigma) \xrightarrow{\alpha} (P',\sigma')}{(\Sigma_x P,\sigma) \xrightarrow{\alpha} (P',\sigma')} \qquad \longrightarrow\text{-COND}\quad \frac{[\![b]\!] \qquad (P,\sigma) \xrightarrow{\alpha} (P',\sigma')}{(b : P,\sigma) \xrightarrow{\alpha} (P',\sigma')} \qquad \longrightarrow\text{-ITER}\quad \frac{(P,\sigma) \xrightarrow{\alpha} (P',\sigma')}{(P^*,\sigma) \xrightarrow{\alpha} (P'\cdot P^*,\sigma')}$$

Most of the reduction rules are standard in spirit [32]. However, the handling of actions and their contracts make this process algebra language non-standard. More specifically, the non-standard $\longrightarrow$-ACT reduction rule for action handling permits the state σ to change in any way, as long as these changes comply with the action contract. We will later use the $\longrightarrow$-ACT rule to connect shared-memory updates in programs, to action contract-complying state changes on the process level.

Moreover, the notion of successful termination is used to define the reduction rule for sequential composition, $\longrightarrow$-SEQ-R, which is standard in process algebra languages with ε [33]. (An alternative on the explicit use of successful termination is to introduce internal (τ-)transitions for the reductions

of ε. However, this might make the remaining formalisation less elegant, for example by requiring a notion of weak bisimilarity, instead of the notion of strong bisimilarity that is introduced later in this section.)

3.1.4. Process-Algebraic Verification

Process-algebraic verification in our approach amounts to verifying that all reachable assertional processes $?(b)$ are always satisfied, which we are interested in so that the program logic can *rely* on the b's. Any process configuration (P, σ) fails to verify, or *exhibits a fault*, which we write $\lightning(P, \sigma)$, if it can directly violate an assertion. Verifying a process, i.e., checking for fault absence, could for example be reduced to checking the μ-calculus formula $[\mathsf{true}^* \cdot \lightning]\mathsf{false}$, e.g., using the mCRL2 model checker, where $\lightning$ is modelled as an explicit fault state, meaning "no faults are every reachable".

Fault exhibition is defined inductively as follows.

Definition 4 (Faulting process configuration)**.**

$$\frac{}{\lightning(?(b), \sigma)}\ \lightning\text{-ASSN} \quad \neg[\![b]\!]\sigma$$

$$\frac{\lightning(P, \sigma)}{\lightning(P \cdot Q, \sigma)}\ \lightning\text{-SEQ-L}$$

$$\frac{P\downarrow \qquad \lightning(Q, \sigma)}{\lightning(P \cdot Q, \sigma)}\ \lightning\text{-SEQ-R}$$

$$\frac{\lightning(P, \sigma)}{\lightning(P + Q, \sigma)}\ \lightning\text{-ALT-L}$$

$$\frac{\lightning(Q, \sigma)}{\lightning(P + Q, \sigma)}\ \lightning\text{-ALT-R}$$

$$\frac{\lightning(P, \sigma)}{\lightning(P \parallel Q, \sigma)}\ \lightning\text{-PAR-L}$$

$$\frac{\lightning(Q, \sigma)}{\lightning(P \parallel Q, \sigma)}\ \lightning\text{-PAR-R}$$

$$\frac{\lightning(P, \sigma)}{\lightning(P \lfloor\!\lfloor Q, \sigma)}\ \lightning\text{-LMERGE}$$

$$\frac{\lightning(P[x/v], \sigma)}{\lightning(\Sigma_x P, \sigma)}\ \lightning\text{-SUM}$$

$$\frac{[\![b]\!]\sigma \qquad \lightning(P, \sigma)}{\lightning(b : P, \sigma)}\ \lightning\text{-COND}$$

$$\frac{\lightning(P, \sigma)}{\lightning(P^*, \sigma)}\ \lightning\text{-ITER}$$

Any process configuration (P, σ) is defined to be *safe*, denoted as $\checkmark(P, \sigma)$, if it can never reach a faulting configuration. More formally:

Definition 5 (Safe process configurations)**.** *The $\checkmark \subset ProcConf$ predicate is coinductively defined such that, whenever $\checkmark(P, \sigma)$ holds, then* (1) *$\neg\lightning(P, \sigma)$; and* (2) *for any P', σ' and α, if $(P, \sigma) \xrightarrow{\alpha} (P', \sigma')$, then $\checkmark(P, \sigma)$.*

Definition 6 (Verified processes)**.** *Any well-formed process P is defined to be* verified with respect to a (pre)condition *b, which is written $\models \{b\}\ P$, if $\forall\sigma\,.\,[\![b]\!]\sigma \implies \checkmark(P, \sigma)$.*

3.1.5. Bisimulation

Our verification approach allows handling process-algebraic models up to (strong) bisimulation.

Definition 7 (Bisimulation)**.** *Any binary relation $\mathscr{R} \subseteq Proc \times Proc$ over processes is defined to be a* bisimulation relation *if, whenever $P\,\mathscr{R}\,Q$, then:*

(1) $P\downarrow$ if and only if $Q\downarrow$.
(2) $\lightning(P, \sigma)$ if and only if $\lightning(Q, \sigma)$, for any σ.
(3) For any σ, P', σ' and α, if $(P, \sigma) \xrightarrow{\alpha} (P', \sigma')$, then there exists a Q' such that $(Q, \sigma) \xrightarrow{\alpha} (Q', \sigma')$ and $P'\,\mathscr{R}\,Q'$.
(4) For any σ, Q', σ' and α, if $(Q, \sigma) \xrightarrow{\alpha} (Q', \sigma')$, then there exists a P' such that $(P, \sigma) \xrightarrow{\alpha} (P', \sigma')$ and $P'\,\mathscr{R}\,Q'$.

Any two processes P and 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 $P\,\mathscr{R}\,Q$. Bisimilarity expresses that both processes exhibit the same behaviour, in the sense that their action sequences describe the same state changes. Any bisimulation relation constitutes an equivalence relation. Furthermore, bisimilarity is a congruence for all process algebraic connectives.

Successful termination $P\downarrow$ can intuitively be understood as P being bisimilar to the process $\varepsilon + P$, that is, by having the choice to have no further behaviour.

Proposition 1. *If $P \downarrow$ then $P \cong \varepsilon + P$.*

Lemma 1. *If $P \cong Q$ and $\checkmark(P, \sigma)$, then $\checkmark(Q, \sigma)$.*

Figure 2 gives a list of bisimulation equivalences that hold for our process algebra language. Note that the left-merge connective $\between$ is not strictly needed, in the sense that our approach does not rely on it, but can be used to prove for example that $a(e) \parallel a'(e')$ is bisimilar to $a(e) \cdot a'(e') + a'(e') \cdot a(e)$.

Sequential connectives

A1 $P + Q \cong Q + P$ A2 $P + (Q + R) \cong (P + Q) + R$ A3 $P + P \cong P$ A4 $(P + Q) \cdot R \cong P \cdot R + Q \cdot R$

A5 $P \cdot (Q \cdot R) \cong (P \cdot Q) \cdot R$ A6 $P + \delta \cong P$ A7 $\delta \cdot P \cong \delta$ A8 $P \cdot \varepsilon \cong P$ A9 $\varepsilon \cdot P \cong P$ COND1 $\mathsf{true} : P \cong P$

COND2 $\mathsf{false} : P \cong \delta$ COND3 $b_1 : b_2 : P \cong b_1 \wedge b_2 : P$ KLEENE1 $P^* \cong P \cdot P^* + \varepsilon$ KLEENE2 $\delta^* \cong \varepsilon$ KLEENE3 $\varepsilon^* \cong \varepsilon$ KLEENE4 $P^{**} \cong P^*$

KLEENE5 $P^* \cdot P^* \cong P^*$ KLEENE6 $(P + Q)^* \cong P^* \cdot (Q \cdot P^*)^*$ KLEENE7 $P^\omega \cong P \cdot P^\omega$ SUM1 $\Sigma_x P \cong P[x/v] + \Sigma_x P$

SUM2 $\Sigma_x (P + Q) \cong \Sigma_x P + \Sigma_x Q$ SUM3 $\dfrac{x \notin \mathsf{fv}(Q)}{(\Sigma_x P) \cdot Q \cong \Sigma_x (P \cdot Q)}$ SUM4 $\dfrac{x \notin \mathsf{fv}(b)}{\Sigma_x b : P \cong b : \Sigma_x P}$ SUM5 $\dfrac{x \notin \mathsf{fv}(P)}{\Sigma_x P \cong P}$

Parallel connectives

PAR1 $P \parallel Q \cong Q \parallel P$ PAR2 $P \parallel (Q \parallel R) \cong (P \parallel Q) \parallel R$ PAR3 $P \parallel Q \cong P \between Q + Q \between P$ PAR4 $\varepsilon \parallel P \cong P$

PAR5 $P \parallel \delta \cong P \cdot \delta$ LMERGE1 $\delta \between P \cong \delta$ LMERGE2 $\varepsilon \between \delta \cong \delta$ LMERGE3 $\varepsilon \between (a \cdot P) \cong \delta$ LMERGE4 $(a \cdot P) \between Q \cong a \cdot (P \parallel Q)$ LMERGE5 $\varepsilon \between \varepsilon \cong \varepsilon$

LMERGE6 $\varepsilon \between (P + Q) \cong \varepsilon \between P + \varepsilon \between Q$ LMERGE7 $(P + Q) \between R \cong P \between R + Q \between R$ LMERGE8 $(P \between Q) \between R \cong P \between (Q \parallel R)$

LMERGE9 $P \between \delta \cong P \cdot \delta$

Figure 2. Standard bisimulation equivalences of the process algebra language.

3.2. Programs

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

Definition 8 (Expressions, conditions, conditions, commands)**.**

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

This language is a variation of the language proposed by O'Hearn [24] and Brookes [23]. In particular, we extend their language with *specification-only* commands (code annotations) for handling process-algebraic models. These commands are coloured blue. Note that the blue colourings do not have any semantic meaning; they only indicate which language constructs are specification-only. Moreover, we interchangeably refer to commands also as programs.

3.2.1. Standard Language Constructs

The notation $[E]$ stands for *heap dereferencing*, where E is an expression whose evaluation determines the heap location to dereference. The commands $X := [E]$ and $[E] := E'$ denote heap reading and writing: they read from, and write to, the heap at location E, respectively. Moreover, $X := \texttt{alloc}\ E$ allocates a free heap location and writes the value represented by E to it, whereas $\texttt{dispose}\ E$ deallocates the heap location at E.

Regarding concurrency, the command $C_1 \parallel C_2$ is the statically-scoped parallel composition of C_1 and C_2 and expresses their concurrent execution. In the sequel, we sometimes refer to commands that are put in parallel as different *threads*; for example C_1 and C_2 in the above. Moreover, $\texttt{atomic}\ C$ expresses a statically-scoped lock: it represents the atomic execution of C, that is, without interference of other threads. The command $\texttt{inatom}\ C$ represents *partially executed* atomic programs: ones that are currently being executed, where C is the remaining program that still has to be executed atomically. Such commands are sometimes referred to as "runtime syntax", as they are not written by users of the language, but are instead an artefact of program execution.

3.2.2. Specification-Only Constructs

The instructions that are displayed in blue are the specification-only language constructs, for handling process-algebraic models in the logic. These instructions are ignored during regular program execution and are essentially handled as if they were code comments.

Specification-wise, $X := \texttt{process}\ (\lambda x.P)(E)\ \texttt{over}\ \Pi$ initialises a new process-algebraic model P in the proof system that takes a single input argument named x, namely (the evaluation of) the expression E. This model is used (1) as a specification of how a particular region of shared memory, specified by Π, is allowed to evolve over time; and (2) to support reasoning over the model to indirectly prove properties of how the heap evolves. The Π component is an *abstraction binder*, which is also defined in Definition 8 and is used to connect process-algebraic variables to heap locations in the program. In particular, the abstraction binders make the connections/links between process-algebraic state and shared-memory program state (that is, heap locations). In the sequel, we often use abstraction binders as if they were finite partial mappings, $\Pi : \mathit{ProcVar} \rightharpoonup_{\mathsf{fin}} \mathit{Expr}$, from process-algebraic variables to the expressions whose evaluation determine the corresponding heap location. Finally, the variable X identifies the process-algebraic model after initialisation.

The command $\texttt{finish}\ E$ is used to *finalise* the process-algebraic model identified by E in the logic, given that it can successfully terminate. Finalisation is later explained in more detail, in Section 3.4.

The specification command `action` E $a(E')$ `do` C is used to link the execution of programs with the execution of process-algebraic models. More specifically, it executes the program C in the context of the model identified by E, as the process-algebraic action a that takes (the evaluation of) E' as an input argument. The soundness argument of the program logic establishes a refinement relation between programs and their models, and this relation is established by synchronising program execution with process execution, with help of these *action blocks*.

The `inact` C command denotes a *partially executed* action program; one that still has to execute C. Likewise to `inatom`, this command can only occur during runtime and is not written by users.

Lastly, `query` E is used to connect process-algebraic reasoning to deductive reasoning: it allows the deductive proof of the program to rely on (or *assume*) properties that are proven to hold (or *guaranteed*) on the process-algebraic model identified by E, via process-algebraic analysis. These are the properties that are encoded as assertions $?(\cdot)$ in this model. Of course, this would require linking process-algebraic state to program state, which we come to later, in Sections 3.3 and 3.4.

3.2.3. Free Variables and Substitution

We use the standard (overloaded) notations $\mathsf{FV}(E)$, $\mathsf{FV}(B)$, $\mathsf{FV}(\Pi)$ and $\mathsf{FV}(C)$ to refer to the set of free *program* variables in the given (Boolean) expression E and B, abstraction binder Π, and command C, respectively. Moreover, the notation $E[X/E']$ denotes the *substitution* of the program variable X for the expression E' inside E; and likewise for Boolean expressions, abstraction binders, and commands. The full definitions of $\mathsf{FV}(\cdot)$ and $(\cdot)[X/E]$ are mostly standard, and therefore deferred to [19].

3.2.4. User Programs

As just discussed, our simple programming language contains runtime syntax—instructions that are not written by users but are only introduced during runtime. Commands that are free of such runtime constructs are called *user commands*.

Definition 9 (User commands)**.** *Any command C is defined to be a* user command*, denoted* $\mathsf{user}(C)$*, if C does not contain sub-commands of the forms* `inatom` C' *and* `inact` C'*, for any command C'.*

3.2.5. Wellformedness

Moreover, our verification approach only applies to *well-formed commands*. Notably, our technique requires that, for any program of the form `action` _ `do` C and `inact` C, the inner action program C only contains a subcategory of commands, excluding atomic commands and specification-only constructs, in particular nested action blocks. The latter is needed since actions must be atomically observable by environmental threads. This restriction is captured by the following definition.

Definition 10 (Basic programs, well-formed programs)**.** *Any command C is defined to be* basic*, denoted* $\mathsf{basic}(C)$*, if C does not contain any atomic sub-programs, i.e.,* `atomic` *or* `inatom`*, nor specification-specific language constructs, i.e.,* `process`*,* `action`*,* `inact`*,* `finish`*, or* `query`*.*

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

Lemma 2. $\mathsf{basic}(C)$ *implies* $\mathsf{wf}(C)$ *for any command C.*

3.2.6. Operational Semantics

The denotational semantics of expressions $[\![E]\!]s$ and conditions $[\![B]\!]s$ are again defined in the standard way, and evaluate to *Val* and *Bool*, respectively, where $s \in \mathit{Store} \triangleq \mathit{Var} \to \mathit{Val}$ is a *(program) store* that gives an interpretation to all program variables.

The operational semantics of programs is defined in terms of a binary small-step reduction relation $\rightsquigarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ between *program configurations*. A program configuration $\mathfrak{C} = (C, h, s) \in \mathit{Conf} \triangleq$

$Cmd \times Heap \times Store$ is a triple, consisting of a command C as well as a *heap* h that models shared memory, and a store $s \in Store$ that models thread-local memory. Any program configuration of the form $(\texttt{skip}, h, s)$ is defined to be *final* or *terminated*. Heaps $h \in Heap \triangleq Val \rightharpoonup_{\mathsf{fin}} Val$ are defined to be finite partial mappings from values to values. Heap locations are themselves values, so that they can be assigned to, and read from, local variables, and thus be handled as any value. The function $\mathsf{dom} : Heap \to 2^{Val}$ denotes the *mapped domain* of a given heap, so that $\mathsf{dom}(h) \triangleq \{v \mid h(v) \neq \text{undefined}\}$.

Definition 11 (Small-step operational semantics of programs)**.**

$\rightsquigarrow$-ASSIGN
$$(X := E, h, s) \rightsquigarrow (\texttt{skip}, h, s[X \mapsto [\![E]\!]s])$$

$\rightsquigarrow$-READ
$$(X := [E], h, s) \rightsquigarrow (\texttt{skip}, h, s[X \mapsto h([\![E]\!]s)])$$

$\rightsquigarrow$-WRITE
$$\frac{v \in \mathsf{dom}(h)}{([E_1] := E_2, h, s) \rightsquigarrow (\texttt{skip}, h[v \mapsto [\![E_2]\!]s], s)} \; \textit{where } v = [\![E_1]\!]s$$

$\rightsquigarrow$-ALLOC
$$\frac{v \notin \mathsf{dom}(h)}{(X := \texttt{alloc}\ E, h, s) \rightsquigarrow (\texttt{skip}, h[v \mapsto [\![E]\!]s], s[X \mapsto v])}$$

$\rightsquigarrow$-DISPOSE
$$(\texttt{dispose}\ E, h, s) \rightsquigarrow (\texttt{skip}, h \setminus [\![E]\!]s, s)$$

$\rightsquigarrow$-SEQ-L
$$\frac{(C_1, h, s) \rightsquigarrow (C_1', h', s')}{(C_1; C_2, h, s) \rightsquigarrow (C_1'; C_2, h', s')}$$

$\rightsquigarrow$-SEQ-R
$$(\texttt{skip}; C, h, s) \rightsquigarrow (C, h, s)$$

$\rightsquigarrow$-IF-TRUE
$$\frac{[\![B]\!]s}{(\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, h, s) \rightsquigarrow (C_1, h, s)}$$

$\rightsquigarrow$-IF-FALSE
$$\frac{\neg [\![B]\!]s}{(\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2, h, s) \rightsquigarrow (C_2, h, s)}$$

$\rightsquigarrow$-WHILE
$$(\texttt{while}\ B\ \texttt{do}\ C, h, s) \rightsquigarrow (\texttt{if}\ B\ \texttt{then}\ (C; \texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else}\ \texttt{skip}, h, s)$$

$\rightsquigarrow$-PAR-L
$$\frac{\neg \mathsf{locked}(C_2) \quad (C_1, h, s) \rightsquigarrow (C_1', h', s')}{(C_1 \parallel C_2, h, s) \rightsquigarrow (C_1' \parallel C_2, h', s')}$$

$\rightsquigarrow$-PAR-R
$$\frac{\neg \mathsf{locked}(C_1) \quad (C_2, h, s) \rightsquigarrow (C_2', h', s')}{(C_1 \parallel C_2, h, s) \rightsquigarrow (C_1 \parallel C_2', h', s')}$$

$\rightsquigarrow$-PAR-SKIP
$$(\texttt{skip} \parallel \texttt{skip}, h, s) \rightsquigarrow (\texttt{skip}, h, s)$$

$\rightsquigarrow$-ATOM
$$(\texttt{atomic}\ C, h, s) \rightsquigarrow (\texttt{inatom}\ C, h, s)$$

$\rightsquigarrow$-INATOM-STEP
$$\frac{(C, h, s) \rightsquigarrow (C', h', s')}{(\texttt{inatom}\ C, h, s) \rightsquigarrow (\texttt{inatom}\ C', h', s')}$$

$\rightsquigarrow$-INATOM-SKIP
$$(\texttt{inatom}\ \texttt{skip}, h, s) \rightsquigarrow (\texttt{skip}, h, s)$$

$\rightsquigarrow$-PROC
$$(X := \texttt{process}\ (\lambda x.P)(E)\ \texttt{over}\ \Pi, h, s) \rightsquigarrow (\texttt{skip}, h, s)$$

$\rightsquigarrow$-FINISH
$$(\texttt{finish}\ E, h, s) \rightsquigarrow (\texttt{skip}, h, s)$$

$\rightsquigarrow$-ACT
$$(\texttt{action}\ E\ a(E')\ \texttt{do}\ C, h, s) \rightsquigarrow (\texttt{inact}\ C, h, s)$$

$\rightsquigarrow$-INACT-STEP
$$\frac{(C, h, s) \rightsquigarrow (C', h', s')}{(\texttt{inact}\ C, h, s) \rightsquigarrow (\texttt{inact}\ C', h', s')}$$

$\rightsquigarrow$-INACT-SKIP
$$(\texttt{inact}\ \texttt{skip}, h, s) \rightsquigarrow (\texttt{skip}, h, s)$$

$\rightsquigarrow$-QUERY
$$(\texttt{query}\ E, h, s) \rightsquigarrow (\texttt{skip}, h, s)$$

Most of the transition rules are standard; see for example [34]. The *update* notation $s[X \mapsto v]$ defines a store that is equal to s, except that X is mapped to v. A similar notation is used for heaps, namely $h[v_1 \mapsto v_2]$. Moreover, the notation $h \setminus v$ denotes the *removal* of the entry at v in h.

