An Abstraction Technique for Describing Concurrent Program Behaviour

Wytse Oortwijn[1], Stefan Blom[1], Dilian Gurov[2], Marieke Huisman[1], and Marina Zaharieva-Stojanovski[1]

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

Abstract. This paper presents a technique to reason about functional properties of shared-memory concurrent software by means of abstraction. The abstract behaviour of the program is described using process algebras. In the program we indicate which concrete atomic steps correspond to the actions that are used in the process algebra term. Each action comes with a specification that describes its effect on the shared state. Program logics are used to show that the concrete program steps adhere to this specification. Separately, we also use program logics to prove that the program behaves as described by the process algebra term. Finally, via process algebraic reasoning we derive properties that hold for the program from its abstraction. This technique allows reasoning about the behaviour of highly concurrent, non-deterministic and possibly non-terminating programs. The paper discusses various verification examples to illustrate our approach. The verification technique is implemented as part of the VerCors toolset. We demonstrate that our technique is capable of verifying data- and control-flow properties that are hard to verify with alternative approaches, especially with mechanised tool support.

1 Introduction

The major challenge when reasoning about concurrent or distributed software is to come up with an appropriate abstraction that provides sufficient detail to capture the intended properties, while at the same time making verification manageable. This paper presents a new powerful abstraction approach that enables reasoning about the intended properties of the program in a purely non-deterministic setting, and can abstract code at different levels of granularity. The presentation of the abstraction technique in this paper focuses on shared-memory concurrent programs and safety properties, but many extensions may be explored, for example for distributed programs or progress properties, as sketched in the paragraph on future work. The paper illustrates our approach by discussing multiple verification examples in which we verify various data- and control-flow properties. We demonstrate that the proposed technique can be used to verify program properties that are hard to verify with alternative approaches, especially in a practical manner via mechanised tools.

To motivate our approach, consider the program shown in Figure 1. The figure shows a parallel version of the classical Euclidean algorithm for finding a

```
1  int x, y;
2
3  void threadx() {
4  |  bool stop := false;
5  |  while ¬stop do {
6  |  |  acquire lock;
7  |  |  if (x > y) { x := x − y; }
8  |  |  stop := x = y;
9  |  |  release lock;
10 |  }
11 }
12
13 void thready() {
14 |  bool stop := false;
15 |  while ¬stop do {
16 |  |  acquire lock;
17 |  |  if (y > x) { y := y − x; }
18 |  |  stop := x = y;
19 |  |  release lock;
20 |  }
21 }
22
23 int startgcd(int a, int b) {
24 |  x := a;  y := b;
25 |  init lock;
26 |  handle t1 := fork threadx();
27 |  handle t2 := fork thready();
28 |  join t1;
29 |  join t2;
30 |  destroy lock;
31 |  return x;
32 }
```

Fig. 1: A parallel implementation of the Euclidean algorithm for finding the greatest common divisor of two (positive) integers x and y.

greatest common divisor, $\texttt{gcd}(x, y)$, of two given positive integers x and y. This is done by forking two concurrent threads: one thread to decrement the value of x whenever possible, and one thread to decrement the value of y.

We are interested in verifying deductively that this program indeed computes the greatest common divisor of x and y. To accomplish this in a scalable fashion requires that our technique be *modular*, or more precisely procedure-modular and thread-modular, to allow the individual functions and threads to be analysed independently of one another. The main challenge in achieving this lies in finding a suitable way of capturing the *effect* of function calls and threads on the shared memory in a way that is independent of the other functions and threads. Our proposal is to capture these effects as *sequences of exclusive accesses* (in this example increments and decrements) to shared memory (in this example the variables x and y). We abstract such accesses into so-called *actions*, and their sequences into process algebraic terms.

In our example above we abstract the assignments $x := x - y$ and $y := y - x$ needed to decrease the values of x and y into actions decrx and decry, respectively. Action behaviour is specified by means of *contracts* consisting of a guard and an effect; the explanation of the details of this are deferred to Section 3. Using these actions, we can specify the effects of the two threads by means of the process algebra terms tx and ty, respectively, which are defined as follows:

$$\textbf{process}\ \mathsf{tx}() := \mathsf{decrx} \cdot \mathsf{tx}() + \mathsf{done} \qquad \textbf{process}\ \mathsf{ty}() := \mathsf{decry} \cdot \mathsf{ty}() + \mathsf{done}$$

Here the action done indicates termination of a process. The functional behaviour of the program can then be specified by the process pargcd defined as the term $\mathsf{tx}() \parallel \mathsf{ty}()$. Standard process algebraic reasoning can be applied to show that executing pargcd results in calculating the correct $\texttt{gcd}$.