An interesting aspect of the operational semantics is that atomic programs are executed using a small-step reduction strategy (via $\rightsquigarrow$-INATOM-STEP and $\rightsquigarrow$-INATOM-SKIP), rather than a big-step execution, which is more customary. This is done for technical reasons: it simplifies the establishment of a simulation/refinement between programs and their models. Consequently, we use a notion of a *locked program* to define the transition rules for atomic programs. Any command C is said to be *(globally) locked* if C executes an atomic program, i.e., if C has $\texttt{inatom}\ C'$ as a subprogram for some C'.

Definition 12 (Locked programs)**.** *Any command C is* locked *if* $\mathsf{locked}(C)$ *holds, where* $\mathsf{locked} \subset \mathit{Cmd}$ *is defined as follows, by structural recursion on C:*

$$\mathsf{locked}(C) \triangleq \begin{cases} \mathsf{true} & \text{if } C = \texttt{inatom}\ C' \\ \mathsf{locked}(C_1) & \text{if } C = C_1; C_2 \\ \mathsf{locked}(C_1) \vee \mathsf{locked}(C_2) & \text{if } C = C_1 \parallel C_2 \\ \mathsf{locked}(C') & \text{if } C = \texttt{inact}\ C' \\ \mathsf{false} & \text{otherwise} \end{cases}$$

The rules $\rightsquigarrow$-PAR-L and $\rightsquigarrow$-PAR-R for parallel composition allow a thread to make an execution step only if the other thread is not locked, thereby preventing thread interference while executing atomic programs. One might ask whether this handling of locks could not potentially lead to deadlock scenarios, for example by encountering configurations $(C_1 \parallel C_2, h, s)$ during runtime for which both $\mathsf{locked}(C_1)$ and $\mathsf{locked}(C_2)$ hold. However, we will later see and prove that no such deadlocks can be reached, given that one starts with an initial configuration that contains a user program.

Furthermore, the specification-only language constructs do not affect the state of the program (not the heap nor the store) and are essentially handled as if they were comments. Notice however, that commands of the form $\texttt{action}\ \_\ \texttt{do}\ C$ are first reduced to $\texttt{inact}\ C$ before C is being executed. This is done for technical reasons, as this makes it more convenient to later establish a simulation relation between execution steps of programs and processes.

The semantics of programs has the following preservation properties.

Lemma 3. *Program execution preserves basicality and wellformedness:*

1. *If* $\mathsf{basic}(C)$ *and* $(C, h, s) \rightsquigarrow (C', h', s')$*, then* $\mathsf{basic}(C')$*.*
2. *If* $\mathsf{wf}(C)$ *and* $(C, h, s) \rightsquigarrow (C', h', s')$*, then* $\mathsf{wf}(C')$*.*

3.2.7. Fault Semantics

Apart from an operational semantics, we also define a *fault semantics* for programs [35] that classifies runtime errors that may occur during program execution. Its definition uses two auxiliary functions, $\mathsf{acc}(C, s)$ and $\mathsf{writes}(C, s)$, for obtaining the set of heap locations that *can be accessed* or *written-to*, respectively, in a next reduction step of C. Their definitions are deferred to [19] as well, as they are quite lengthy and not essential for understanding the definition of the fault semantics.

The fault semantics of program configurations $\mathfrak{C}$ is expressed as a predicate $\lightning(\mathfrak{C})$ that is inductively defined as follows.

Definition 13 (Fault semantics of programs)**.**

$$\frac{[\![E]\!]s \notin \mathsf{dom}(h)}{\lightning(X := [E], h, s)}\ \lightning\text{-READ} \qquad \frac{[\![E_1]\!]s \notin \mathsf{dom}(h)}{\lightning([E_1] := E_2, h, s)}\ \lightning\text{-WRITE} \qquad \frac{[\![E]\!]s \notin \mathsf{dom}(h)}{\lightning(\texttt{dispose}\ E, h, s)}\ \lightning\text{-DISPOSE} \qquad \frac{\lightning(C_1, h, s)}{\lightning(C_1; C_2, h, s)}\ \lightning\text{-SEQ}$$

$$\frac{\lightning(C_1, h, s) \qquad \neg\mathsf{locked}(C_2)}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-PAR-L} \qquad \frac{\lightning(C_2, h, s) \qquad \neg\mathsf{locked}(C_1)}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-PAR-R} \qquad \frac{\mathsf{locked}(C_1) \qquad \mathsf{locked}(C_2)}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-DEADLOCK}$$

$$\frac{\neg\mathsf{locked}(C_1) \qquad \neg\mathsf{locked}(C_2) \qquad \mathsf{acc}(C_1, s) \cap \mathsf{writes}(C_2, s) \neq \emptyset}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-RACE-1}$$

$$\frac{\neg\mathsf{locked}(C_1) \qquad \neg\mathsf{locked}(C_2) \qquad \mathsf{acc}(C_2, s) \cap \mathsf{writes}(C_1, s) \neq \emptyset}{\lightning(C_1 \parallel C_2, h, s)}\ \lightning\text{-RACE-2}$$

$$\frac{\lightning(C, h, s)}{\lightning(\texttt{inatom}\ C, h, s)}\ \lightning\text{-ATOMIC} \qquad \frac{\lightning(C, h, s)}{\lightning(\texttt{inact}\ C, h, s)}\ \lightning\text{-ACTION}$$

Intuitively, a program configuration exhibits a fault if it (1) accesses unallocated memory, or (2) is deadlocked, or (3) allows performing a data-race.

More specifically, $\lightning$-READ expresses that heap reading $X := [E]$ faults if the heap location at E is unoccupied. For the same reason, also heap writing ($\lightning$-WRITE) and heap deallocation ($\lightning$-DISPOSE) may fault. The $\lightning$-PAR-L rule expresses that any parallel program $C_1 \parallel C_2$ can fault if C_1 can fault, given that C_2 is not locked, or the other way around ($\lightning$-PAR-R covers the other direction). Program configurations that hold multiple global locks are also considered to be faulting, by $\lightning$-DEADLOCK. Finally, the fault semantics encodes the definition of a *data-race*, via $\lightning$-RACE-1 and $\lightning$-RACE-2. To clarify, any configuration (C, h, s) exhibits a data-race if C has (at least) two threads that can both access a common location in h in the next reduction step, where at least one of these accesses is a write.

We will later see that the soundness argument of our program logic covers that verified programs are free of faults. More specifically, we will prove that, for any program C for which a proof can be derived, we have that C is fault-free with respect to any heap h and store s that satisfy C's precondition, and moreover, that every configuration that is reachable from (C, h, s) is also fault-free.

Finally, to show that the operational semantics of programs is coherent with respect to faults, we prove that the operational semantics is *progressive* for all non-faulting program configurations.

Theorem 1 (Progress of $\rightsquigarrow$)**.** *For any program configuration $\mathfrak{C}$ for which $\neg\lightning(\mathfrak{C})$ holds, either $\mathfrak{C}$ is final, or there exists a configuration $\mathfrak{C}'$ such that $\mathfrak{C} \rightsquigarrow \mathfrak{C}'$.*

3.3. Assertions

The assertion language of our verification approach is defined by the following grammar.

Definition 14 (Assertions)**.**

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

Assertions can be built from plain Boolean expressions B, and may contain several standard connectives from predicate logic: universal and existential quantifiers, and disjunction. Moreover, logical conjunction ($\wedge$) is replaced by the *separating conjunction* $*$ from Concurrent Separation Logic

(CSL). The $*_{i\in I}\mathcal{P}_i$ connective is the *iterated separating conjunction*, with I a finite set that represents $\mathcal{P}_0 * \cdots * \mathcal{P}_n$, given that $I = \{0, \ldots, n\}$. The $-\!\!*$ connective is known as the *magic wand* and is used to describe hypothetical judgments, much like the logical implication from predicate logic.

Apart from these standard CSL connectives, the assertion language contains three different heap ownership predicates $\overset{\pi}{\hookrightarrow}_t$, with $\pi \in \mathbb{Q}$ a rational number that represents a *fractional permission*, and t the *heap ownership type*, as well as an ownership predicate Proc_π for program abstractions. Finally $\widetilde{P} \approx \widetilde{Q}$ intuitively means that $\widetilde{P}$ and $\widetilde{Q}$ are bisimilar processes with respect to the current state.

The definitions of free variables $\mathsf{FV}(\mathcal{P})$ of assertions $\mathcal{P}$, and substitution $\mathcal{P}[X/E]$ in $\mathcal{P}$, are the standard ones and are therefore deferred to [19]. Assertions that are free of $\overset{\pi}{\hookrightarrow}_t$ and Proc_π predicates are called *pure*. Any assertion that is not pure is said to be *spatial*.

3.3.1. Heap Ownership

The assertion $E_1 \overset{\pi}{\hookrightarrow}_t E_2$ is the *heap ownership assertion* and expresses that the heap contains the value represented by the expression E_2 at heap location E_1. Moreover, π and t together determine the access rights to this heap location. In more detail, depending on the ownership type t, the $\overset{\pi}{\hookrightarrow}_t$ ownership predicates express different access rights to the associated heap location:

- *Standard heap ownership.* $E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E_2$ is the *standard heap ownership predicate* from (intuitionistic) separation logic that provides read-access whenever $0 < \pi < 1$, and write-access in case $\pi = 1$. Moreover, the subscript std indicates that the associated heap location E_1 is *not* bound to any process-algebraic model. We say that a heap location $v \in \mathit{Val}$ 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 v, that is, $v \in \mathsf{dom}(\Pi)$.
- *Process heap ownership.* $E \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E'$ is the *process heap ownership predicate*, which indicates that the heap location at E is bound by an active process-algebraic abstraction, but in a purely *read-only* manner. More precisely, $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ assertions exclusively grant read-access, even in case $\pi = 1$.
- *Action heap ownership.* $E \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E'$ is the *action heap ownership predicate*, which indicates that the heap location E is bound by an active process-algebraic model, and is used in the context of an action block, in a *read/write* manner.

Observe that action points-to assertions $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ essentially give the same access rights as $\overset{\pi}{\hookrightarrow}_{\mathsf{std}}$ assertions. Nevertheless, they are both needed, to be able to distinguish between bound and unbound heap locations in the logic. For example, the program logic must not allow to deallocate memory that is currently bound to (protected by) an active process-algebraic model, as this would be unsound.

Moreover, even though $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates never grant write access, we will later see that the proof system allows $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates to be upgraded to $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ inside action blocks, and $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ again provides write access when $\pi = 1$. More precisely, $E \overset{1}{\hookrightarrow}_{\mathsf{proc}} E'$ predicates grant the *capability to regain write access to* E, in the context of an action program. This system of upgrading enforces that all modifications to E happen in the context of `action` $E_{abstr}\ a(E'_{abstr})$ `do` C commands, so that the modifications are protected and can be recorded by the program abstraction identified by E_{abstr}, as the action a.

In addition to these three heap ownership predicates, we derive a fourth such predicate, called the *process–action heap ownership predicate*. This ownership predicate is equivalent to $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ only if π denotes write access, and otherwise it is equivalent to $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$.

Definition 15 (Process–action heap ownership)**.**

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

This derived predicate is for later use, in the proof system of our program logic. Finally, the notation $E \overset{\pi}{\hookrightarrow}_t -$ is sometimes used as shorthand for $\exists X. E \overset{\pi}{\hookrightarrow}_t X$, where $X \notin \mathsf{FV}(E)$.

3.3.2. Process Ownership

The $\mathsf{Proc}_\pi(E, \widetilde{P}, \Pi)$ assertion expresses *ownership* of a program abstraction that is identified by E, where the abstraction is represented by the process $\widetilde{P}$. Ownership in this sense means that the thread has knowledge of the existence of the process-algebraic model $\widetilde{P}$, as well as the right to execute as prescribed by this model. The mapping 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 models. 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 $\widetilde{P}$, and be consumed in the proof system by `action` programs.

Even though reasoning about process-algebraic models is done purely on the level of process-algebraic state, in the program logic it is allowed to mix program state with process-algebraic state. This is indicated by the tilde above the $\widetilde{P}$, which means that P can have both program variables and process-algebraic variables. Such processes are called *hybrid processes* and are defined as follows.

Definition 16 (Hybrid expressions, conditions and processes)**.**

$$\begin{array}{rcl} \widetilde{E} \in \mathit{HExpr} & ::= & v \mid x \mid X \mid \widetilde{E} + \widetilde{E} \mid \widetilde{E} - \widetilde{E} \mid \cdots \\ \widetilde{B} \in \mathit{HCond} & ::= & \mathsf{true} \mid \mathsf{false} \mid \neg\widetilde{B} \mid \widetilde{B} \wedge \widetilde{B} \mid \widetilde{E} = \widetilde{E} \mid \widetilde{E} < \widetilde{E} \mid \cdots \\ \widetilde{P}, \widetilde{Q} \in \mathit{HProc} & ::= & \varepsilon \mid \delta \mid a(E) \mid ?(\widetilde{B}) \mid \widetilde{P} \cdot \widetilde{Q} \mid \widetilde{P} + \widetilde{Q} \mid \widetilde{P} \parallel \widetilde{Q} \mid \widetilde{P} \between \widetilde{Q} \mid \Sigma_x\, \widetilde{P} \mid B : \widetilde{P} \mid \widetilde{P}^* \end{array}$$

These hybrid processes thus allow mixing process-algebraic reasoning with deductive reasoning using our program logic. The function $\mathsf{fv}(\widetilde{P})$ is used for obtaining the set of free *process-algebraic* variables in $\widetilde{P}$, and $\mathsf{FV}(\widetilde{P})$ for obtaining all free *program* variables in $\widetilde{P}$ (and likewise for $\widetilde{E}$ and $\widetilde{B}$).

We shall later see that the program logic allows replaces processes $\widetilde{P}$ inside $\mathsf{Proc}_\pi(E, \widetilde{P}, \Pi)$ predicates by bisimilar ones. However, note that one cannot use the standard notion of bisimilarity as defined in Definition 7 for this in case $\widetilde{P}$ has any program variables occurring freely in it. To resolve this, we include a relation $\widetilde{P} \approx \widetilde{Q}$ in the assertion language, stating that $\widetilde{P}$ and $\widetilde{Q}$ are bisimilar while taking into account any (pure) information that is available from the context. This is further clarified in Section 3.3.7, after we discussed the models of the logic.

3.3.3. Models of the Program Logic

Before Section 3.3.7 discusses the semantics of assertions, this section first introduces *permission heaps* and *process maps*, that form the basis for the models of our concurrent separation logic. Permission heaps extend ordinary program heaps (i.e., *Heap*) to capture the three different types t of heap ownership, whereas process maps capture the state and ownership of process-algebraic abstractions.

Let us start by introducing *fractional permissions*, which are used in the definitions of both permission heaps and process maps.

3.3.4. Fractional Permissions

In the assertion language, all heap/process ownership predicates have an associated rational number $\pi \in \mathbb{Q}$. There are used to express the "amount" of ownership that is available to the corresponding heap location or program model.

We define a rational number π to be a *(Boyland) fractional permission* in case $\pi \in (0, 1]_\mathbb{Q}$ [36]. The original work of Boyland uses fractional permissions to distinguish between write access ($\pi = 1$) and read access ($0 < \pi < 1$) to some shared resource. However, in our work this is slightly different, since the fractional access permissions π annotated to $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ predicates never provide write access.

To conveniently handle fractional permissions, we define basic notions of *validity* ($\mathsf{valid}_\mathbb{Q}$) and *disjointness* ($\perp_\mathbb{Q}$) of rational numbers, as follows.

Definition 17 (Permission validity, Permission disjointness)**.**

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

The predicate $\mathsf{valid}_{\mathbb{Q}} : \mathbb{Q} \to \mathit{Prop}$ determines whether the given rational number is within the range $(0, 1]_{\mathbb{Q}}$, that is, is a valid Boyland fractional permission. (Here *Prop* is the sort of propositions.) The binary relation $\perp_{\mathbb{Q}} : \mathbb{Q} \to \mathbb{Q} \to \mathit{Prop}$ 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 4. $\mathsf{valid}_{\mathbb{Q}}$ *and* $\perp_{\mathbb{Q}}$ *satisfy the following properties.*

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* $\pi_1 \perp_{\mathbb{Q}} (\pi_2 + \pi_3)$*.*

3.3.5. Permission Heaps

The models of our program logic use *permission heaps* to give a semantic meaning to heap ownership. Permission heaps and their heap cells are defined as follows, and are slightly richer than ordinary program heaps (*Heap*) to be able to administer the access permissions and the different ownership types.

Definition 18 (Permission heap cells, Permission heaps)**.**

$$\begin{aligned} \mathit{hc} \in \mathit{PermHeapCell} \;&::=\; \mathsf{free} \mid \langle v \rangle^{\pi}_{\mathsf{std}} \mid \langle v \rangle^{\pi}_{\mathsf{proc}} \mid \langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}} \mid \mathsf{inv} \\ \mathit{ph} \in \mathit{PermHeap} \;&\triangleq\; \mathit{Val} \to \mathit{PermHeapCell} \end{aligned}$$

Permission heaps *ph* are defined to be total functions from values (representing heap locations) to *permission heap cells*, *hc*, which in turn are inductively defined to be one of the following:

- free, which is an *unoccupied heap cell*.
- $\langle v \rangle^{\pi}_{\mathsf{std}}$, which is a *standard heap cell* that stores the value $v \in \mathit{Val}$. Standard heap cells are the models of the standard heap ownership predicates, $\overset{\pi}{\hookrightarrow}_{\mathsf{std}}$.
- $\langle v \rangle^{\pi}_{\mathsf{proc}}$, which is a *process heap cell* that stores the value v. These are used as models of the $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ ownership predicates.
- $\langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}}$, which is an *action heap cell* that stores the value v_1. Action heap cells are used as the models for the $\overset{\pi}{\hookrightarrow}_{\mathsf{act}}$ predicates. Moreover, action heap cells store a second value v_2. This extra value is maintained for technical reasons, to help in establishing soundness of the program logic. The value v_2 is referred to as a *snapshot value*: a copy of the original value stored by the heap cell, that is made when an action block was entered.
- inv, which is an *invalid*, or *corrupted*, permission heap cell.

Note that, unlike program heaps, permission heaps are defined to be total functions, where the heap cells have an explicit notion of being free. This is done to give permission heaps and their cells nicer algebraic properties. The *unit permission heap* is defined to be $\mathbb{1}_{\mathsf{ph}} \triangleq \lambda v \in \mathit{Val}\,.\,\mathsf{free}$, containing free at every entry. Furthermore, permission heap cells also have an explicit notion of being invalid. Invalid heap cells inv represent the erroneous result of composing two incompatible heap cells.

We now define several operations on permission heaps.

Validity. Any permission heap *ph* is defined to be *valid* if the permissions of all *ph*'s heap cells are valid, where free is always valid and inv is never valid.

Definition 19 (Validity of permission heaps)**.** *A permission heap ph is defined to be* valid*, written* $\mathsf{valid}_{\mathsf{ph}}(ph)$*, if* $\mathsf{valid}_{\mathsf{hc}}(ph(v))$ *holds for every* $v \in Val$*, where the* $\mathsf{valid}_{\mathsf{hc}}$ *predicate is defined as follows.*

$$\mathsf{valid}_{\mathsf{hc}}(hc) \triangleq \begin{cases} \mathsf{true} & \text{if } hc = \mathsf{free} \\ \mathsf{valid}_{\mathbb{Q}}(\pi) & \text{if } hc = \langle v \rangle^{\pi}_{\mathsf{std}} \vee hc = \langle v \rangle^{\pi}_{\mathsf{proc}} \vee \langle v, v' \rangle^{\pi}_{\mathsf{act}} \text{ for some } v, v' \\ \mathsf{false} & \text{if } hc = \mathsf{inv} \end{cases}$$

Disjointness. Two permission heaps ph_1 and ph_2 are *disjoint* if all their heap cells are pairwise *compatible* and their underlying permissions are disjoint.

Definition 20 (Disjointness of permission heaps)**.** *Two permission heaps,* ph_1 *and* ph_2*, are* disjoint*, denoted* $ph_1 \perp_{\mathsf{ph}} ph_2$*, if* $ph_1(v) \perp_{\mathsf{hc}} ph_2(v)$ *holds for every* $v \in Val$*, where the* $\perp_{\mathsf{hc}}$ *relation is defined as follows.*

$$hc_1 \perp_{\mathsf{hc}} hc_2 \triangleq \begin{cases} \mathsf{valid}_{\mathsf{hc}}(hc_2) & \text{if } hc_1 = \mathsf{free} \\ \mathsf{valid}_{\mathsf{hc}}(hc_1) & \text{if } hc_2 = \mathsf{free} \\ \pi_1 \perp_{\mathbb{Q}} \pi_2 \wedge v_1 = v_2 & \text{if } hc_1 = \langle v_1 \rangle^{\pi_1}_{\mathsf{std}} \wedge hc_2 = \langle v_2 \rangle^{\pi_2}_{\mathsf{std}} \\ \pi_1 \perp_{\mathbb{Q}} \pi_2 \wedge v_1 = v_2 & \text{if } hc_1 = \langle v_1 \rangle^{\pi_1}_{\mathsf{proc}} \wedge hc_2 = \langle v_2 \rangle^{\pi_2}_{\mathsf{proc}} \\ \pi_1 \perp_{\mathbb{Q}} \pi_2 \wedge v_1 = v_2 \wedge v_1' = v_2' & \text{if } hc_1 = \langle v_1, v_1' \rangle^{\pi_1}_{\mathsf{act}} \wedge hc_2 = \langle v_2, v_2' \rangle^{\pi_2}_{\mathsf{act}} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Disjoint union. The following operation defines the disjoint union (i.e., the composition) of two permission heaps.

Definition 21 (Disjoint union of permission heaps)**.** *The disjoint union* $ph_1 \uplus_{\mathsf{ph}} ph_2$ *of any two permission heaps* ph_1, ph_2 *is defined to be the permission heap* $\lambda v \in Val\,.\, ph_1(v) \uplus_{\mathsf{hc}} ph_2(v)$*, with* $\uplus_{\mathsf{hc}}$ *defined as follows.*

$$hc_1 \uplus_{\mathsf{hc}} hc_2 \triangleq \begin{cases} hc_1 & \text{if } hc_2 = \mathsf{free} \\ hc_2 & \text{if } hc_1 = \mathsf{free} \\ \langle v_1 \rangle^{\pi_1 + \pi_2}_{\mathsf{std}} & \text{if } hc_1 = \langle v_1 \rangle^{\pi_1}_{\mathsf{std}} \wedge hc_2 = \langle v_2 \rangle^{\pi_2}_{\mathsf{std}} \wedge v_1 = v_2 \\ \langle v_1 \rangle^{\pi_1 + \pi_2}_{\mathsf{proc}} & \text{if } hc_1 = \langle v_1 \rangle^{\pi_1}_{\mathsf{proc}} \wedge hc_2 = \langle v_2 \rangle^{\pi_2}_{\mathsf{proc}} \wedge v_1 = v_2 \\ \langle v_1, v_1' \rangle^{\pi_1 + \pi_2}_{\mathsf{act}} & \text{if } hc_1 = \langle v_1, v_1' \rangle^{\pi_1}_{\mathsf{act}} \wedge hc_2 = \langle v_2, v_2' \rangle^{\pi_2}_{\mathsf{act}} \wedge v_1 = v_2 \wedge v_1' = v_2' \\ \mathsf{inv} & \text{otherwise} \end{cases}$$

Note that $\uplus_{\mathsf{hc}}$ only gives a non-corrupted entry when applied to two compatible heap cells. Furthermore, free is neutral with respect to $\uplus_{\mathsf{hc}}$ while inv is absorbing.

Below are the most important properties of validity, disjointness and disjoint union.

Lemma 5. *(The analogous operations on permission heap cells have the exact same properties.)*

1. $ph_1 \uplus_{\mathsf{ph}} (ph_2 \uplus_{\mathsf{ph}} ph_3) = (ph_1 \uplus_{\mathsf{ph}} ph_2) \uplus_{\mathsf{ph}} ph_3$.
2. $ph_1 \uplus_{\mathsf{ph}} ph_2 = ph_2 \uplus_{\mathsf{ph}} ph_1$.
3. *If* $ph \uplus_{\mathsf{ph}} \mathbb{1}_{\mathsf{ph}} = ph$.
4. *If* $ph_1 \perp_{\mathsf{ph}} ph_2$*, then* $\mathsf{valid}_{\mathsf{ph}}(ph_1 \uplus_{\mathsf{ph}} ph_2)$.
5. *If* $ph_1 \perp_{\mathsf{ph}} ph_2$ *and* $(ph_1 \uplus_{\mathsf{ph}} ph_2) \perp_{\mathsf{ph}} ph_3$*, then also*
   - *(a)* $ph_2 \perp_{\mathsf{ph}} ph_3$ *and*
   - *(b)* $ph_1 \perp_{\mathsf{ph}} (ph_2 \uplus_{\mathsf{ph}} ph_3)$.

3.3.6. Process Maps

The models of the logic also use *process maps* in addition to permission heaps, to give a semantic meaning to process ownership predicates Proc_π in the logic. Process maps and their entries are defined as follows, where *binders* Λ are finite partial mappings from process variables to heap locations (i.e., values). These binders are the models for the abstraction binders Π defined earlier in Definition 8.

Definition 22 (Process map entries, process maps, binders)**.**

$$\begin{aligned} me \in \mathit{ProcMapEntry} &::= \mathsf{free} \mid \langle P, \Lambda \rangle^\pi \mid \mathsf{inv} \\ pm \in \mathit{ProcMap} &\triangleq \mathit{Val} \to \mathit{ProcMapEntry} \\ \Lambda \in \mathit{Binder} &\triangleq \mathit{ProcVar} \rightharpoonup_{\mathsf{fin}} \mathit{Val} \end{aligned}$$

Process maps are total mappings from values (identifying program abstractions) to *process map entries*, which are, in turn, inductively defined to one of the following three elements:

- free, which models *unoccupied* or *free* entries in *pm*.
- $\langle P, \Lambda \rangle^\pi$, which is an *occupied* process map entry. These are used as models for the $\mathsf{Proc}_\pi(E, \widetilde{P}, \Pi)$ assertions, where E identifies the process map entry in *pm*, and the binder Λ is a model for Π.
- inv, which denotes an *invalid*, or *corrupted*, process map entry.

Likewise to permission heaps, process maps are defined as total functions with entries that can explicitly be free or invalid, as this provides desirable algebraic properties. Corrupted entries represent the erroneous result of taking the disjoint union of two incompatible, non-disjoint entries. The *unit process map* is defined to be $\mathbb{1}_{\mathsf{pm}} \triangleq \lambda v \in \mathit{Val}\,.\,\mathsf{free}$, containing free at every entry.

We now define several operations and relations on process maps that are analogous to the operations defined earlier for permission heaps, starting with bisimilarity.

Bisimilarity. Any two process maps are said to be *bisimilar*, if all their entries are equal point-wise, or contain occupied entries with process components that are bisimilar.

Definition 23 (Process map bisimilarity)**.** *Two process maps pm_1 and pm_2 are defined to be bisimilar, denoted $pm_1 \cong_{\mathsf{pm}} pm_2$, if $pm_1(v) \cong_{\mathsf{mc}} pm_2(v)$ for every $v \in \mathit{Val}$, with the relation $\cong_{\mathsf{mc}}$ defined as follows.*

$$me_1 \cong_{\mathsf{mc}} me_2 \triangleq \begin{cases} \mathsf{true} & \text{if } me_1 = \mathsf{free} \wedge me_2 = \mathsf{free} \\ P_1 \cong P_2 \wedge \Lambda_1 = \Lambda_2 \wedge \pi_1 = \pi_2 & \text{if } me_1 = \langle P_1, \Lambda_1 \rangle^{\pi_1} \wedge me_2 = \langle P_2, \Lambda_2 \rangle^{\pi_2} \\ \mathsf{true} & \text{if } me_1 = \mathsf{inv} \wedge me_2 = \mathsf{inv} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Both $\cong_{\mathsf{pm}}$ and $\cong_{\mathsf{mc}}$ are equivalence relations. A notion of bisimilarity of process maps is needed in addition to ordinary equality, since for example disjoint union of process maps is not associative nor commutative with respect to ordinary equality, as opposed to bisimilarity. Moreover, we will later see that the program logic always allows replacing processes $\widetilde{P}$ inside $\mathsf{Proc}_\pi(X, \widetilde{P}, \Pi)$ predicates by bisimilar ones, as discussed earlier. But to handle such replacements at the semantic level, we allow process maps and their entries to be handled up to $\cong_{\mathsf{pm}}$ and $\cong_{\mathsf{mc}}$, respectively.

Validity. Any process map *pm* is said to be *valid* intuitively if none of *pm*'s entries are corrupt and all occupied entries of *pm* hold a valid associated fractional permission.

Definition 24 (Process map validity)**.** *Any process map pm is defined to be* valid*, denoted $\mathsf{valid}_{\mathsf{pm}}(pm)$, if $\mathsf{valid}_{\mathsf{mc}}(pm(v))$ holds for every $v \in \mathit{Val}$, with the $\mathsf{valid}_{\mathsf{mc}} : \mathit{ProcMapEntry} \to \mathit{Prop}$ predicate defined as follows.*

$$\mathsf{valid}_{\mathsf{mc}}(me) \triangleq \begin{cases} \mathsf{true} & \textit{if } me = \mathsf{free} \\ \mathsf{valid}_{\mathbb{Q}}(\pi) & \textit{if } me = \langle P, \Lambda \rangle^{\pi} \textit{ for some } P \textit{ and } \Lambda \\ \mathsf{false} & \textit{if } me = \mathsf{inv} \end{cases}$$

It is not difficult to see that $\mathbb{1}_{\mathsf{pm}}$ is trivially valid, and that bisimilarity is validity-preserving, i.e., $\mathsf{valid}_{\mathsf{pm}}(pm_1)$ and $pm_1 \cong_{\mathsf{pm}} pm_2$ implies $\mathsf{valid}_{\mathsf{pm}}(pm_2)$ for every pm_1 and pm_2; and likewise for $\mathsf{valid}_{\mathsf{mc}}$.

Disjointness. Two process maps are said to be *disjoint* if none of their entries are corrupt, and all fractional permissions of their entries are point-wise disjoint, as captured by the following definition.

Definition 25 (Process map disjointness)**.** *Any two process maps pm_1 and pm_2 are defined to be* disjoint*, denoted $pm_1 \perp_{\mathsf{pm}} pm_2$, if $pm_1(v) \perp_{\mathsf{mc}} pm_2(v)$ for every $v \in Val$, with the $\perp_{\mathsf{mc}}$ relation defined as follows.*

$$me_1 \perp_{\mathsf{mc}} me_2 \triangleq \begin{cases} \mathsf{valid}_{\mathsf{mc}}(me_1) & \textit{if } me_2 = \mathsf{free} \\ \mathsf{valid}_{\mathsf{mc}}(me_2) & \textit{if } me_1 = \mathsf{free} \\ \pi_1 \perp_{\mathbb{Q}} \pi_2 \wedge \Lambda_1 = \Lambda_2 & \textit{if } me_1 = \langle P_1, \Lambda_1 \rangle^{\pi_1} \wedge me_2 = \langle P_2, \Lambda_2 \rangle^{\pi_2} \textit{ for some } P_1 \textit{ and } P_2 \\ \mathsf{false} & \textit{otherwise} \end{cases}$$

The intuition of disjointness is that disjoint process maps can safely be composed without corrupting any of their entries. Disjointness is a symmetric relation and is a congruence with respect to bisimilarity, meaning that $pm_1 \perp_{\mathsf{pm}} pm_2$ and $pm_1 \cong_{\mathsf{pm}} pm_1'$ and $pm_2 \cong_{\mathsf{pm}} pm_2'$ implies $pm_1' \perp_{\mathsf{pm}} pm_2'$.

Disjoint union. The following operation defines the disjoint union of two process map (entries).

Definition 26 (Disjoint union of process maps)**.** *The* disjoint union *of two process maps pm_1 and pm_2 is defined as $pm_1 \uplus_{\mathsf{pm}} pm_2 \triangleq \forall v \,.\, pm_1(v) \uplus_{\mathsf{mc}} pm_2(v)$, with $\uplus_{\mathsf{mc}}$ defined as follows.*

$$me_1 \uplus_{\mathsf{mc}} me_2 \triangleq \begin{cases} me_1 & \textit{if } me_2 = \mathsf{free} \\ me_2 & \textit{if } me_1 = \mathsf{free} \\ \langle P_1 \parallel P_2, \Lambda_1 \rangle^{\pi_1 + \pi_2} & \textit{if } me_1 = \langle P_1, \Lambda_1 \rangle^{\pi_1} \wedge me_2 = \langle P_2, \Lambda_2 \rangle^{\pi_2} \wedge \Lambda_1 = \Lambda_2 \\ \mathsf{inv} & \textit{otherwise} \end{cases}$$

Likewise to disjoint union of permission heaps, the composition of incompatible process map entries produces a corrupted inv entry. The free entry is again neutral, whereas inv is absorbing (that is, composing inv with any entry results in inv). Disjoint union is a congruence with respect to bisimilarity, so that $pm_1 \cong_{\mathsf{pm}} pm_2$ and $pm_1' \cong_{\mathsf{pm}} pm_2'$ implies $pm_1 \uplus_{\mathsf{pm}} pm_1' \cong_{\mathsf{pm}} pm_2 \uplus_{\mathsf{pm}} pm_2'$.

Lemma 6. *(The analogous operations on process map entries have the exact same properties.)*

1. $pm_1 \uplus_{\mathsf{pm}} (pm_2 \uplus_{\mathsf{pm}} pm_3) \cong_{\mathsf{pm}} (pm_1 \uplus_{\mathsf{pm}} pm_2) \uplus_{\mathsf{pm}} pm_3$.
2. $pm_1 \uplus_{\mathsf{pm}} pm_2 \cong_{\mathsf{pm}} pm_2 \uplus_{\mathsf{pm}} pm_1$.
3. $pm \uplus_{\mathsf{pm}} \mathbb{1}_{\mathsf{pm}} \cong_{\mathsf{pm}} pm$.
4. *If* $pm_1 \perp_{\mathsf{pm}} pm_2$, *then* $\mathsf{valid}_{\mathsf{pm}}(pm_1 \uplus_{\mathsf{pm}} pm_2)$.
5. *If* $pm_1 \perp_{\mathsf{pm}} pm_2$ *and* $(pm_1 \uplus_{\mathsf{pm}} pm_2) \perp_{\mathsf{pm}} pm_3$, *then also*
   (a) $pm_2 \perp_{\mathsf{pm}} pm_3$, *and*
   (b) $pm_1 \perp_{\mathsf{pm}} (pm_2 \uplus_{\mathsf{pm}} pm_3)$.

3.3.7. Semantics of Assertions

Let us now define the semantic meaning of assertions. The semantics of assertions is defined in terms of 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). Its definition depends on an operation $[\![\cdot]\!] : \mathit{AbstrBinder} \to \mathit{Store} \to \mathit{Binder}$ for evaluating abstraction binders $\Pi = \{x_0 \mapsto E_0, \ldots, x_n \mapsto E_n\}$, that is defined as follows.

Definition 27 (Abstraction binder evaluation)**.**

$$[\![\{x_0 \mapsto E_0, \ldots, x_n \mapsto E_n\}]\!]s \;\triangleq\; \lambda x\,.\, \begin{cases} [\![E_i]\!]s & \text{if } x = x_i \text{ for some } 0 \leq i \leq n \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Moreover, recall that hybrid processes may contain both program variables and process-algebraic variables. The semantics of assertions relies on a closure operation for "closing" processes with respect to any program variable occurring in it. More specifically, given any hybrid process $\widetilde{P}$ and store s, the *s-closure of* $\widetilde{P}$, written $\widetilde{P}[s]$, is defined to be $\widetilde{P}[X/s(X)]_{X \in \mathsf{FV}(\widetilde{P})}$, i.e., replacing every free program variable X in $\widetilde{P}$ by $s(X)$. This operation is "closing" $\widetilde{P}$ in the sense that $\mathsf{FV}(\widetilde{P}[s]) = \emptyset$ and $\widetilde{P}[s] \in \mathit{Proc}$.