Therefore, by proving that the implementation executes as prescribed by pargcd, we simultaneously establish its functional property of producing the correct result. The pargcd process thus describes the program behaviour.

Once the program has been specified, the access exclusiveness of the actions is verified by a suitable extension of separation logic with permission accounting [19,5]. On top of it, we develop rules that allow to prove, in a thread-local fashion, that the program indeed follows its prescribed process. The details of our technique applied to the above program are presented in Section 3.

In previous work [4,27] we developed an approach that records the actions of a concurrent program as the program executes. Reasoning with this approach is only suitable for terminating programs, and occurs at the end of its execution, requiring the identification of repeating patterns. In contrast, the current approach requires a process algebra term upfront that describes the patterns of atomic concurrent actions, which allows the specification of functional behaviour of reactive, non-terminating programs. For instance, we can verify properties such as "the values of the shared variables x and y will be equal infinitely often", expressed in LTL by the formula $\Box\Diamond(x = y)$, of a program that forks separate threads to modify x and y, similarly to the above parallel GCD program.

Compared to many of the other modern logics to reason about concurrent programs, such as CAP [9], CaReSL [26], Iris [17], and TaDA [7], our approach does the abstraction at a different level. Our abstraction connects program code with individual actions, while these other approaches essentially encode an abstract state machine, describing how program steps evolve from one abstract program state to the next abstract program state, and explicitly consider the changes that could be made by the thread environment. As a result, in our approach the global properties are specified in a way that is independent of the program implementation. This makes it easier for non-experts to understand the program specification. The main contributions of this paper are:

- An abstraction technique to specify and verify the behaviour of possibly non-terminating, shared-memory concurrent programs, where the abstractions are implementation-independent and may be non-deterministic;
- A number of verification examples that illustrate our approach and can mechanically be verified via the VerCors toolset; and thus
- Tool support for our model-based reasoning approach.

The remainder of this paper is organised as follows. Section 2 provides a brief background on separation logic and process algebras. Then, Section 3 illustrates in more detail how abstract models are used in the verification of the parallel GCD example. Section 4 elaborates on the proof rules as they are used by the VerCors tool set. Section 5 discusses two more verification examples that apply our approach: verifying a concurrent counter and verifying a locking protocol. Finally, Section 6 discusses related work and Section 7 concludes.

2 Background

Our program logic is an extension of Concurrent Separation Logic (CSL) with permission accounting [22,19,1]. The main difference with classical Hoare logic

is that each allocated heap location is associated with a fractional permission π, modelled as a rational number in the range $(0, 1]$ [6,5]. By allocating a heap location ℓ, the allocating thread gets *full* ownership over ℓ, represented by the $\ell \overset{1}{\hookrightarrow} v$ predicate. The $\overset{1}{\hookrightarrow}$ predicate gives writing permission to the specified heap location, whereas $\overset{\pi}{\hookrightarrow}$ for $\pi < 1$ only gives reading permission. The $\overset{\pi}{\hookrightarrow}$ predicates may be split and merged along π, so that $\ell \overset{\pi_1}{\hookrightarrow} v * \ell \overset{\pi_2}{\hookrightarrow} v \Leftrightarrow \ell \overset{\pi_1+\pi_2}{\hookrightarrow} v$. In this case, $\Leftrightarrow$ can be read as "splitting" from right to left, or "merging" from left to right. The $*$ connector is the *separating conjunction*; the assertion $\mathcal{P} * \mathcal{Q}$ means that the heap can be split into two disjoint parts, so that one part satisfies the assertion $\mathcal{P}$ and the other part satisfies $\mathcal{Q}$. CSL allows (splitted) points-to predicates that are separated via the $*$-connective to be distributed over concurrent threads (under certain conditions), thereby allowing to reason about race freedom and about functional behaviour of concurrent programs.