Definition 28 (Semantics of assertions)**.** *The modelling relation $ph, pm, s, g \models \mathcal{P}$ is defined by structural recursion on $\mathcal{P}$ as follows.*

$$\begin{array}{rcl}
ph, pm, s, g \models B & \textit{iff} & [\![B]\!]s \\
ph, pm, s, g \models \forall X.\mathcal{P} & \textit{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} & \textit{iff} & \exists v\,.\,(ph, pm, s[X \mapsto v], g[X \mapsto v]) \models \mathcal{P} \\
ph, pm, s, g \models \mathcal{P}_1 \vee \mathcal{P}_1 & \textit{iff} & ph, pm, s, g \models \mathcal{P}_1 \vee ph, pm, s, g \models \mathcal{P}_2 \\
ph, pm, s, g \models \mathcal{P}_1 * \mathcal{P}_2 & \textit{iff} & \exists ph_1, ph_2\,.\, ph_1 \perp_{\mathsf{ph}} ph_2 \wedge ph_1 \uplus_{\mathsf{ph}} ph_2 = ph \wedge \\
 & & \quad \exists pm_1, pm_2\,.\, pm_1 \perp_{\mathsf{pm}} pm_2 \wedge pm_1 \uplus_{\mathsf{pm}} pm_2 \cong_{\mathsf{pm}} pm \wedge \\
 & & \quad\quad ph_1, pm_1, s, g \models \mathcal{P}_1 \wedge ph_2, pm_2, s, g \models \mathcal{P}_2 \\
ph, pm, s, g \models *_{i \in I} \mathcal{P}_i & \textit{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 \mathcal{P}_1 \mathrel{-\!\!*} \mathcal{P}_2 & \textit{iff} & \forall ph', pm'\,.\,(ph \perp_{\mathsf{ph}} ph' \wedge pm \perp_{\mathsf{pm}} pm' \wedge ph', pm', s, g \models \mathcal{P}_1) \implies \\
 & & \quad ph \uplus_{\mathsf{ph}} ph', pm \uplus_{\mathsf{pm}} pm', s, g \models \mathcal{P}_2 \\
ph, pm, s, g \models E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{std}} E_2 & \textit{iff} & \mathsf{valid}_{\mathbb{Q}}(\pi) \wedge \exists \pi'\,.\, ph([\![E_1]\!]s) = \langle [\![E_2]\!]s \rangle^{\pi'}_{\mathsf{std}} \wedge \pi \leq \pi' \\
ph, pm, s, g \models E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{proc}} E_2 & \textit{iff} & \mathsf{valid}_{\mathbb{Q}}(\pi) \wedge \exists \pi'\,.\, ph([\![E_1]\!]s) = \langle [\![E_2]\!]s \rangle^{\pi'}_{\mathsf{proc}} \wedge \pi \leq \pi' \\
ph, pm, s, g \models E_1 \overset{\pi}{\hookrightarrow}_{\mathsf{act}} E_2 & \textit{iff} & \mathsf{valid}_{\mathbb{Q}}(\pi) \wedge \exists \pi, v\,.\, ph([\![E_1]\!]s) = \langle [\![E_2]\!]s, v \rangle^{\pi'}_{\mathsf{act}} \wedge \pi \leq \pi' \\
ph, pm, s, g \models \mathsf{Proc}_{\pi}(E, \widetilde{P}, \Pi) & \textit{iff} & \exists me'\,.\, me \perp_{\mathsf{mc}} me' \wedge pm([\![E]\!]g) \cong_{\mathsf{pm}} me \uplus_{\mathsf{mc}} me' \\
 & & \quad \textit{where } me = \langle \widetilde{P}[s], [\![\Pi]\!]s \rangle^{\pi} \\
ph, pm, s, g \models \widetilde{P} \approx \widetilde{Q} & \textit{iff} & \widetilde{P}[s] \cong \widetilde{Q}[s]
\end{array}$$

As usual, any separating conjunction $\mathcal{P}_1 * \mathcal{P}_2$ is satisfied by a model (ph, pm, s, g) if that model can both be split along ph and pm into two disjoint models, such that one satisfies $\mathcal{P}_1$ and the other satisfies $\mathcal{P}_2$. The semantic meaning of iterated separating conjunctions can be expressed simply in terms of the interpretation of the binary separating conjunction. Magic wands $\mathcal{P}_1 \mathrel{-\!\!*} \mathcal{P}_2$ are satisfied by a model if, for any disjoint extension of that model satisfying $\mathcal{P}_1$, the extended model satisfies $\mathcal{P}_2$.

Moving to the non-standard connectives; 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 $\mathsf{Proc}_{\pi}(E, \widetilde{P}, \Pi)$ are satisfied if the process map holds a matching entry at the position described by E, with a fractional permission at least π, and a process that at least includes all the behaviours of the process $\widetilde{P}[s]$. Finally, $\widetilde{P} \approx \widetilde{Q}$ is satisfied if $\widetilde{P}$ and $\widetilde{Q}$ are bisimilar with respect to the current state. To give an example of the use of $\approx$, consider the assertion $\mathsf{Proc}_{\pi}(E, 0 < X : \widetilde{P}, \Pi) * X = 2$. One might wish to replace $0 < X : \widetilde{P}$ with $\widetilde{P}$, considering that $X = 2$. But since $0 < X : \widetilde{P}$ is a process that includes program

variables (namely X), one can not immediately deduce that it is bisimilar to $\widetilde{P}$ according to Definition 7. However, we do have that $0 < X : \widetilde{P} \approx \widetilde{P} * X = 2$, since for every model (ph, pm, s, g) satisfying this assertion it holds that $s(X) = 2$. We shall later give entailment rules that allow such *context-dependent bisimulation equivalences* to be used to simplify processes inside Proc_π ownership predicates.

Lemma 7. *The $\models$ modelling relation satisfies the following properties:*

1. *$ph, pm, s, g \models \mathcal{P}$ and $pm \cong_{\mathsf{pm}} pm'$ implies $ph, pm', s, g \models \mathcal{P}$.*
2. *If $ph, pm, s, g \models \mathcal{P}$, then for any ph' and pm' such that $ph \perp_{\mathsf{ph}} ph'$ and $pm \perp_{\mathsf{pm}} pm'$ it holds that $ph \uplus_{\mathsf{ph}} ph', pm \uplus_{\mathsf{pm}} pm', s, g \models \mathcal{P}$.*

Lemma 7.1 is essential for allowing replacing process-algebraic abstractions by bisimilar ones inside the program logic. Lemma 7.2 expresses monotonicity, and states that adding resources does not invalidate the satisfiability of any assertion (i.e., adding more resources makes the assertion "more true"). This is a key property of *intuitionistic* separation logic and is necessary for proving soundness of the weakening rule, which we introduce later in Section 3.4.1.

3.3.8. Semantic Entailment

Let the denotation $[\![\mathcal{P}]\!] \triangleq \{(ph, pm, s, g) \mid (ph, pm, s, g) \models \mathcal{P}\}$ be the set of all models that are satisfied by the assertion $\mathcal{P}$. Given any two assertions $\mathcal{P}$ and $\mathcal{Q}$, the assertion $\mathcal{P}$ is defined to *semantically entail* $\mathcal{Q}$, denoted $\mathcal{P} \models \mathcal{Q}$, if every model of $\mathcal{P}$ is also a model of $\mathcal{Q}$, that is, $\mathcal{P} \models \mathcal{Q} \triangleq [\![\mathcal{P}]\!] \subseteq [\![\mathcal{Q}]\!]$. Semantic entailment is thus a preorder and a congruence for all connectives of the assertion language.

3.4. Proof System

This section introduces the proof system of our model-based verification technique, which consists of structural proof rules (Section 3.4.1) as well as Hoare proof rules (Section 3.4.2). This proof system essentially extends the CSL of [34] by adding permission accounting [36,37] and machinery for handling process-algebraic program abstractions.

3.4.1. Entailment Rules

Figure 3 presents the structural rules of the program logic. The notation $\mathcal{P} \dashv\vdash \mathcal{Q}$ is a shorthand for both $\mathcal{P} \vdash \mathcal{Q}$ and $\mathcal{Q} \vdash \mathcal{P}$, and indicates that the rule can be used in both directions.

The rules for the standard connectives are mostly as expected. PLAIN-DUPL expresses that plain expressions can freely be duplicated, whereas $*$-PLAIN shows that $*$ has the same meaning as $\wedge$ in the case of plain assertions. The rule $*$-WEAK shows that our concurrent separation logic is affine (intuitionistic) by allowing to forget about resources. The rules $*$-ASSOC and $*$-COMM express that the separating conjunction is associative and commutative, respectively, whereas $*$-TRUE allows composing any resource with true. The rule true-INTRO is the introduction rule for true, while false-ELIM is the elimination rule for false stating that anything can be derived from falsehood. The $-\!\!*$-INTRO and $-\!\!*$-ELIM rules show that magic wands can be used similarly to the modus ponens inference rule of propositional logic, with respect to $*$. The rules $\forall$-INTRO, $\forall$-ELIM, $\exists$-INTRO and $\exists$-ELIM are the standard introduction and elimination rules for universal and existential quantifiers. ITER-SPLIT-MERGE enables splitting and merging iterated separating conjunctions along the associated (finite) index set.

Clarifying the rules for handling heap ownership; $\hookrightarrow$-SPLIT-MERGE expresses that heap ownership predicates $\overset{\pi}{\hookrightarrow}_t$ of any type t may be *split* (in the left-to-right direction) as well as be *merged* (right-to-left) 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 $\hookrightarrow$-INCOMPATIBLE rule. Any heap ownership assertion with an invalid fractional permission associated to it entails false by $\hookrightarrow$-INVALID Furthermore, the $*$-PROCACT-SPLIT-MERGE inference rule states that iterated procact heap ownership predicates can be split into disjoint iterated proc and act predicates, or be merged into one such iteration.

Standard connectives (excerpt)

$$\text{PLAIN-DUPL}\quad B \dashv\vdash B * B \qquad *\text{-PLAIN}\quad B_1 * B_2 \dashv\vdash B_1 \wedge B_2 \qquad *\text{-WEAK}\quad \mathcal{P} * \mathcal{Q} \vdash \mathcal{P} \qquad *\text{-ASSOC}\quad \mathcal{P} * (\mathcal{Q} * \mathcal{R}) \dashv\vdash (\mathcal{P} * \mathcal{Q}) * \mathcal{R}$$

$$*\text{-COMM}\quad \mathcal{P} * \mathcal{Q} \vdash \mathcal{Q} * \mathcal{P} \qquad *\text{-TRUE}\quad \mathcal{P} \vdash \mathcal{P} * \mathsf{true} \qquad *\text{-MONO}\quad \frac{\mathcal{P} \vdash \mathcal{P}' \qquad \mathcal{Q} \vdash \mathcal{Q}'}{\mathcal{P} * \mathcal{Q} \vdash \mathcal{P}' * \mathcal{Q}'} \qquad \mathsf{true}\text{-INTRO}\quad \mathcal{P} \vdash \mathsf{true} \qquad \mathsf{false}\text{-ELIM}\quad \mathsf{false} \vdash \mathcal{P}$$

$$-\!\!*\text{-INTRO}\quad \frac{\mathcal{P} * \mathcal{Q} \vdash \mathcal{R}}{\mathcal{P} \vdash \mathcal{Q} -\!\!* \mathcal{R}} \qquad -\!\!*\text{-ELIM}\quad \frac{\mathcal{P} \vdash \mathcal{Q} -\!\!* \mathcal{Q}'}{\mathcal{P} * \mathcal{Q} \vdash \mathcal{Q}'} \qquad \forall\text{-INTRO}\quad \frac{\forall v\,.\,(\mathcal{P} \vdash \mathcal{Q}[X/v])}{\mathcal{P} \vdash \forall X.\mathcal{Q}} \qquad \forall\text{-ELIM}\quad \frac{\mathcal{P} \vdash \forall X.\mathcal{Q}}{\mathcal{P} \vdash \mathcal{Q}[X/v]} \qquad \exists\text{-INTRO}\quad \frac{\mathcal{P} \vdash \mathcal{Q}[X/v]}{\mathcal{P} \vdash \exists X.\mathcal{Q}}$$

$$\exists\text{-ELIM}\quad \frac{\mathcal{P} \vdash \exists X.\mathcal{Q}}{\exists v\,.\,(\mathcal{P} \vdash \mathcal{Q}[X/v])} \qquad \vee\text{-ELIM-L}\quad \frac{\mathcal{P} \vdash \mathcal{Q}_1}{\mathcal{P} \vdash \mathcal{Q}_1 \vee \mathcal{Q}_2} \qquad \vee\text{-ELIM-R}\quad \frac{\mathcal{P} \vdash \mathcal{Q}_2}{\mathcal{P} \vdash \mathcal{Q}_1 \vee \mathcal{Q}_2} \qquad \text{ITER-SPLIT-MERGE}\quad *_{i \in I_1 \uplus I_2} \mathcal{P}_i \dashv\vdash (*_{i \in I_1} \mathcal{P}_i) * (*_{i \in I_2} \mathcal{P}_i)$$

Heap ownership

$$\hookrightarrow\text{-SPLIT-MERGE}\quad \frac{\pi_1 \perp_{\mathbb{Q}} \pi_2}{E_1 \overset{\pi_1+\pi_2}{\hookrightarrow}_t E_2 \dashv\vdash E_1 \overset{\pi_1}{\hookrightarrow}_t E_2 * E_1 \overset{\pi_2}{\hookrightarrow}_t E_2} \qquad \hookrightarrow\text{-INCOMPATIBLE}\quad \frac{t_1 \neq t_2}{E \overset{\pi_1}{\hookrightarrow}_{t_1} E_1 * E \overset{\pi_2}{\hookrightarrow}_{t_2} E_2 \vdash \mathsf{false}} \qquad \hookrightarrow\text{-INVALID}\quad \frac{\neg \mathsf{valid}_{\mathbb{Q}}(\pi)}{E_1 \overset{\pi}{\hookrightarrow}_t E_2 \vdash \mathsf{false}}$$

$$*\text{-PROCACT-SPLIT-MERGE}\quad \frac{\forall i \in I_1\,.\,0 < \pi_i < 1 \qquad \forall i \in I_2\,.\,\pi_i = 1}{*_{i \in I_1 \uplus I_2} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{procact}} E_i' \dashv\vdash \left(*_{i \in I_1} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i'\right) * \left(*_{i \in I_2} E_i \overset{\pi_i}{\hookrightarrow}_{\mathsf{act}} E_i'\right)}$$

Process ownership

$$\approx\text{-BISIM}\quad \frac{P \cong Q}{\vdash P \approx Q} \qquad \approx\text{-REFL}\quad \vdash \widetilde{P} \approx \widetilde{P} \qquad \approx\text{-SYMM}\quad \widetilde{P} \approx \widetilde{Q} \vdash \widetilde{Q} \approx \widetilde{P} \qquad \approx\text{-TRANS}\quad \widetilde{P} \approx \widetilde{Q} * \widetilde{Q} \approx \widetilde{R} \vdash \widetilde{P} \approx \widetilde{R} \qquad \approx\text{-DUPL}\quad \widetilde{P} \approx \widetilde{Q} \vdash \widetilde{P} \approx \widetilde{Q} * \widetilde{P} \approx \widetilde{Q}$$

$$\approx\text{-CONG-}\circ\quad \frac{\circ \in \{\,\cdot\,, +\,, \parallel\,, \between\,\}}{\widetilde{P_1} \approx \widetilde{P_2} * \widetilde{Q_1} \approx \widetilde{Q_2} \vdash \widetilde{P_1} \circ \widetilde{Q_1} \approx \widetilde{P_2} \circ \widetilde{Q_2}} \qquad \approx\text{-CONG-SUM}\quad \frac{x \notin \mathsf{fv}(\widetilde{P}, \widetilde{Q})}{\widetilde{P} \approx \widetilde{Q} \vdash \Sigma_x \widetilde{P} \approx \Sigma_x \widetilde{Q}} \qquad \approx\text{-CONG-COND}\quad \frac{B_1 \dashv\vdash B_2}{\widetilde{P} \approx \widetilde{Q} \vdash B_1 : \widetilde{P} \approx B_2 : \widetilde{Q}}$$

$$\approx\text{-CONG-ITER}\quad \widetilde{P} \approx \widetilde{Q} \vdash \widetilde{P}^* \approx \widetilde{Q}^* \qquad \approx\text{-COND-TRUE}\quad \frac{B_1 \vdash B_2}{B_1 \vdash B_2 : \widetilde{P} \approx \widetilde{P}} \qquad \approx\text{-COND-FALSE}\quad \frac{B_1 \vdash B_2}{B_1 \vdash \neg B_2 : \widetilde{P} \approx \delta} \qquad \approx\text{-SUM-ALT}\quad \vdash \Sigma_x \widetilde{P} \approx \widetilde{P}[x/E] + \Sigma_x \widetilde{P}$$

$$\approx\text{-TERM}\quad \frac{\widetilde{P}\downarrow}{\vdash \widetilde{P} \approx \widetilde{P} + \varepsilon} \qquad \mathsf{Proc}\text{-}\cong\quad \mathsf{Proc}_\pi(X, \widetilde{P}, \Pi) * \widetilde{P} \approx \widetilde{Q} \dashv\vdash \mathsf{Proc}_\pi(X, \widetilde{Q}, \Pi) \qquad \mathsf{Proc}\text{-INVALID}\quad \frac{\neg \mathsf{valid}_{\mathbb{Q}}(\pi)}{\mathsf{Proc}_\pi(X, \widetilde{P}, \Pi) \vdash \mathsf{false}}$$

$$\mathsf{Proc}\text{-SPLIT-MERGE}\quad \frac{\pi_1 \perp_{\mathbb{Q}} \pi_2}{\mathsf{Proc}_{\pi_1+\pi_2}(X, \widetilde{P_1} \parallel \widetilde{P_2}, \Pi) \dashv\vdash \mathsf{Proc}_{\pi_1}(X, \widetilde{P_1}, \Pi) * \mathsf{Proc}_{\pi_2}(X, \widetilde{P_2}, \Pi)}$$

Figure 3. The entailment rules of the program logic.

Moving to the entailment rules for handling process-algebraic abstractions; Proc-$\cong$ allows replacing any process by one that is bisimilar in the current context. The rules $\approx$-REFL, $\approx$-SYMM and $\approx$-TRANS show that context-dependent bisimilarity forms an equivalence relation in the logic with respect to separating conjunction, while $\approx$-CONG-$\circ$, $\approx$-CONG-SUM, $\approx$-CONG-COND and $\approx$-CONG-ITER

together show that $\approx$ is a congruence relation in the logic for all process-algebraic connectives. The rules $\approx$-COND-TRUE and $\approx$-COND-FALSE allows eliminating conditionals in any processes (together with the Proc-$\cong$ rule that is). Moreover, $\approx$-SUM-ALT allows singling out a single choice out of a process-algebraic summation of choices. Notice here that one can pick any arbitrary *program* expression for singling out such a choice, which makes this rule particularly useful. Similarly to heap ownership, any process ownership with an invalid fractional permission entails false by Proc-INVALID. The rule $\approx$-TERM again makes explicit the intuitive meaning of successful termination, matching Proposition 1. Finally, Proc-SPLIT-MERGE allows splitting and merging process ownership predicates in the same style as $\overset{\pi}{\hookrightarrow}_t$, to distribute parallel processes over parallel threads. Notably, by splitting a predicate $\mathsf{Proc}_{\pi_1+\pi_2}(X, \widetilde{P}_1 \parallel \widetilde{P}_2, \Pi)$ 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 $\mathsf{Proc}_{\pi_i}(X, \widetilde{P}_i, \Pi)$ of the abstract model. Afterwards, when the threads join again the remaining partial abstractions can be merged back into a single $\mathsf{Proc}_{\pi_1+\pi_2}$ predicate. This system of splitting and merging thus provides a compositional, thread-modular way of verifying that programs meet their abstraction. The logical machinery of this is further discussed in Section 3.4.2.

Any deduction that can be derived using the rules of Figure 3 is sound in the standard sense:

Theorem 2 (Soundness of the entailment rules)**.** *$\mathcal{P} \vdash \mathcal{Q}$ implies $\mathcal{P} \models \mathcal{Q}$.*

3.4.2. Program Judgments

We now define *program judgments* and give the Hoare rules of the program logic. Judgments of programs are sequents (quintuples) of the form $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$. The right-hand side is a traditional Hoare triple, whereas $\mathcal{R}$ is a *resource invariant* that captures resources available only to atomically executing programs (i.e., in executions that are free of thread interference), and Γ an environment in the style of interface specifications of [38]. These *process environments* have the following definition.