2.1 Dynamic locking

To reason about dynamically allocated locks we use the program logic techniques proposed by Gotsman et al. [11]. Our language includes the **init** $\mathcal{L}$ statement, which initialises a new lock associated with the *lock label* $\mathcal{L}$. The program logic requires that a *resource invariant* is specified for each initialised lock. A resource invariant is a predicate that expresses the ownership predicates protected by the lock. In the program logic a $\mathsf{Lock}_1(\mathcal{L})$ predicate is produced by **init** $\mathcal{L}$, which represents the knowledge of the existence of a lock labelled $\mathcal{L}$ and this predicate is required to obtain the lock later. Obtaining a lock labelled $\mathcal{L}$ is done via the **acquire** $\mathcal{L}$ statement which, on the program logic level, consumes the $\mathsf{Lock}_\pi(\mathcal{L})$ predicate and exchanges it for the resource invariant that is associated to $\mathcal{L}$. Releasing a lock is done via the **release** $\mathcal{L}$ statement, which has the reverse effect: it takes the resource invariant of $\mathcal{L}$ and exchanges it for $\mathsf{Lock}_\pi(\mathcal{L})$. The **destroy** $\mathcal{L}$ statement destroys the lock $\mathcal{L}$ and thereby consumes $\mathsf{Lock}_1(\mathcal{L})$ in the program logic and gives back the resource invariant associated to $\mathcal{L}$.

2.2 Process algebra terms

The abstract models we use to reason about programs are represented as process algebra terms. A subset of the μCRL [13,12] language is used as a suitably expressive process algebra with data. The basic primitives are actions, each representing an indivisible process behaviour. Processes are defined by combining actions and recursive process calls, which both may be parameterised by data. Process algebra terms have the following structure:

$$P, Q \ ::= \ \varepsilon \mid \delta \mid a(\overline{E}) \mid p(\overline{E}) \mid P \cdot Q \mid P + Q \mid P \parallel Q \mid \textbf{if}\ B\ \textbf{then}\ P\ \textbf{else}\ P$$

where E are arithmetic expressions, B are Boolean expressions, a are action labels, and p are process labels. With $\overline{E}$ we mean a sequence of expressions.

The empty process is denoted ε and the deadlock process by δ. The process $a(\overline{E})$ is an action call and $p(\overline{E})$ a recursive process invocation, with $\overline{E}$ the argument sequence. Two process terms P and Q may compose either sequentially

$P \cdot Q$ or alternatively $P + Q$. The parallel composition $P \parallel Q$ allows the actions of P and Q to be interleaved during execution. The conditional construct **if** B **then** P **else** Q resembles the classical "if-then-else"; it yields either P or Q, depending on the result of evaluating the expression B.

3 Motivating Example

This section demonstrates our approach by verifying functional correctness of the parallel GCD verification example that was discussed in the introduction. With *functional correctness* we mean verifying that, after the program terminates, the *correct* value has been calculated. In this example, the correct value is the mathematical GCD of the two (positive) values given as input to the algorithm.

Our approach uses the following steps:

(1) *Actions* and their associated *guards* and *effects* are defined that describe in what ways the program is allowed to make updates to shared memory.
(2) The actions are composed into *processes* by using the process algebraic connectives discussed in Section 2. These processes determine the desired behaviour of (parts of) the concrete program. Notably, processes that are composed in parallel correspond to forked threads in the program.
(3) All defined processes that have a *contract* are verified. Concretely, we automatically verify whether the postconditions of processes can be ensured by all traces that start from a state in which the precondition is satisfied.
(4) Finally we verify that every thread forked by the program *behaves as specified* by the process algebraic specification. If this is the case, the verification results that are established from *(3)* can be used in the program logic.

Tool support for model-based reasoning is provided as part of the VerCors verification tool set [3,2]. The VerCors tool set aims to verify programs under various concurrency models, notably heterogeneous and homogeneous concurrency, written in high-level programming languages such as Java and C. Although most of the examples presented in this paper have been worked out and verified in PVL, the Prototypal Verification Language that we use to prototype new verification features, tool support is also provided for both Java and C.

All verification examples presented in this paper have been verified with the VerCors tool set. Moreover, all example programs are accessible via an online interface to VerCors, available at `http://utwente.nl/vercors`.

Parallel GCD. We demonstrate our model-based reasoning approach by capturing the functional behaviour of a parallel GCD algorithm. The parallel GCD verification problem is taken from the VerifyThis challenge held at ETAPS 2015[3] and considers a parallel version of the classical Euclidean algorithm.

The standard Euclidean algorithm is defined as a function `gcd` such that, given two positive integers x and y, $\texttt{gcd}(x, x) = x$, $\texttt{gcd}(x, y) = \texttt{gcd}(x - y, y)$

[3] see also `http://etaps2015.verifythis.org`.

```
1  int x, y;
2
3  guard x > 0 ∧ y > x
4  effect x = old(x) ∧ y = old(y) − old(x)
5  action decrx;
6
7  guard y > 0 ∧ x > y
8  effect x = old(x) − old(y) ∧ y = old(y)
9  action decry;
10
11 guard x = y
12 action done;
13
14 process tx() := decrx · tx() + done;
15 process ty() := decry · ty() + done;
16
17 requires x > 0 ∧ y > 0
18 ensures x = y
19 ensures x = gcd(old(x), old(y))
20 process pargcd() := tx() ∥ ty();
```

Fig. 2: The processes used for the *parallel GCD* verification example. Three actions are used: decrx, decry, and done; the first two actions capture modifications made to the (shared) variables x and y, and done indicates termination.

if $x > y$, and $\texttt{gcd}(x, y) = \texttt{gcd}(x, y - x)$ if $y > x$. The parallel version of this algorithm uses two concurrent threads: the first thread continuously decrements the value of x when $x > y$, the second thread continuously decrements the value of y when $y > x$, and this process continues until x and y converge to the gcd of the two original input values. Model-based reasoning is used to describe the interleaving of the concurrent threads and to prove functional correctness of the parallel algorithm in an elegant way. Figure 2 presents the setup of the pargcd process, which models the behaviour of a parallel GCD algorithm with respect to the two global variables x and y. The pargcd process uses three different actions, named: decrx, decry, and done. Performing the action decrx captures the effect of decreasing x, provided that $x > y$ before the action is performed. Likewise, performing decry captures the effect of decreasing y. Finally, the done action may be performed when $x = y$ and is used to indicate termination of the algorithm.

The pargcd process is defined as the parallel composition of two processes; the process tx() describes the behaviour of the thread that decreases x, and ty() describes the behaviour of the thread that decreases y. The pargcd process requires that the shared variables x and y are both positive, and ensures that both x and y contain the gcd of the original values of x and y. Proving that pargcd satisfies its contract is done via standard process algebraic reasoning: first pargcd is converted to a linear process (i.e. a process without parallel constructs), which is then analysed (e.g. via model checking) to show that every thread interleaving leads to a correct answer, in this case $\texttt{gcd}(\textbf{old}(x), \textbf{old}(y))$.

Verifying program correctness. Figure 3 shows the `startgcd` function, which is the entry point of the parallel GCD algorithm. According to `startgcd`'s contract, two positive integers must be given as input and permission is required to write to x and y. On line 8 a model is initialised and named m, which describes that all further program executions behave as specified by the pargcd process. Since pargcd is defined as the parallel composition of the processes tx and ty, its definition may be *matched* in the program code by forking two concurrent threads and giving each thread one of the components of tx() ∥ ty(). In this case,

```
1  resource lock := ∃v1, v2 : v1 > 0 ∗
2      v2 > 0 ∗ x ↪[1]_p v1 ∗ y ↪[1]_p v2;
3
4  requires a > 0 ∧ b > 0
5  ensures x = y ∧ x = gcd(a, b)
6  void startgcd(int a, int b) {
7  |  x := a; y := b;
8  |  model m := init pargcd() over x, y;
9  |  init lock;
10 |  handle t1 := fork threadx(m);
11 |  handle t2 := fork thready(m);
12 |  join t1;
13 |  join t2;
14 |  destroy lock;
15 |  finish m;
16 }
```

Fig. 3: The entry point of the *parallel GCD* algorithm. Two threads are forked and continuously decrement either x or y until $x = y$, which is when the threads converge. The functional property of actually producing a `gcd` is proven by analysing the process.

the thread executing `threadx`() continues from the process tx() and the thread executing `thready`() continues from ty(). By later joining the two threads and finishing the model by using the ghost statement **finish** (which is only possible if pargcd has been fully executed), we may establish that `startgcd` satisfies its contract. However, we still have to show that the threads executing `threadx` and `thready` behave as described by the model m.