Definition 29 (Process environments)**.**

$$\Gamma \in \mathit{ProcEnv} \;\; ::= \;\; \emptyset \mid \Gamma, \{\widetilde{B}\}\, P(x)$$

That is, process environments are comma-separated sequences of pairs $\{\widetilde{B}\}\, P(x)$ of processes P and their precondition $\widetilde{B}$, with x a placeholder variable for an input parameter that may occur freely in both P and $\widetilde{B}$. Note that processes do not have postconditions here; if desired one could encode process Hoare triples op top of these pairs as $\{\widetilde{B_{pre}}\}\, P(x)\, \{\widetilde{B_{post}}\} \triangleq \{\widetilde{B_{pre}}\}\, (P(x) \cdot ?(\widetilde{B_{post}}))$—by adding a trailing assertion. Moreover, even though process environments are given as sequences, they are used as if they were (finite) sets, as is customary, in the sense that the order of their pairs is unimportant.

Process environments contain the contracts of the process-algebraic models defined for the program under verification. In particular, they allow for assume-guarantee style reasoning: the proof system may *assume* validity of these contracts when dealing with process-algebraic models, since they must be *guaranteed* externally, for example via interactive theorem proving or model checking, e.g., using mCRL2 [25]. Validity of process contracts and process environments is defined as follows.

Definition 30 (Validity of process environments)**.** *Any pair $\{\widetilde{B}\}\, P(x)$ of a process P and its precondition $\widetilde{B}$ is defined to be* valid*, denoted $\models \{\widetilde{B}\}\, P(x)$, if $\forall s, \sigma\, .\, [\![\widetilde{B}[s]]\!]\sigma \implies \checkmark(P, \sigma)$.*

Any process environment Γ is defined to be valid *if $\models_{\mathsf{env}} \Gamma$, which is a judgment inductively defined by the following two rules.*

$$\frac{}{\models_{\mathsf{env}} \emptyset} \qquad\qquad \frac{\models_{\mathsf{env}} \Gamma \qquad \models \{\widetilde{B}\}\, P(x)}{\models_{\mathsf{env}} \Gamma, \{\widetilde{B}\}\, P(x)}$$

Figures 4 and 5 give the Hoare rules of the logic. The standard structural rules are essentially the same as the ones of classical CSL [34]. One minor difference is that HT-ATOMIC leaves true instead of "emp" after obtaining a resource invariant, since our logic is intuitionistic. (The assertion language of classical CSL contains an extra emp construct, for explicitly denoting that the heap is empty. In our intuitionistic version of the logic, resources are allowed to be thrown away using $*$-WEAK. As a consequence, assertions cannot express "precise" properties about the content of the heap, including emptiness of heaps.) Moreover, our assertion language does not contain the logical conjunction $\wedge$ connective, but has separating conjunction instead.

Standard structural rules

HT-FRAME
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \qquad \mathsf{FV}(\mathcal{F}) \cap \mathsf{mod}(C) = \emptyset}{\Gamma; \mathcal{R} \vdash \{\mathcal{P} * \mathcal{F}\}\, C\, \{\mathcal{Q} * \mathcal{F}\}}$$

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

HT-SHARE
$$\frac{\Gamma; \mathcal{R} * \mathcal{R}' \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P} * \mathcal{R}'\}\, C\, \{\mathcal{Q} * \mathcal{R}'\}}$$

HT-DISJ
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P}_1\}\, C\, \{\mathcal{Q}_1\} \quad \Gamma; \mathcal{R} \vdash \{\mathcal{P}_2\}\, C\, \{\mathcal{Q}_2\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}_1 \vee \mathcal{P}_2\}\, C\, \{\mathcal{Q}_1 \vee \mathcal{Q}_2\}}$$

HT-EX
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\} \qquad X \notin \mathsf{FV}(C)}{\Gamma; \mathcal{R} \vdash \{\exists X.\mathcal{P}\}\, C\, \{\exists X.\mathcal{Q}\}}$$

Standard proof rules

HT-SKIP
$$\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, \mathtt{skip}\, \{\mathcal{P}\}$$

HT-SEQ
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C_1\, \{\mathcal{S}\} \quad \Gamma; \mathcal{R} \vdash \{\mathcal{S}\}\, C_2\, \{\mathcal{Q}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C_1; C_2\, \{\mathcal{Q}\}}$$

HT-ATOMIC
$$\frac{\Gamma; \mathsf{true} \vdash \{\mathcal{P} * \mathcal{R}\}\, C\, \{\mathcal{Q} * \mathcal{R}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, \mathtt{atomic}\ C\, \{\mathcal{Q}\}}$$

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

HT-WHILE
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P} * B\}\, C\, \{\mathcal{P}\}}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, \mathtt{while}\ B\ \mathtt{do}\ C\, \{\mathcal{P} * \neg B\}}$$

HT-ASSIGN
$$\frac{X \notin \mathsf{FV}(\mathcal{R})}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}[X/E]\}\, X := E\, \{\mathcal{P}\}}$$

HT-PAR
$$\frac{\Gamma; \mathcal{R} \vdash \{\mathcal{P}_1\}\, C_1\, \{\mathcal{Q}_1\} \qquad \mathsf{FV}(\mathcal{R}, \mathcal{P}_1, C_1) \cap \mathsf{mod}(C_2) = \emptyset \quad \Gamma; \mathcal{R} \vdash \{\mathcal{P}_2\}\, C_2\, \{\mathcal{Q}_2\} \qquad \mathsf{FV}(\mathcal{R}, \mathcal{P}_2, C_2) \cap \mathsf{mod}(C_1) = \emptyset}{\Gamma; \mathcal{R} \vdash \{\mathcal{P}_1 * \mathcal{P}_2\}\, C_1 \parallel C_2\, \{\mathcal{Q}_1 * \mathcal{Q}_2\}}$$

Heap-related proof rules

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

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

HT-ALLOC
$$\frac{X \notin \mathsf{FV}(\mathcal{R}, E)}{\Gamma; \mathcal{R} \vdash \{\mathsf{true}\}\, X := \mathtt{alloc}\ E\, \{X \overset{1}{\hookrightarrow}_{\mathsf{std}} E\}}$$

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

Figure 4. Standard proof rules of the program logic.

Process-related proof rules

HT-PROC-INIT

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

HT-PROC-UPDATE

$$\frac{\begin{array}{c}\mathsf{fv}(a) = \{x_0, \ldots, x_n\} \subseteq \mathsf{dom}(\Pi) \qquad I = \{0, \ldots, n\} \\ B_{pre} = (\mathsf{guard}\ a\ E')[x_i/E_i]_{\forall i \in I} \qquad B_{post} = (\mathsf{effect}\ a\ E')[x_i/E_i]_{\forall i \in I} \\ \Gamma; \mathcal{R} \vdash \{*_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{procact}} E_i * B_{pre} * \mathcal{P}\}\ C\ \{*_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{procact}} E_i' * B_{post} * \mathcal{Q}\}\end{array}}{\begin{array}{l}\Gamma; \mathcal{R} \vdash \\ \qquad \left\{ \ *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i * B_{pre} * \mathsf{Proc}_\pi(E, a(E') \cdot \widetilde{P} + \widetilde{Q}, \Pi) * \mathcal{P} \ \right\} \\ \qquad \texttt{action}\ E\ a(E')\ \texttt{do}\ C \\ \qquad \left\{ \ *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i' * B_{post} * \mathsf{Proc}_\pi(E, \widetilde{P}, \Pi) * \mathcal{Q} \ \right\}\end{array}}$$

HT-PROC-FINISH

$$\frac{\mathsf{dom}(\Pi) = \{x_0, \ldots, x_n\} \qquad I = \{0, \ldots, n\}}{\begin{array}{l}\Gamma; \mathcal{R} \vdash \\ \qquad \left\{ \ *_{i \in I} \Pi(x_i) \overset{1}{\hookrightarrow}_{\mathsf{proc}} E_i * \mathsf{Proc}_1(E, \varepsilon + \widetilde{P}, \Pi) \ \right\} \\ \qquad \texttt{finish}\ E \\ \qquad \left\{ *_{i \in I} \Pi(x_i) \overset{1}{\hookrightarrow}_{\mathsf{std}} E_i \right\}\end{array}}$$

HT-PROC-QUERY

$$\frac{\mathsf{fv}(\widetilde{B}) = \{x_0, \ldots, x_n\} \subseteq \mathsf{dom}(\Pi) \qquad I = \{0, \ldots, n\} \qquad B_{assn} = \widetilde{B}[x_i/E_i]_{\forall i \in I}}{\begin{array}{l}\Gamma; \mathcal{R} \vdash \\ \qquad \left\{ \ *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i * \mathsf{Proc}_\pi(E, ?(\widetilde{B}) \cdot \widetilde{P} + \widetilde{Q}, \Pi) \ \right\} \\ \qquad \texttt{query}\ E \\ \qquad \left\{ *_{i \in I} \Pi(x_i) \overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}} E_i * \mathsf{Proc}_\pi(E, \widetilde{P}, \Pi) * B_{assn} \right\}\end{array}}$$

Figure 5. The extended proof rules related to handling process-algebraic models.

3.4.3. Heap Ownership

The HT-READ rule states that reading from the heap is allowed with *any* type t of heap ownership $\overset{\pi}{\hookrightarrow}_t$, whereas heap writing (HT-WRITE) is only allowed with ownership predicates of type std or act. The HT-WRITE rule thus restricts $\overset{\pi}{\hookrightarrow}_{\mathsf{proc}}$ assertions to exclusively grant read-access to the associated location. We will in a moment see that the proof rule for `action` programs 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 to which the heap location is subject to, inside an action block. The rule HT-ALLOC for heap allocation generates a new points-to predicate of type std, indicating that the allocated heap location is not (yet) subject to any program abstraction. Heap deallocation (HT-DISPOSE) requires a full *standard* ownership predicate for the associated heap location, thereby making sure that the deallocation does not break any bindings of active program abstractions, which would be unsound.

3.4.4. Process Ownership

Figure 5 gives the Hoare rules for introducing, eliminating and updating process-algebraic abstractions. The HT-PROC-INIT rule handles initialisation of an abstract model P with input parameter y, over a set of heap locations as specified by Π. This rule requires *standard* heap write ownership for any heap location that is to be bound by P according to Π, and these are converted to $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$. Moreover,

HT-PROC-INIT requires that the precondition B of P holds, which is constructed from $\widetilde{B}$ by replacing all process variables x_i by the values E_i at the corresponding heap locations as specified by Π. (Here we slightly abuse notation for ease of presentation. In the proof rule, we write $\widetilde{B}[x_i/E_i]_{\forall i \in I}$ for converting a condition $\widetilde{B}$ to a condition over only program variables, by substituting all free process variables x_i occurring in $\widetilde{B}$ by a program expression E_i. However, in our Coq formalisation we of course have a special operation for such conversions.) A Proc_1 predicate with full permission is ensured, giving the current thread full ownership of the abstraction.

The HT-PROC-UPDATE rule handles updates to program abstractions, by performing an action $a(E')$ in the context of an `action` E $a(E')$ `do` C program. This rule imposes four preconditions on handling `action` programs. First, a predicate of the form $\mathsf{Proc}_\pi(E, a(E') \cdot \widetilde{P} + \widetilde{Q}, \Pi)$ is required for some π. In particular, the process component must be of the form $a(E') \cdot \widetilde{P} + \widetilde{Q}$ and therewith allow performing the a action. After performing a the process will be reduced to $\widetilde{P}$, and $\widetilde{Q}$ will be discarded as the choice is made not to follow execution as prescribed by $\widetilde{Q}$. To get processes in the required format, the entailment rules in Figure 3 can be used together with the bisimulation equivalences given earlier in Figure 2. To give an example, processes of the form $a(E') \cdot \widetilde{P}$ can always be rewritten to $a(E') \cdot \widetilde{P} + \delta$ 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, a's guard must indeed hold. (The notation $\mathsf{guard}\ a\ E'$ is a shorthand for $(\mathsf{guard}\ a\ z)[z/E']$ for some fresh $z \in \mathit{ProcVar}$, and likewise for $\mathsf{effect}\ a\ E'$.) And last, the remaining resource $\mathcal{P}$ must hold as well.

Among the premises of HT-PROC-UPDATE is a proof derivation for the sub-program C, in which all required $\overset{\pi_i}{\hookrightarrow}_{\mathsf{proc}}$ predicates are "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 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 $\overset{\pi_i}{\hookrightarrow}_{\mathsf{procact}}$ instead, which simplifies the correctness proof. The HT-PROC-UPDATE rule ensures a process ownership predicate that holds the resulting process $\widetilde{P}$ after execution of a. In addition, updates to the heap are ensured that comply with the postconditions of the proof derivation of C.

HT-PROC-FINISH handles finalisation of process-algebraic models that can successfully terminate. A predicate $\mathsf{Proc}_1(E, \varepsilon + \widetilde{P}, \Pi)$ with *full* permission is required, implying that no other thread can have a fragment of the abstract model. The rule converts all bound $\overset{1}{\hookrightarrow}_{\mathsf{proc}}$ ownership predicates back to $\overset{1}{\hookrightarrow}_{\mathsf{std}}$ ownerships to indicate that these are no longer subject to the abstract model.

Lastly, HT-PROC-QUERY allows "querying" for properties $\widetilde{B}$ that are verified on the process algebra level. Recall that the main objective of process-algebraic analysis is to verify that all reachable assertions $?(\widetilde{B})$ hold. Observe that in this rule, the assertions may contain program variables in addition to process-algebraic variables, as these may have been introduced via summations ($\approx$-SUM-ALT) or input parameters (HT-PROC-INIT). The soundness argument of the logic makes sure that the process-algebraic analysis can still be done fully on the process level (i.e., without relying on program state), meaning that this rule really makes a fusion between process-algebraic reasoning and deductive reasoning.

3.5. Soundness

This section defines the semantic meaning of program judgments and discuss the soundness proof of the program logic. This soundness proof has been mechanised using Coq, which was non-trivial and required substantial auxiliary definitions. This section discusses the most important auxiliary definitions and explains their use. For further proof details we refer to the Coq development [19].

The soundness theorem relates program judgments to the operational semantics of programs and boils down to the following: if (1) a proof $\Gamma; \mathcal{R} \vdash \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ can be derived for any program C and (2) the contracts in Γ of all abstract models of C are satisfied (proven externally), 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 reduction steps with respect to the fault semantics of programs; see Definition 13.

Our definition of execution safety extends the well-known inductive definition of configuration safety of Vafeiadis [34] 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 the executions of all active models (with respect to $\xrightarrow{\alpha}$). 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 intermediate semantics is defined in Section 3.5.1 in terms of *ghost transitions* $\rightsquigarrow_{\mathsf{ghost}}$ that essentially define the lock-step execution of program transitions $\rightsquigarrow$ and the transitions $\xrightarrow{\alpha}$ of their abstractions. Our definition of "*executing safely for n execution steps*" includes that all $\rightsquigarrow$ steps can be simulated by $\rightsquigarrow_{\mathsf{ghost}}$ steps, and vice versa, for n execution steps. Thus, the end-result is a *refinement* between programs and their abstractions.

In addition to establishing such refinements, our definition of execution safety must ensure that the HT-PROC-QUERY proof rule is sound. In other words, it must allow relying on any assertions embedded in the process-algebraic models in a sound manner, as these are (assumed to be) verified externally. To account for these assertions, the definition of execution safety needs to maintain the invariant that all *active* program abstractions *preserve their execution safety* as defined in Definition 5 for n execution steps, with respect to the current state of the program. (Recall that any process P is said to be *safe* according to this definition if P's assertions always hold.) The details of maintaining this invariant are discussed further in Section 3.5.3.

Finally, Section 3.5.4 formally defines process execution safety—the semantic meaning of program judgments—and presents the exact soundness statement.

3.5.1. 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 $\rightsquigarrow_{\mathsf{ghost}} \subseteq \mathit{GhostConf}$ between *ghost configurations* $\mathfrak{G} = (C, h, pm, s, g) \in \mathit{GhostConf}$, which extend program configurations by two extra components, namely:

- A process map $pm \in \mathit{ProcMap}$ that is used to administer the state of all *active* (initialised, but not yet finalised) process-algebraic 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 specification-only (ghost) data strictly separated. Doing so eases establishing that variables referred to in ghost code do not interfere with regular program execution, and vice versa.

Ghost reductions essentially describe the *lock-step execution* of concrete programs ($\rightsquigarrow$ steps) and their abstractions ($\xrightarrow{\alpha}$ steps). Figure 6 presents an excerpt of the transition rules. This excerpt only contains the reduction rules related to program abstraction; all other rules are essentially the same as those of $\rightsquigarrow$, with the two extra configuration components simply carried over and left unchanged. Recall that the blue colourings are merely visual cues and do not have any special semantical meaning.

Clarifying the ghost reduction rules, ghost-PROC-INIT instantiates a new program abstraction and stores it in a free entry in pm. ghost-PROC-FINISH finalises program abstractions that are able to terminate successfully. The rules ghost-ACT-INIT, ghost-ACT-STEP and ghost-ACT-END handle the execution of action blocks. Before discussing these, first observe that the ghost semantics maintains an extra component m in `inact` m C commands, containing *(ghost) metadata*: extra runtime information about the process-algebraic model in whose context the program C is being executed. Concretely, ghost metadata m is defined as a quadruple $m = (a, v, v', h) \in \mathit{Act} \times \mathit{Val} \times \mathit{Val} \times \mathit{Heap}$, consisting of:

1. The label a of the action that is being executed;
2. The input argument v for this action;

3. The identifier v' of the corresponding process-algebraic model in the process map, in which the action a is to be executed; and
4. A copy h of the heap, made when the program started to execute the action block; that is, when the `action` program was reduced to `inact` by $\rightsquigarrow$.

ghost-PROC-INIT

$$\frac{pm(v) = \mathsf{free} \qquad Q = P[x/[\![E]\!]s] \qquad \Lambda = [\![\Pi]\!]s}{(X := \texttt{process}\ (\lambda x.P)(E)\ \texttt{over}\ \Pi, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (\texttt{skip}, h, pm[v \mapsto \langle Q, \Lambda\rangle^1], s, g[X \mapsto v])}$$

ghost-PROC-FINISH

$$\frac{v = [\![E]\!]g \qquad pm(v) \cong_{\mathsf{mc}} \langle P, \Lambda\rangle^\pi \qquad P \downarrow}{(\texttt{finish}\ E, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (s, h, pm \setminus v, s, g)}$$

ghost-ACT-INIT

$$(\texttt{action}\ E\ a(E')\ \texttt{do}\ C, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (\texttt{inact}\ (a, [\![E']\!]s, [\![E]\!]g, h)\ C, h, pm, s, g)$$

ghost-ACT-STEP

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

ghost-ACT-END

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

ghost-QUERY

$$\frac{v = [\![E]\!]g \qquad pm(v) \cong_{\mathsf{mc}} \langle P, \Lambda\rangle^\pi \qquad P, \|\Lambda\|(h) \xrightarrow{assn} P', \|\Lambda\|(h)}{(\texttt{query}\ E, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (\texttt{skip}, h, pm[v \mapsto \langle P', \Lambda\rangle^\pi], s, g)}$$

Figure 6. An excerpt of the transition rules of the ghost operational semantics.

The ghost-ACT-INIT reduction rule starts executing an `action` block by reducing it to an `inact` program, thereby assembling and attaching ghost metadata. In particular, a copy of the heap is made at this point, so that the ghost-ACT-END rule for finalising `inact` programs is able to access the original contents of the heap. This is needed to allow the abstraction to make a matching $\xrightarrow{\alpha}$ step; in particular to determine 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 maintained in process maps. Therefore, to be able to make an $\xrightarrow{\alpha}$ step, the ghost-ACT-END rule first needs to construct process-algebraic state out of the current program state. This is done using the auxiliary function $\|\cdot\| : \mathit{Binder} \to \mathit{Heap} \to \mathit{ProcStore}$ referred to as the *abstract state reification function*.

Definition 31 (Abstract state reification)**.**

$$\|\Lambda\|(h) \triangleq \lambda x \in \mathit{ProcVar}\,.\, \begin{cases} h(\Lambda(x)) & \text{if } x \in \mathsf{dom}(\Lambda) \text{ and } \Lambda(x) \in \mathsf{dom}(h) \\ \mathbb{1}_{\mathit{Val}} & \text{otherwise} \end{cases}$$

The ghost-ACT-STEP rule allows making reductions in the context of `inact` programs. Finally ghost-QUERY handles reductions of assertions and synchronises any $\xrightarrow{assn}$ reductions on the process level with reductions of queries on the program level, with respect to the reified program state.

3.5.2. Faulting Ghost Configurations

In addition to faulting program configurations (Definition 13) we also define a fault semantics for ghost configurations. This *ghost fault semantics* is expressed in terms of a predicate $\lightning_{\mathsf{ghost}}(\mathfrak{G})$ over ghost configurations $\mathfrak{G}$. Figure 7 gives an excerpt of the rules. Only the rules related to specification-only constructs are shown. All other rules are essentially the same as those of Definition 13. We shall later show and prove properties that connect the two faulting semantics.

$$\frac{\forall v\,.\,pm(v) \neq \mathsf{free}}{\lightning_{\mathsf{ghost}}(X := \texttt{process}\ (\lambda x.P)(E)\ \texttt{over}\ \Pi, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-PROC-FULL}$$

$$\frac{pm([\![E]\!]g) \in \{\mathsf{free}, \mathsf{inv}\}}{\lightning_{\mathsf{ghost}}(\texttt{finish}\ E, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-PROC-FINISH-1}$$

$$\frac{pm([\![E]\!]g) \cong_{\mathsf{mc}} \langle P, \Lambda\rangle^{\pi} \qquad P \not\downarrow}{\lightning_{\mathsf{ghost}}(\texttt{finish}\ E, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-PROC-FINISH-2}$$

$$\frac{\lightning_{\mathsf{ghost}}(C, h, pm, s, g)}{\lightning_{\mathsf{ghost}}(\texttt{inact}\ m\ C, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-ACT-STEP}$$

$$\frac{pm(v') \in \{\mathsf{free}, \mathsf{inv}\}}{\lightning_{\mathsf{ghost}}(\texttt{inact}\ (a, v, v', h_{old})\ \texttt{skip}, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-ACT-SKIP-1}$$

$$\frac{pm(v') \cong_{\mathsf{mc}} \langle P, \Lambda\rangle^{\pi} \quad \mathsf{image}(\Lambda) \not\subseteq \mathsf{dom}(h) \cap \mathsf{dom}(h')}{\lightning_{\mathsf{ghost}}(\texttt{inact}\ (a, v, v', h_{old})\ \texttt{skip}, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-ACT-SKIP-2}$$

$$\frac{pm(v') \cong_{\mathsf{mc}} \langle P, \Lambda\rangle^{\pi} \quad \neg\exists P'\,.\,P, \|\Lambda\|(h_{old}) \xrightarrow{a(v)} P', \|\Lambda\|(h)}{\lightning_{\mathsf{ghost}}(\texttt{inact}\ (a, v, v', h_{old})\ \texttt{skip}, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-ACT-SKIP-3}$$

$$\frac{pm([\![E]\!]g) \in \{\mathsf{free}, \mathsf{inv}\}}{\lightning_{\mathsf{ghost}}(\texttt{query}\ E, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-QUERY-FULL}$$

$$\frac{pm([\![E]\!]g) \cong_{\mathsf{mc}} \langle P, \Lambda\rangle^{\pi} \quad \mathsf{image}(\Lambda) \not\subseteq \mathsf{dom}(h) \cap \mathsf{dom}(h')}{\lightning_{\mathsf{ghost}}(\texttt{query}\ E, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-QUERY-COVER}$$

$$\frac{pm([\![E]\!]g) \cong_{\mathsf{mc}} \langle P, \Lambda\rangle^{\pi} \quad \neg\exists P'\,.\,P, \|\Lambda\|(h) \xrightarrow{\mathit{assn}} P', \|\Lambda\|(h)}{\lightning_{\mathsf{ghost}}(\texttt{query}\ E, h, pm, s, g)}\ \lightning_{\mathsf{ghost}}\text{-QUERY-STEP}$$

Figure 7. An excerpt of the fault semantics of ghost configurations.

Clarifying the ghost fault semantics; the initialisation of any process-algebraic model faults if there is no free entry available in pm (by $\lightning_{\mathsf{ghost}}$-PROC-FULL). The finalisation of program abstractions can fault if the corresponding entry in the process map is (1) either unoccupied or invalid ($\lightning_{\mathsf{ghost}}$-PROC-FINISH-1), or (2) contains a process-algebraic abstraction that is unable to successfully terminate (by the rule $\lightning_{\mathsf{ghost}}$-PROC-FINISH-2). Reductions within action blocks `inact` m C may fault if (1) m does not refer to an abstraction ($\lightning_{\mathsf{ghost}}$-ACT-SKIP-1), or (2) the abstraction relies on process variables that have an incorrect binding (by the rule $\lightning_{\mathsf{ghost}}$-ACT-SKIP-2), or (3) the process is not able to make a matching step ($\lightning_{\mathsf{ghost}}$-ACT-SKIP-3), or (4) the subprogram C is able to fault (by $\lightning_{\mathsf{ghost}}$-ACT-STEP). Any `query` program can fault under similar conditions as those of action programs.

The ghost semantics enjoys the same progress property as the standard operational semantics.

Theorem 3 (Progress of $\rightsquigarrow_{\mathsf{ghost}}$)**.** *For any ghost configuration $\mathfrak{G}$ for which $\neg\lightning_{\mathsf{ghost}}(\mathfrak{G})$ holds, either $\mathfrak{G}$ is final, or there exists a $\mathfrak{G}'$ such that $\mathfrak{G} \rightsquigarrow_{\mathsf{ghost}} \mathfrak{G}'$.*

Moreover, it is quite straightforward to establish a *forward simulation* between $\rightsquigarrow$ and $\rightsquigarrow_{\mathsf{ghost}}$. A matching backward simulation is ensured by the soundness argument of the program logic, as is customary for establishing refinements [39].

Lemma 8 (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* $(C, h, pm, s, g) \rightsquigarrow_{\mathsf{ghost}} (C', h', pm', s', g')$*, then* $(C, h, s) \rightsquigarrow (C', h', s')$*.*
2. *If* $\lightning (C, h, s)$*, then also* $\lightning_{\mathsf{ghost}} (C, h, pm, s, g)$*, for any pm and g.*

The above theorem also shows that the ghost fault semantics extends $\lightning$. The soundness argument of the program logic establishes that verified programs do not fault with respect to $\rightsquigarrow_{\mathsf{ghost}}$, and thus also do not fault with respect to $\rightsquigarrow$ by the above Lemma.

3.5.3. Preservation of Process Execution Safety

As already hinted upon in the preamble of this section, establishing soundness of the program logic requires maintaining an invariant stating that all active program abstractions retain their execution safety throughout program execution, with respect to Definitions 5 and 6. Since process maps are used to administer the status of all active program abstractions, we lift the notion of process configuration safety (Definition 5) to *safety of process maps*. Process map safety is expressed in terms of judgments of the form $h \models_{\mathsf{pm}} pm$ stating that pm is *safe* if all process-algebraic models stored in pm execute safely with respect to Definition 5 together with a process store that is constructed (reified) from h.

Definition 32 (Process map safety)**.**

$$h \models_{\mathsf{pm}} pm \triangleq \forall v \in Val\,.\, h \models_{\mathsf{mc}} pm(v)$$

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

$$h \models_{\mathsf{mc}} me \triangleq \begin{cases} \mathsf{true} & \text{if } me = \mathsf{free} \\ \checkmark(P, \|\Lambda\|(h)) & \text{if } me = \langle P, \Lambda \rangle^{\pi} \text{ for some } \pi \\ \mathsf{false} & \text{if } me = \mathsf{inv} \end{cases}$$

Free process cells are always safe whereas corrupted entries inv are never safe. Moreover, both $\models_{\mathsf{pm}}$ and $\models_{\mathsf{mc}}$ are closed under bisimilarity of process maps and their entries, respectively.

Lemma 9.

1. *If* $h \models_{\mathsf{pm}} pm$ *and* $pm \cong_{\mathsf{pm}} pm'$*, then* $h \models_{\mathsf{pm}} pm'$*.*
2. *If* $h \models_{\mathsf{mc}} me$ *and* $me \cong_{\mathsf{mc}} me'$*, then* $h \models_{\mathsf{mc}} me'$*.*

In a moment we will also define a notion of execution safety for commands. This notion of *program execution safety* maintains the aforementioned invariant that $h \models_{\mathsf{pm}} pm$ always holds throughout program execution, where h and pm are constructed from the current state, at every execution step. This invariant is needed to establish soundness of the HT-PROC-QUERY proof rule.

However, one must be careful on how to exactly state this invariant, to allow re-establishing it after every computation step. In most cases re-establishing the invariant is straightforward. For example, $h \models_{\mathsf{pm}} pm$ can be re-established after initialising a new program abstraction using the HT-PROC-INIT proof rule, by Definition 6 and by the structure of that proof rule. The invariant can also trivially be re-established after finalising an abstraction using HT-PROC-FINISH, 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, as illustrated below.

Technicality 1 (Potential problems due to heap writing)**.** *To see the potential problem, consider the following code snippet.*

```
1 guard x > 0;
2 effect x = 0;
3 action reset( );
```

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

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

The root of the problem is that the invariant should not necessarily have to hold during intermediate reduction steps while executing `action` programs, but only at the pre- and poststate of such programs. 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-ACT-INIT), 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 storing *snapshot values* v_2 alongside "concrete" values v_1. These snapshot values are used to construct *snapshot heaps*.

Definition 33 (Snapshot heaps)**.** *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* $\lfloor ph \rfloor_{\mathsf{snapshot}} \triangleq \lambda v \in \mathit{Val} \,.\, \lfloor ph(v) \rfloor_{\mathsf{snapshot}}$*, with*

$$\lfloor hc \rfloor_{\mathsf{snapshot}} \triangleq \begin{cases} v & \text{if } hc = \langle v \rangle^{\pi}_{\mathsf{proc}} \text{ for some } \pi \\ v_2 & \text{if } hc = \langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}} \text{ for some } v_1 \text{ and } \pi \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The snapshot heap $\lfloor ph \rfloor_{\mathsf{snapshot}}$ of any permission heap ph only contains heap cells bound by process-algebraic models, and is constructed by taking the snapshot values of all ph's action heap cells. As we shall see in a moment, the final invariant maintained by program execution safety will be $\lfloor ph \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm$, 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 abstract models, provide sufficient means for proving soundness of the program logic.

3.5.4. Adequacy

This section defines *program execution safety* and uses it to define the semantic meaning of program judgments, from which the soundness theorem (i.e., adequacy of the logic) can be formulated. Program execution safety extends on the well-known notion of *configuration safety* of [34], by adding permission accounting, process-algebraic state, and the machinery introduced earlier in this section.

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

Definition 34 (Concretisation)**.** Concretisation *of permission heaps is defined as a total function* $\lfloor \cdot \rfloor_{\mathsf{concr}} : \mathit{PermHeap} \to \mathit{Heap}$*, so that* $\lfloor ph \rfloor_{\mathsf{concr}} \triangleq \lambda v \in \mathit{Val} \,.\, \lfloor ph(v) \rfloor_{\mathsf{concr}}$*, with* $\lfloor ph(v) \rfloor_{\mathsf{concr}}$ *defined as*

$$\lfloor hc \rfloor_{\mathsf{concr}} \triangleq \begin{cases} v & \text{if } hc = \langle v \rangle^{\pi}_{\mathsf{std}} \text{ for some } \pi \\ v & \text{if } hc = \langle v \rangle^{\pi}_{\mathsf{proc}} \text{ for some } \pi \\ v_1 & \text{if } hc = \langle v_1, v_2 \rangle^{\pi}_{\mathsf{act}} \text{ for some } \pi \text{ and } v_2 \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

The heap concretisation operator constructs (program) heaps out of permission heaps by simply discarding all internal structure regarding process-algebraic 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 takes snapshot values whenever possible.

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

Definition 35 (Program execution safety)**.** *The* $\mathsf{safe}_0(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$ *predicate always holds, whereas* $\mathsf{safe}_{n+1}(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$ *holds if and only if the following five conditions hold.*

1. *If* $C = \mathtt{skip}$*, 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* $\lightning_{\mathsf{ghost}}(C, \lfloor ph \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}}, pm \uplus_{\mathsf{pm}} pm_F, s, g)$*.*

3. *For any* $v \in \mathsf{acc}(C, s)$ *it holds that* $ph(v) \notin \{\mathsf{free}, \mathsf{inv}\}$*.*

4. *For any* $v \in \mathsf{writes}(C, s)$ *it holds that* $\mathsf{full}_{\mathsf{hc}}(ph(v))$*.*

5. *For any* ph_J*,* ph_F*,* 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.*** $\lfloor ph \uplus_{\mathsf{ph}} ph_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm_C$*, 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.*** $\lfloor ph' \uplus_{\mathsf{ph}} ph'_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm'_C$*, and*
- ***5l.*** $\neg\mathsf{locked}(C')$ *implies* $ph'_J, pm'_J, s', g' \models \mathcal{R}$*, and*
- ***5m.*** $(C, \lfloor ph \uplus_{\mathsf{ph}} ph_J \uplus_{\mathsf{ph}} ph_F \rfloor_{\mathsf{concr}}, pm_C, s, g) \rightsquigarrow_{\mathsf{ghost}} (C', h', pm'_C, s', g')$*, and*
- ***5n.*** $\mathsf{safe}_n(C', ph', pm', s', g', \mathcal{R}, \mathcal{Q})$*.*

Clarifying the above definition, any configuration is safe for $n+1$ steps intuitively if: the postcondition $\mathcal{Q}$ is satisfied if 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***). (The predicate $\mathsf{full}_{\mathsf{hc}}(hc)$ is true whenever hc is an occupied heap cell with an associated fractional permission π equal to 1.) 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 $\rightsquigarrow_{\mathsf{ghost}}$ step (by ***5m***). Moreover, the resource invariant $\mathcal{R}$ must remain satisfied (due to ***5c*** and ***5l***) after making a computation step, whenever the current program is not locked. In addition, the process maps invariably remain safe with respect to the snapshot heap due to ***5e*** and ***5k***, as discussed in Section 3.5.3.

Lemma 10. *Program execution safety satisfies the following properties.*

1. *If* $\mathsf{safe}_n(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$ *and* $m \leq n$*, then* $\mathsf{safe}_m(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$*.*

2. *If* $\mathsf{safe}_n(C, ph, pm_1, s, g, \mathcal{R}, \mathcal{Q})$ *and* $pm_1 \cong_{\mathsf{pm}} pm_2$, *then* $\mathsf{safe}_n(C, ph, pm_2, s, g, \mathcal{R}, \mathcal{Q})$.
3. *If* $\mathsf{safe}_n(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$ *and* $\mathcal{Q} \models \mathcal{Q}'$, *then* $\mathsf{safe}_n(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q}')$.

1 in Lemma 10 states monotonicity in the sense that being safe for n reduction steps implies safety for less than n steps. 2 in Lemma 10 states that process maps can always be replaced by bisimilar ones in safe configurations. Finally, 3 in Lemma 10 states that postconditions may always be weakened.

3.5.5. Semantics of Program Judgments

The semantics of program judgments is defined in terms of a quintuple $\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$, expressing that C is safe for any number of reduction steps starting from any state satisfying $\mathcal{P}$.

Definition 36 (Semantics of program judgments)**.** $\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathcal{Q}\}$ *holds if and only if:*

(1) $\mathsf{user}(C)$, *and*
(2) *If* $\models_{\mathsf{env}} \Gamma$ *and* $\mathsf{wf}(C)$, *then for any* ph, pm, s, g *such that* $\mathsf{valid}_{\mathsf{ph}}(ph)$ *and* $\mathsf{valid}_{\mathsf{pm}}(pm)$ *and* $\lfloor ph \rfloor_{\mathsf{snapshot}} \models_{\mathsf{pm}} pm$ *and* $ph, pm, s, g \models \mathcal{P}$ *hold, it holds that:*

$$\forall n\, .\, \mathsf{safe}_n(C, ph, pm, s, g, \mathcal{R}, \mathcal{Q})$$

The underlying idea of the above definition, i.e., having a continuation-passing style definition for program judgments, has first been applied in [40] and has further been generalised in [41,42]. Moreover, the idea of defining execution safety in terms of an inductive predicate originates from [43]. These two concepts have been reconciled in [34] into a formalisation for the classical CSL of Brookes [23], that has been encoded and mechanically been proven in both Isabelle and Coq. Our definition builds on the latter, by having a refinement between programs and abstractions encoded in safe.

Observe that only judgments of user programs (i.e., commands free of runtime constructs like `inatom` and `inact`) have a semantic meaning. Also observe that the semantics of program judgments is conditional on the safety of Γ. It states that C executes safely for any number n of computation steps with respect to any state satisfying $\mathcal{P}$, only if Γ is safe—that is, only if all process-algebraic models for C are (assumed to be externally) verified. From the above definition it trivially follows that $\Gamma; \mathcal{R} \models \{\mathsf{false}\}\, C\, \{\mathcal{P}\}$ for any Γ, $\mathcal{R}$, $\mathcal{P}$ and user program C. Notice however that $\Gamma; \mathcal{R} \models \{\mathcal{P}\}\, C\, \{\mathsf{true}\}$ does not hold in general, since C might be able to fault, for example by having data-races.

The following main soundness theorem states that verified programs (i.e., programs for which a proof can be derived according to the proof rules given earlier in Figures 4 and 5) are semantically valid (that is, are fault-free, memory-safe, and refine their process-algebraic models).

Theorem 4 (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 Git repository accompanying this article [19].

The HT-PROC-UPDATE and HT-PROC-QUERY proof rules were the most difficult to prove sound, as their proofs require, among other things, (1) showing that the abstract model can always match the program with a simulating execution step, as well as (2) maintaining the invariant that any process-algebraic abstraction inside the process map is safe with respect to the reified program state. On top of that, the combination of (1) and (2) requires some extra bookkeeping to ensure that the snapshot heaps stored in ghost metadata agree with the snapshot values stored in permission heaps. This additional bookkeeping has been left out of the formalisation presented so far, but the details of this can be studied in the Coq formalisation.

4. Implementation

The presented verification approach has been implemented in the VerCors concurrency verifier, which specialises in automated verification of parallel and concurrent programs written in high-level languages like (subsets of) Java and C [8]. VerCors can reason about programs with heterogeneous concurrency features as in Java, as well as homogeneous concurrency like in OpenCL, and compiler directives as in OpenMP. VerCors allows specifying (concurrent) programs with annotations from a separation logic with permission accounting. VerCors supports reasoning about freedom of data-races, memory safety and functional program behaviour—compliance of the program annotations.

4.1. Tool Support

Tool support for our technique has been implemented in VerCors for languages with fork/join concurrency and statically-scoped parallel constructs [14]. 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.

Note however that VerCors does not yet support writing and reasoning about assertional processes $?(\cdot)$. Instead, the properties to verify on a process-algebraic level are specified as postconditions of the models. That is, VerCors currently only allows reasoning about postcondition properties of process-algebraic models, while the formalisation as presented in this article allows reasoning about properties also at intermediate points of process execution. But since the formalisation is more general than the VerCors implementation (as was already indicated in Section 3.4.2), soundness is retained.

Recall that process-algebraic abstractions are to be verified externally with our approach, for example using an interactive theorem prover or a model checker like mCRL2. Nevertheless, VerCors itself also has capabilities to reason about process-algebraic models. This is done by first *linearising* all specified processes, and then encoding these linearised processes together with their contracts into the Viper language, and delegate further reasoning to Viper. Any process is said to be *linear* if it does not use the $\|$ and $\lfloor\!\lfloor$ connectives. Linearisation is a mechanical (automated) procedure based on a rewrite system that uses a subset of the bisimulation equivalences of Figure 2 as rewrite rules [44] (but in one direction only), to try to eliminate parallel connectives. For example, a process term $(a_1 \cdot a_2) \parallel a_3$ can automatically be linearised to the bisimilar process $a_1 \cdot a_2 \cdot a_3 + a_1 \cdot a_3 \cdot a_2 + a_3 \cdot a_1 \cdot a_2$. Note that linearisation may not always succeed. VerCors outputs a verification error in case linearlisation fails.

Process-algebraic models may also algorithmically be analysed, for example using a model checker. We are currently actively investigating the use of the mCRL2 toolset [21] and the Ivy verifier [45] to reason about (different forms of) processes, and in different use cases; see for example [17], in which we use process-algebraic abstractions to reason about distributed message passing programs.

Finally, we would like to remark that the VerCors implementation of the abstraction technique is much richer than the simple language of Section 3.2 that is used to formalise the approach on. For example, the abstraction language in VerCors supports general recursion instead of Kleene iteration. VerCors also has support for several axiomatic data types that enrich the expressivity of reasoning with program abstractions, like (multi)sets, sequences and option types.

4.2. Coq Formalisation

The formalisation and soundness proof (Sections 3.1–3.5) of the program logic have been fully mechanised using Coq, as a deep embedding inspired by [34]. The overall implementation comprises over 23.000 lines of code. The Coq development and its documentation are available online [19].

5. Case Study

Finally, we demonstrate our verification approach on a well-known version of the leader election protocol [46] that is based on shared memory. Most importantly, this case study shows how our approach bridges the typical abstraction gap between process algebraic models and program implementations. In particular, it shows how a high-level process algebraic model of a leader election protocol, together with a contract for this model (checked with mCRL2 for various inputs), is formally connected to an actual program implementation of the protocol, using VerCors.

The protocol is performed by N concurrent workers that are organised in a ring, so that worker i only sends to worker $i+1$ and only receives from worker $i-1$, modulo N. The goal is to determine a leader among these workers. To find a leader, the election procedure assumes that each worker i receives a unique integer value to start with, and then operates in N rounds. In every round, (1) each worker sends the highest value it encountered so far to its right neighbour, (2) receives a value from its left neighbour, and (3) remembers the highest of the two. The result after N rounds is that all workers know the highest unique value in the network, allowing its original owner to announce itself as leader.

The case study has been verified with VerCors using the presented approach. All workers communicate via two standard non-blocking operations for message passing: `mp_send(r,msg)` for sending a message *msg* to the worker with rank r, and $msg :=$ `mp_recv(r)` for receiving a message from worker r. (The identifiers of workers are typically called *ranks* in message passing terminology. Ranks are simply natural numbers.) The election protocol is implemented on top of this message passing system.

The main challenge of this case study is to define a message passing system on the process algebra level that matches this implementation. To design such a system we follow the ideas of [46]; by defining two actions, $\mathsf{send}(r, msg)$ and $\mathsf{recv}(r, msg)$, that abstractly describe the behaviour of the concrete implementations in `mp_send` and `mp_recv`, respectively. Process algebraic summation $\Sigma_x \widetilde{P}$ is used 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 the `mp_send` and `mp_recv` procedures in the program, respectively. The latter rule uses a summation (shorthand) of the form $\Sigma_{x \in Msg} \widetilde{P}$ that can be considered equivalent to the process $\Sigma_x (x \in Msg : \widetilde{P})$.

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

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

5.1. Behavioural Specification

The main goal is proving that the implementation determines the correct leader upon termination. To prove this we first define a *behavioural specification* of the election protocol that hides all irrelevant implementation details, and prove the correctness property on this specification. Process algebra provides a proper abstraction language that suits our needs well, as the behaviour of leader election can concisely be specified in terms of sequences of sends and receives.

Figure 8 presents the process algebraic specification. In particular, ParElect specifies the *global* behaviour of the program whereas Elect specifies its *thread-local* behaviour. The ParElect process encodes the parallel composition of all eligible participants. ParElect takes a sequence *vs* of initial values as argument, whose length equals the total number of workers by its precondition. ParElect's postcondition (i.e., trailing assertion) 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.

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

Figure 8. The behavioural process-algebraic specification of the leader election protocol. Processes of the form `if` b `then` P `else` Q are a shorthand for $b : P + \neg b : Q$. Moreover, all `ensures` clauses of ParElect are translated into trailing assertions, as described earlier at the beginning of Section 3.4.2.

The Elect process takes four arguments, which are: the rank of the worker, the initial unique value v_0 of that worker, the current highest value v encountered by that worker, and finally the number n of remaining rounds. The rounds are implemented via general recursion. In each round all workers send their current highest value v to their right neighbour (on line 24), receive a value v' from their left neighbour (line 25), and continue with the highest of the two. The announce action is declared and used to announce the leader after n rounds. The effect 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 taken from a domain *Msg*. Since workers are organised in a ring it suffices to have only a single queue for every worker, meaning that the global communication channel architecture can be defined as a sequence of message queues: *chan* in the figure. The action contract of send(r, msg) expresses enqueuing *msg* onto the message queue $chan[r]$ of the worker with rank r. The effect 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, recv(r, msg)'s contract expresses dequeuing *msg* from $chan[r]$. The expression \old$(e)$ indicates that e is to be evaluated with respect to the pre-state of computation.

5.2. Protocol Implementation

Figure 9 presents the annotated implementation of the election protocol. (It should be noted here that the presentation is slightly different from the version that is verified with VerCors, to better connect to the theory discussed in the earlier sections to the case study. This is because VerCors uses Implicit Dynamic Frames [47] as its underlying logical framework, which is equivalent to separation logic [48] but handles ownership slightly differently. The details of this are deferred to [8,49].) The `elect` method contains the code that is executed by every worker. The contract of `elect`$(rank, v_0, v)$ states that the method body adheres to the behavioural specification $\mathsf{Elect}(rank, v_0, v, N)$ of the election protocol. Each worker performing `elect` enters a `for`-loop that iterates N times, whose loop invariant states that, at iteration i, the remaining program behaves as prescribed by the process $\mathsf{Elect}(rank, v_0, v, i-1)$. The invocations to `mp_send` and `mp_recv` on lines 32 and 36 are annotated with `with` clauses that resolve the assignments required by the `given` clauses in the contracts of `mp_send` and `mp_recv`. The `given` $\overline{\eta}$ annotation expresses that the parameter list $\overline{\eta}$ are extra ghost arguments for the sake of specification. Stated differently, $\overline{\eta}$ is a sequence of logical variables which are universally quantified at the (outer) level of the method contract (the types of these are left implicit for ease of presentation). 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 in the logic.

The `mp_send`$(rank, msg)$ method implements the operation of enqueuing *msg* onto the message queue of worker *rank*. Its implementation has been omitted for brevity. The contract of `mp_send` expresses that the enqueuing operation is abstracted as a $\mathsf{send}(rank, msg)$ action that is prescribed by an abstract model identified by X. The `mp_recv`$(X, rank)$ method implements the operation of dequeuing and returns the first message of the message queue of worker *rank*. The receive is prescribed as an abstract recv action, where the received message is ranged over by the summation on line 17.

Figure 10 presents bootstrapping code for the implementation of message passing. The `main` procedure initialises the communication channels whereas `parelect` spawns all workers. `main`(vs) additionally initialises and finalises the abstraction $\mathsf{ParElect}(vs)$ on the specification level (on lines 76 and 83, resp.) whose analysis allows establishing `main`'s postconditions. The procedure `parelect`(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 55–59) is called an *iteration contract* and assigns pre- and postconditions to every parallel instance. For more details on iteration contracts we refer to [50]. Most importantly, the iteration contract of each parallel worker states (on line 58) that it behaves as specified by Elect. Thus, we deductively verify in a *thread-modular way* that the program implements its behavioural specification. Observe that 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 62–64. Finally, the main correctness property is conveyed from the process level to the program level by the `query` X annotation on line 82, which "queries" for ParElect's postconditions.

```
1  ref seq⟨seq⟨Msg⟩⟩ C; // implementation of communication channels
2  ref int N; // total number of workers
3  ref 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 X, P̃, Q̃, Π, π, π′;
8  context {chan ↦ C} ∈ Π;
9  context ∃n . N ↪π_proc n ∗ 0 ≤ rank < n;
10 requires Proc_π′(X, send(rank, msg) · P̃ + Q̃, Π);
11 ensures Proc_π′(X, P̃, Π);
12 void mp_send(int rank, Msg msg) { /∗ omitted ∗/ }
13
14 given X, P̃, Q̃, Π, π, π′;
15 context {chan ↦ C} ∈ Π;
16 context ∃n . N ↪π_proc n ∗ 0 ≤ rank < n;
17 requires Proc_π′(X, (Σ_{x∈Msg} recv(rank, x) · P̃) + Q̃, Π);
18 ensures Proc_π′(X, P̃[x/\result], Π);
19 Msg mp_recv(int rank) { /∗ omitted ∗/ }
20
21 given X, n, Π, π, π′;
22 context {lead ↦ L, chan ↦ C} ∈ Π;
23 context N ↪π_proc n ∗ 0 ≤ rank < n;
24 requires Proc_π′(X, Elect(rank, v0, v, n), Π);
25 ensures Proc_π′(X, ε, Π);
26 void elect(int rank, Msg v0, Msg v)
27 {
28   loop_invariant 0 ≤ i ≤ n;
29   loop_invariant Proc_π′(X, Elect(rank, v0, v, n − i), Π);
30   for (int i := 0 to N)
31   {
32     mp_send((rank + 1) % N, v) with {
33       P̃ := Σ_{x∈Msg} recv(rank, x) · Elect(rank, v0, max(v,x), n − i − 1),
34       X := X, Q̃ := ε, Π := Π, π := π, π′ := π′
35     };
36     Msg v′ := mp_recv(rank) with {
37       P̃ := Elect(rank, v0, max(v,v′), n − i − 1),
38       X := X, Q̃ := ε, Π := Π, π := π, π′ := π′
39     };
40     v := max(v,v′);
41   }
42   if (v = v0) {
43     atomic {
44       action X announce(rank) do L := rank;
45     }
46   }
47 }
```

Figure 9. An excerpt of the annotated implementation of the leader election protocol. Annotations of the form `context` $\mathcal{P}$ are shorthand for `requires` $\mathcal{P}$; `ensures` $\mathcal{P}$.

```
given X, Π;
context {lead ↦ L, chan ↦ C} ∈ Π;
context N ↪(1/2)proc |vs| ∗ 0 < |vs|;
requires Proc1(X, ParElect(vs), Π);
ensures Proc1(X, ε, Π);
void parelect(seq⟨Msg⟩ vs)
{
  context N ↪(1/(4|vs|))proc |vs| ∗ 0 < |vs|;
  context 0 ≤ rank ≤ |vs|;
  context {lead ↦ L, chan ↦ C} ∈ Π;
  requires Proc1/|vs|(X, Elect(rank, vs[rank], vs[rank], |vs|), Π);
  ensures Proc1/|vs|(X, ε, Π);
  par (int rank := 0 to N)
  {
    elect(rank, vs[rank], vs[rank]) with {
      X := X, n := |vs|, Π := Π, π := 1/(4|vs|), π' := 1/|vs|
    };
  }
}

context N ↪(1)std − ∗ C ↪(1)std − ∗ L ↪(1)std −;
requires ∀i, j : int . (0 ≤ i < |vs| ∧ 0 ≤ j < |vs| ∧ vs[i] = vs[j]) ⟹ i = j;
ensures 0 ≤ \result < |vs|;
ensures ∀i : int . (0 ≤ i < |vs|) ⟹ vs[i] ≤ vs[\result];
int main(seq⟨Msg⟩ vs)
{
  N := |vs|;
  C := initialiseChannels(N);
  X := process ParElect(vs) over {chan ↦ C, lead ↦ L};

  commitLock(); // initialise the lock invariant
  parelect(vs) with { X := X, Π := {chan ↦ C, lead ↦ L} }; // spawn all worker threads
  uncommitLock(); // reclaim the lock invariant

  query X; // obtain the global correctness property from the abstract model
  finish X; // finalise the abstract model

  return L;
}
```

Figure 10. Bootstrap procedures of the leader election protocol.

5.3. Specification and Verification Details

The VerCors encoding of the presented leader election protocol comprises 433 lines of code, of which 275 are specification annotations (63, 5% of the total) and 27 lines are comments (6, 2% of the total). Out of the 275 lines of specification code, 62 are used for specifying the process-algebraic model (22, 5%), and the remaining 213 lines (77, 5%) for formally linking this model to the program code.

The average verification time with VerCors is 19, 75s (average of 30 runs), measured on a Macbook with an Intel Core i5 CPU with 2,9 GHz and 8Gb memory. All verification files are available online [19].

5.4. Industrial Applicability

Apart from the presented leader election case study, our approach has been applied in a larger, *industrial* case study covering the formal verification of a traffic tunnel emergency control system [15]. In this case study, we successfully verified a safety-critical component of an emergency control system

of an actual traffic tunnel that is currently in use in the Netherlands. This particular software component is responsible for handling any emergency situations that occur inside the traffic tunnel. For example, whenever a fire breaks out inside the tunnel or an accident occurs, it must start an emergency procedure to evacuate all people and turn on the emergency lights to help guide them out; control the fans to blow away any smoke; et cetera. Naturally the reliability demands imposed on such a software component are very high. Our research goal was to see if formal verification could help.

Our approach for this case study was to use mCRL2 to construct a formal, process-algebraic model of the software design, which was written informally as a state machine together with pseudo-code descriptions of the different system behaviours. We then analysed the state space of this model and checked whether it satisfies desirable properties, which we composed together with the company that wrote the actual code. Ultimately we found problematic behaviour: the system could, due to an unlucky combination of timing and events, reach a calamity state in which the emergency procedure is not started. However, the software company already knew of this problematic behaviour and deliberately provided us with an older version of their software. Nevertheless, we demonstrated that formal methods can indeed help to improve the quality of real-world industrial software.

In addition to modelling and analysing the software design with mCRL2, we also used VerCors to prove that the actual code implementation is soundly abstracted by the process-algebraic model, using the techniques presented in this article. We did this to increase the value of our formal model, and to ensure that its analysis is meaningful. Moreover, our verification also (indirectly) proved that the code implementation adheres to the pseudo-code specifications in the original software design.

Overall this case study highlights how the presented approach is applicable to real-world projects. We were able to identify potential vulnerabilities in the software design, and could link (a formal model of) the software design to the actual code. We are currently involved in a follow-up project with the same company, and aim to apply our technique *during* the software development process.

6. Related Work

Significant progress has been made on the theory of concurrent software verification over the last years [1–5,51–53]. This line of research proposes advanced program logics that all provide some notion of expressing and restricting thread interference of various complexity, via *protocols* [54]—formal descriptions of how shared program state is allowed to evolve over time. In our approach protocols have the form of processes.

The original work on CSL [24] allows specifying simple thread interference in shared-memory concurrent programs via resource invariants and critical regions. Later, RGSep [55] merges CSL with rely-guarantee reasoning to enable describing more fine-grained inter-thread interference by identifying atomic concurrent actions. Many modern program logics build on these principles and propose even more advanced ways of verifying shared-memory concurrency. For example, TaDa [5] and CaReSL [3] express thread interference protocols through state-transition systems. iCAP [51] and Iris [56] propose a more unified approach by accepting user-defined monoids to express protocols on shared state, together with invariants restricting these protocols. Iris provides reasoning support for proving language properties in Coq, whereas our focus is on proving (concrete) programs correct.

In the distributed setting, Disel [6] allows specifying protocols for distributed systems. Disel builds on dependent type theory and is implemented as a shallow embedding in Coq. Even though their approach is more expressive than ours, it has to be used in the context of Coq and thus can be applied only semi-automatically at the moment. Villard et al. [57] present a program logic for message passing concurrency, where threads may communicate over channels using native send/receive primitives. This program logic allows specifying protocols via *contracts*, which are state-machines in the style of Session Types [58] to describe channel behaviour. Our technique is more general however, as the approach of Villard et al. is tailored specifically to reason about basic shared-memory message passing (Section 5 for example demonstrates how a system of message passing can be realised using process-algebraic abstractions). Actor Services [59] is a program logic with assertions to express the

consequences of asynchronous message transfers between actors—independent program units that communicate via message passing. The meta-theory of Actor Services has not been proven sound.

Most of the related work given so far is essentially theoretical and tend to focus primarily on expressivity—on contributing approaches that are expressive yet not necessarily easy to implement into SMT-based program verifiers like for example VerCors of Viper. In fact, for most of these approaches it is very challenging to implement such automated tool support. Instead, they have to be applied in pen-and-paper style, or in the context of an interactive theorem prover like Coq or Isabelle. Our abstraction approach is different, in the sense that its aim is not to maximise expressivity (for example by integrating into higher-order separation logics, like for example [60]). Instead, we aim for a verification approach that balances expressivity and usability—an approach that is expressive enough to reason about real-world concurrent programs that follow some protocol, while being implementable into automated code verifiers; in this case, VerCors and Viper.

Related concurrency verifiers are SmallfootRG [61], VeriFast [7], CIVL [62], THREADER [63] and Viper [9,10]; the latter tool is used as the main back-end of VerCors. SmallfootRG is a memory-safety verifier based on RGSep. VeriFast is a rich toolset for verifying (multi-threaded) Java and C programs using separation logic. The CIVL framework can reason about race-freedom and functional correctness of MPI programs written in C [64,65]. The reasoning is done via bounded model checking combined with symbolic execution. THREADER is an automated verifier for multi-threaded C, based on model checking and counterexample-guided abstraction refinement.

One approach that is particularly noteworthy is the one of Penninckx et al. [66], who propose a logic to specify and verify input/output (I/O) behaviour of (sequential) programs. The logic has been implemented in VeriFast. In this approach the I/O behaviour of programs is specified essentially as a Petri Net. Its assertion language has constructs to specify I/O permission tokens, and its proof system has inference rules that allow reducing a Petri Net specification alongside the structure of the program, similar to our approach. However, their specification/verification strategy is to make predictions about the behaviour of the environment (which may or may not turn out true), by specifying assumptions on what the environment will input given a particular I/O operation and output, which is in contrast to our approach. In fact, with our approach one could analyse and use process-algebraic models together with an extra process that models an environment, to achieve stronger reasoning capabilities.

Apart from the proposed technique, VerCors also allows using process algebraic abstractions as *histories* [67,68]. Also related in this respect are the time-stamped histories of [69], which records atomic state changes in concurrent programs as a history, which are, likewise to our approach, handled as resources in the logic. However, history recording is only suitable for terminating programs.

There is also related earlier work on using process-algebraic abstractions to reason about message passing distributed programs [17]. This work introduces the assertional processes $?(\cdot)$ as well as process-algebraic summation, as they are used in this article. In fact, this article merges the core ideas of [16,17] into a single logical framework, to make the original work of [16] more general.

Finally, there is a lot of general work on proving linearisability [70–72], which essentially allows reasoning about fine-grained concurrency by using sequential verification techniques. Our technique, as well as the history-based technique of [67] uses process algebraic linearisation to do so.

7. Conclusions

To reason effectively about realistic concurrent and distributed software, we have presented a verification technique that performs the reasoning at a suitable level of abstraction that hides irrelevant implementation details, is scalable to realistic programs by being modular and compositional, and is practical by being supported by automated tools. The approach is expressive enough to allow reasoning about realistic software as is demonstrated by the case study as well as by [15], and can be implemented as part of an automated deductive SMT-based program verifier, viz. VerCors. The proof system underlying our technique has mechanically been proven sound using Coq. Our technique is therefore supported by a strong combination of theoretical justification and practical usability.

This article extends [16], which we considered to be the beginning of a comprehensive verification framework that aims to capture many different concurrent and distributed programming paradigms. This extended version makes a step forward towards such a framework, by unifying the core ideas of [16–18], thereby generalising the original framework, most importantly to include assertional processes.

We are currently further investigating the use of mCRL2 to reason algorithmically about program abstractions, e.g., [73]. We are looking in particular into distributed programs that communicate over channels, since process algebra have been used extensively to model such programs (see for example the work and uses of the π-calculus). We are also (somewhat more passively) looking into whether Ivy would be a suitable tool to reason about our process specifications, possibly with mild adaptions. Moreover, we are planning to investigate the preservation of liveness properties in addition to safety.

Author Contributions: Investigation, W.O., D.G. and M.H. All authors have read and agreed to the published version of the manuscript.

Funding: The third author is supported by the NWO VICI 639.023.710 Mercedes project.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Feng, X.; Ferreira, R.; Shao, Z. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *Proceedings of the European Symposium on Programming (ESOP), Braga, Portugal, 24 March–1 April 2007*; De Nicola, R., Ed.; Springer: Berlin, Germany, 2007; pp. 173–188.
2. Dinsdale-Young, T.; Dodds, M.; Gardner, P.; Parkinson, M.; Vafeiadis, V. Concurrent Abstract Predicates. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Maribor, Slovenia, 21–25 June 2010*; LNCS; D'Hondt, T., Ed.; 2010; Volume 6183, pp. 504–528.
3. Turon, A.; Dreyer, D.; Birkedal, L. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-Order Concurrency. In Proceedings of the 2013 International Conference on Functional Programming (ICFP), Boston, MA, USA, 25–27 September 2013; pp. 377–390.
4. Nanevski, A.; Ley-Wild, R.; Sergey, I.; Delbianco, G. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In *Proceedings of the European Symposium on Programming (ESOP), Grenoble, France, 5–13 April 2014*; Shao, Z., Ed.; Springer: Berlin/Heidelberg, Germany, 2014; pp. 290–310.
5. Rocha Pinto, P.d.; Dinsdale-Young, T.; Gardner, P. TaDA: A Logic for Time and Data Abstraction. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Uppsala, Sweden, 28 July–1 August 2014*; LNCS; Jones, R., Ed.; Springer: Berlin/Heidelberg, Germany, 2014; pp. 207–231.
6. Sergey, I.; Wilcox, J.; Tatlock, Z. Programming and Proving with Distributed Protocols. *Princ. Programm. Lang.* **2017**, *2*, 1–30.
7. Jacobs, B.; Smans, J.; Philippaerts, P.; Vogels, F.; Penninckx, W.; Piessens, F. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the NASA Formal Methods (NFM), Pasadena, CA, USA, 18–20 April 2011*; Bobaru, M., Havelund, K., Holzmann, G., Joshi, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 41–55.
8. Blom, S.; Darabi, S.; Huisman, M.; Oortwijn, W. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Proceedings of the International Conference on Integrated Formal Methods (iFM), Torino, Italy, 20–22 September 2017*; LNCS; Polikarpova, N., Schneider, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10510, pp. 102–110.
9. Juhasz, U.; Kassios, I.; Müller, P.; Novacek, M.; Schwerhoff, M.; Summers, A. *Viper: A Verification Infrastructure for Permission-Based Reasoning*. Technical Report; ETH Zurich: Zurich, Switzerland, 2014.
10. Müller, P.; Schwerhoff, M.; Summers, A. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation (VMCAI), St. Petersburg, FL, USA, 17–19 January 2016*; Jobstmann, B., Leino, K., Eds.; Springer: Berlin/Heidelberg, Germany, 2016; pp. 41–62.
11. INRIA—The Coq Webpage. Available online: https://coq.inria.fr (accessed on 4 June 2020).
12. Bertot, Y.; Castran, P. *Interactive Theorem Proving and Program Development: Coq'Art the Calculus of Inductive Constructions*, 1st ed.; Springer: Berlin/Heidelberg, Germany, 2010.

13. Nipkow, T.; Wenzel, M.; Paulson, L. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*; Springer: Berlin, Germany, 2002. doi:10.1007/3-540-45949-9.
14. Oortwijn, W.; Blom, S.; Gurov, D.; Huisman, M.; Zaharieva-Stojanovski, M. An Abstraction Technique for Describing Concurrent Program Behaviour. In *Proceedings of the Verified Software: Theories, Tools, and Experiments (VSTTE), Heidelberg, Germany, 22–23 July 2017*; Paskevich, A., Wies, T., Eds.; LNCS; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10712, pp. 191–209.
15. Oortwijn, W.; Huisman, M. Formal Verification of an Industrial Safety-Critical Traffic Tunnel Control System. In *Proceedings of the Integrated Formal Methods (iFM), Bergen, Norway, 2–6 December 2019*; LNCS; Ahrendt, W., Tapia Tarifa, S.L., Eds.; Springer: Berlin, Germany, 2019;
16. Oortwijn, W.; Gurov, D.; Huisman, M. Practical Abstractions for Automated Verification of Shared-Memory Concurrency. In *Proceedings of the Verification, Model Checking, and Abstract Interpretation (VMCAI), New Orleans, LA, USA, 19–21 January 2020*; Beyer, D., Zufferey, D., Eds.; Springer International Publishing: Cham, Germany, 2020; pp. 401–425.
17. Oortwijn, W.; Huisman, M. Practical Abstractions for Automated Verification of Message Passing Concurrency. In *Proceedings of the Integrated Formal Methods (iFM), Bergen, Norway, 2–6 December 2019*; LNCS; Ahrendt, W., Tapia Tarifa, S.L., Eds.; Springer: Berlin, Germany, 2019; To appear.
18. Oortwijn, W. Deductive Techniques for Model-Based Concurrency Verification. Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 2019. doi:10.3990/1.9789036548984.
19. Supplementary Material for this Article. Available online: https://vercors.ewi.utwente.nl/csl-abstractions/ (accessed on 4 June 2020).
20. Aldini, A.; Bernardo, M.; Corradini, F. *A Process Algebraic Approach to Software Architecture Design*; Springer Science & Business Media: Berlin, Germany, 2010.
21. Groote, J.; Mousavi, M. *Modeling and Analysis of Communicating Systems*; MIT Press: Cambridge, MA, USA, 2014.
22. Lamport, L. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Softw. Eng.* **1977**, *SE-3*, 125–143. doi:10.1109/TSE.1977.229904.
23. Brookes, S. A Semantics for Concurrent Separation Logic. *Theor. Comput. Sci.* **2007**, *375*, 227–270.
24. O'Hearn, P. Resources, Concurrency and Local Reasoning. *Theor. Comput. Sci.* **2007**, *375*, 271–307.
25. Bunte, O.; Groote, J.; Keiren, J.; Laveaux, M.; Neele, T.; Vink, E.d.; Wesselink, W.; Wijs, A.; Willemse, T. The mCRL2 Toolset for Analysing Concurrent Systems. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Prague, Czech Republic, 6–11 April 2019*; Vojnar, T., Zhang, L., Eds.; Springer: Berlin, Germany, 2019; pp. 21–39.
26. Owicki, S.; Gries, D. An Axiomatic Proof Technique for Parallel Programs. *Acta Inform.* **1975**, *6*, 319–340. doi:10.1007/BF00268134.
27. Rocha Pinto, P.D.; Dinsdale-Young, T.; Gardner, P. Steps in Modular Specifications for Concurrent Modules. In Proceedings of the Mathematical Foundations of Programming Semantics (MFPS), Nijmegen, The Netherlands, 22–25 June 2015.
28. Jones, C. Tentative Steps Toward a Development Method for Interfering Programs. *Trans. Programm. Lang. Syst.* **1983**, *5*, 596–619.
29. Jung, R.; Krebbers, R.; Birkedal, L.; Dreyer, D. Higher-Order Ghost State. In Proceedings of the International Conference on Functional Programming (ICFP), Nara, Japan, 18–24 September 2016; Volume 51, pp. 256–269.
30. Bergstra, J.; Klop, J. Process algebra for Synchronous Communication. *Inf. Control* **1984**, *60*, 109–137.
31. Moller, F. The Importance of the Left Merge Operator in Process Algebras. In *Proceedings of the Automata, Languages and Programming (ICALP), Warwick, UK, 16–20 July 1990*; Paterson, M., Ed.; Springer: Berlin, Germany, 1990; pp. 752–764.
32. Fokkink, W.; Zantema, H. Basic Process Algebra with Iteration: Completeness of its Equational Axioms. *Comput. J.* **1994**, *37*, 259–267.
33. Baeten, J. *Process Algebra with Explicit Termination*; Department of Mathematics and Computing Science, Eindhoven University of Technology: Eindhoven, The Netherlands, 2000.
34. Vafeiadis, V. Concurrent separation logic and operational semantics. In Proceedings of the Mathematical Foundations of Programming Semantics (MFPS), Pittsburgh, PA, USA, 25–28 May 2011; Volume 276, pp. 335–351.

35. Reynolds, J. Separation Logic: A Logic for Shared Mutable Data Structures. In Proceedings of the Logic in Computer Science (LICS), Copenhagen, Denmark, 22–25 July 2002; pp. 55–74. doi:10.1109/LICS.2002.1029817.
36. Boyland, J. Checking Interference with Fractional Permissions. In *Proceedings of the Static Analysis (SAS), San Diego, CA, USA, 11–13 June 2003*; LNCS; Cousot, R., Ed.; Springer: Berlin, Germany, 2003; Volume 2694, pp. 55–72.
37. Bornat, R.; Calcagno, C.; O'Hearn, P.; Parkinson, M. Permission Accounting in Separation Logic. In Proceedings of the Principles of Programming Languages (POPL), Long Beach, CA, USA, 12–14 January 2005; pp. 259–270.
38. O'Hearn, P.; Yang, H.; Reynolds, J. Separation and Information Hiding. In Proceedings of the Principles of Programming Languages (POPL), Venice, Italy, 14–16 January 2004; ACM: New York, NY, USA, 2004; pp. 268–280.
39. Roever, W.D.; Engelhardt, K.; Buth, K. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*; Cambridge University Press: Cambridge, UK, 1998; Volume 47.
40. Appel, A.; Blazy, S. Separation Logic for Small-Step CMINOR. In *Theorem Proving in Higher Order Logics (TPHOLs)*; Schneider, K., Brandt, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 5–21.
41. Hobor, A. Oracle Semantics. Ph.D. Thesis, Princeton University: Princeton, NJ, USA, 2008.
42. Hobor, A.; Appel, A.; Nardelli, F. Oracle Semantics for Concurrent Separation Logic. In *Proceedings of the Programming Languages and Systems (ESOP), Budapest, Hungary, 29 March–6 April 2008*; Drossopoulou, S., Ed. Springer: Berlin/Heidelberg, Germany, 2008; pp. 353–367.
43. Appel, A.; Melliès, P.; Richards, C.; Vouillon, J. A Very Modal Model of a Modern, Major, General Type System. In Proceedings of the Principles of Programming Languages (POPL), Nice, France, 17–19 January 2007; ACM: New York, NY, USA, 2007; pp. 109–122. doi:10.1145/1190216.1190235.
44. Usenko, Y. *Linearization in μCRL*; Technische Universiteit Eindhoven: Eindhoven, The Netherlands, 2002.
45. Padon, O.; McMillan, K.; Panda, A.; Sagiv, M.; Shoham, S. Ivy: Safety Verification by Interactive Generalization. In Proceedings of the Programming Language Design and Implementation (PLDI), Santa Barbara, CA, USA, 13–17 June 2016; ACM: New York, NY, USA, 2016; pp. 614–630. doi:10.1145/2908080.2908118.
46. Oortwijn, W.; Blom, S.; Huisman, M. Future-based Static Analysis of Message Passing Programs. In *Programming Language Approaches to Concurrency- & Communication-cEntric Software (PLACES)*; Open Publishing Association: Waterloo, Australia, 2016, pp. 65–72.
47. Leino, K.; Müller, P.; Smans, J. Verification of Concurrent Programs with Chalice. In Proceedings of the Foundations of Security Analysis and Design (FOSAD), Bertinoro, Italy, 30 August–4 September 2009; Volume 5705, pp. 195–222.
48. Parkinson, M.; Summers, A. The Relationship between Separation Logic and Implicit Dynamic Frames. In *Proceedings of the European Symposium on Programming (ESOP), Saarbrücken, Germany, 26 March–3 April 2011*; Barthe, G., Ed.; Springer: Berlin, Germany, 2011; pp. 439–458.
49. Joosten, S.; Oortwijn, W.; Safari, M.; Huisman, M. An Exercise in Verifying Sequential Programs with VerCors. In *Proceedings of the Formal Techniques for Java-like Programs (FTfJP), Amsterdam, The Netherlands, 16 July 2018*; Summers, A., Ed.; ACM: New York, NY, USA, 2018.
50. Blom, S.; Darabi, S.; Huisman, M. Verification of Loop Parallelisations. In *Proceedings of the Fundamental Approaches to Software Engineering (FASE), London, UK, 11–18 April 2015*; LNCS; Egyed, A.; Schaefer, I., Eds.; Springer: Berlin, Germany, 2015; Volume 9033, pp. 202–217.
51. Svendsen, K.; Birkedal, L. Impredicative Concurrent Abstract Predicates. In *Proceedings of the European Symposium on Programming (ESOP), Grenoble, France, 5–13 April 2014*; LNCS; Shao, Z., Ed.; Springer: Berlin, Germany, 2014; Volume 8410, pp. 149–168.
52. Svendsen, K.; Birkedal, L.; Parkinson, M. Modular Reasoning about Separation of Concurrent Data Structures. In P*roceedings of the European Symposium on Programming (ESOP), Rome, Italy, 16–24 March 2013*; Felleisen, M., Gardner, P., Eds.; Springer: Berlin, Germany, 2013; pp. 169–188.
53. Feng, X. Local Rely-Guarantee Reasoning. In Proceedings of the Principles of Programming Languages (POPL), Savannah, Georgia, USA, 21–23 January 2009; ACM: New York, NY, USA, 2009; Volume 44, pp. 315–327.

54. Jung, R.; Swasey, D.; Sieczkowski, F.; Svendsen, K.; Turon, A.; Birkedal, L.; Dreyer, D. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In Proceedings of the Principles of Programming Languages (POPL), Mumbai, India, 12–18 January 2015; ACM: New York, NY, USA, 2015; pp. 637–650.
55. Vafeiadis, V.; Parkinson, M. A Marriage of Rely/Guarantee and Separation Logic. In *Proceedings of the International Conference on Concurrency Theory (CONCUR), Lisbon, Portugal, 4–7 September 2007*; Caires, L., Vasconcelos, V., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 256–271.
56. Krebbers, R.; Jung, R.; Bizjak, A.; Jourdan, J.; Dreyer, D.; Birkedal, L. The Essence of Higher-Order Concurrent Separation Logic. In *Proceedings of the European Symposium on Programming (ESOP), Uppsala, Sweden, 22–29 April 2017*; LNCS; Yang, H., Ed.; Springer: Berlin/Heidelberg, Germany, 2017; Volume 10201, pp. 696–723.
57. Villard, J.; Lozes, É.; Calcagno, C. Proving Copyless Message Passing. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS), Seoul, Korea, 14–16 December 2009*; Hu, Z., Ed.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 194–209.
58. Honda, K.; Vasconcelos, V.; Kubo, M. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *Proceedings of the European Symposium on Programming (ESOP), Lisbon, Portugal, 28 March–4 April 1998;* Hankin, C., Ed.; Springer: Berlin/Heidelberg, Germany, 1998; pp. 122–138.
59. Summers, A.; Müller, P. Actor Services—Modular Verification of Message Passing Programs. In *Proceedings of the European Symposium on Programming (ESOP), Eindhoven, The Netherlands, 2–8 April 2016*; Thiemann, P., Ed.; Springer: Berlin/Heidelberg, Germany, 2016; pp. 699–726.
60. Hinrichsen, J.; Bengtson, J.; Krebbers, R. Actris: Session-Type Based Reasoning in Separation Logic. *Proc. ACM Programm. Lang.* **2019**, *4*, 1–30. doi:10.1145/3371074.
61. Calcagno, C.; Parkinson, M.; Vafeiadis, V. Modular Safety Checking for Fine-Grained Concurrency. In *Proceedings of the Static Analysis (SAS), Kongens Lyngby, Denmark, 22–24 August 2007*; Nielson, H., Filé, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 233–248.
62. Siegel, S.; Zheng, M.; Luo, Z.; Zirkel, T.; Marianiello, A.; Edenhofner, J.; Dwyer, M.; Rogers, M. CIVL: The Concurrency Intermediate Verification Language. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Austin, TX, USA, 15–20 November 2015; ACM: New York, NY, USA, 2015, p. 61.
63. Gupta, A.; Popeea, C.; Rybalchenko, A. Threader: A Constraint-Based Verifier for Multi-threaded Programs. In *Proceedings of the International Conference on Computer Aided Verification (CAV), Snowbird, UT, USA, 14–20 July 2011*; Gopalakrishnan, G., Qadeer, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 412–417. doi:10.1007/978-3-642-22110-1_32.
64. Zheng, M.; Rogers, M.; Luo, Z.; Dwyer, M.; Siegel, S. CIVL: Formal Verification of Parallel Programs. In Proceedings of the Automated Software Engineering (ASE), Lincoln, NE, USA, 9–13 November 2015; pp. 830–835. doi:10.1109/ASE.2015.99.
65. Luo, Z.; Zheng, M.; Siegel, S. Verification of MPI programs using CIVL. In Proceedings of the 24th European MPI Users' Group Meeting (EuroMPI), Chicago, IL, USA, 25–28 September 2017.
66. Penninckx, W.; Jacobs, B.; Piessens, F. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *Proceedings of the European Symposium on Programming (ESOP), London, UK, 11–18 April 2015*; Vitek, J., Ed.; Springer: Berlin/Heidelberg, Germany, 2015; pp. 158–182.
67. Blom, S.; Huisman, M.; Zaharieva-Stojanovski, M. History-Based Verification of Functional Behaviour of Concurrent Programs. In *Proceedings of the Software Engineering and Formal Methods (SEFM), York, UK, 7–11 September 2015*; LNCS; Calinescu, R., Rumpe, B., Eds. Springer: Berlin/Heidelberg, Germany, 2015; Volume 9276, pp. 84–98.
68. Zaharieva-Stojanovski, M. Closer to Reliable Software: Verifying Functional Behaviour of Concurrent Programs. Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 2015.
69. Sergey, I.; Nanevski, A.; Banerjee, A. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *Proceedings of the European Symposium on Programming (ESOP), London, UK, 11–18 April 2015*; LNCS; Vitek, J., Ed.; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9032, pp. 333–358.
70. Herlihy, M.; Wing, J. Linearizability: A Correctness Condition for Concurrent Objects. *Trans. Programm. Lang. Syst.* **1990**, *12*, 463–492.

71. Vafeiadis, V. Automatically Proving Linearizability. In *Proceedings of the Computer-Aided Verification (CAV), Edinburgh, UK, 15–19 July 2010*; Touili, T., Cook, B., Jackson, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 450–464. doi:10.1007/978-3-642-14295-6_40.
72. Krishna, S.; Shasha, D.; Wies, T. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. *Princ. Programm. Lang.* **2017**, *2*, 1–31.
73. Neele, T.; Willemse, T.; Groote, J. Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting. In *Proceedings of the Formal Aspects of Component Software (FACS), Pohang, Korea, 10–12 October 2018*; Bae, K., Ölveczky, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2018; pp. 216–236.