Figure 4 shows the implementation of `threadx` and `thready`. Both procedures require a $\mathsf{Lock}_\pi(lock)$ predicate, which gives the knowledge that a lock with resource invariant labelled *lock* has been initialised, and gives the possibility to acquire this lock and therewith the associated resource invariant. Moreover, both procedures require one half of the splitted $\mathsf{Proc}_1(m, \mathsf{tx}() \parallel \mathsf{ty}())$ predicate that is established in Figure 3 as result of initialising the model on line 8.

The connection between the process and program code is made via the **action** (ghost) statements. To illustrate, in the function `threadx` the decrement of x on line 13 is performed in the context of an action block, thereby forcing the tx() process in the $\mathsf{Proc}_{1/2}$ predicate to perform the decrx action. The **guard** of decrx specifies the condition under which decrx can be executed, and the **effect** clause describes the effect on the (shared) state as result of executing decrx. Eventually, both threads execute the done action to indicate their termination.

The VerCors tool set can automatically verify the parallel GCD verification example discussed above, including the analysis of the processes.

4 Program Logic

This section shortly elaborates on the assertion language and the proof rules of our approach, as used internally by the VerCors tool set to reason about abstractions. We do not present a full formalisation, for full details we refer to [27]. Only the proof rules related to model-based reasoning are discussed.

4.1 Assertion language

Our program logic builds on standard CSL with permission accounting [6] and lock predicates [11]. The following grammar defines its assertion language:

```
1  requires Lock_π(lock)
2  requires Proc_{1/2}(m, tx())
3  ensures Lock_π(lock)
4  ensures Proc_{1/2}(m, ε)
5  void threadx(model m) {
6  |  bool stop := false;
7  |  loop-inv Lock_π(lock);
8  |  loop-inv ¬stop ⇒ Proc_{1/2}(m, tx());
9  |  loop-inv stop ⇒ Proc_{1/2}(m, ε);
10 |  while ¬stop do {
11 |  |  acquire lock;
12 |  |  if (x > y) {
13 |  |  |  action m.decrx() {
14 |  |  |  |  x := x − y;
15 |  |  |  }
16 |  |  }
17 |  |  if (x = y) {
18 |  |  |  action m.done() {
19 |  |  |  |  stop := true;
20 |  |  |  }
21 |  |  }
22 |  |  release lock;
23 |  }
24 }
```

```
1  requires Lock_π(lock)
2  requires Proc_{1/2}(m, ty())
3  ensures Lock_π(lock)
4  ensures Proc_{1/2}(m, ε)
5  void thready(model m) {
6  |  bool stop := false;
7  |  loop-inv Lock_π(lock);
8  |  loop-inv ¬stop ⇒ Proc_{1/2}(m, ty());
9  |  loop-inv stop ⇒ Proc_{1/2}(m, ε);
10 |  while ¬stop do {
11 |  |  acquire lock;
12 |  |  if (y > x) {
13 |  |  |  action m.decry() {
14 |  |  |  |  y := y − x;
15 |  |  |  }
16 |  |  }
17 |  |  if (x = y) {
18 |  |  |  action m.done() {
19 |  |  |  |  stop := true;
20 |  |  |  }
21 |  |  }
22 |  |  release lock;
23 |  }
24 }
```

Fig. 4: The implementation of the procedures used by the two threads to calculate the `gcd` of x and y. The procedure `threadx` decrements x and `thready` decrements y.

$$\mathcal{P}, \mathcal{Q} \ ::= B \mid \forall x.\mathcal{P} \mid \exists x.\mathcal{P} \mid \mathcal{P} \wedge \mathcal{Q} \mid \mathcal{P} * \mathcal{Q} \mid \mathsf{Lock}_\pi(\mathcal{L}) \mid \mathsf{Locked}_\pi(\mathcal{L}) \mid \cdots$$
$$\mid E \overset{\pi}{\hookrightarrow}_{\mathsf{n}} E \mid E \overset{\pi}{\hookrightarrow}_{\mathsf{p}} E \mid E \overset{\pi}{\hookrightarrow}_{\mathsf{a}} E \mid \mathsf{Proc}_\pi(E, p, P)$$

where E are arithmetic expressions, B are Boolean expressions, x are variables, π are fractional permissions, $\mathcal{L}$ are lock labels, and p are process labels. Note that the specification language implemented in VerCors supports more assertion constructs; we only highlight a subset to elaborate on our approach.

Instead of using a single points-to ownership predicate, like in standard CSL, our extensions require three different points-to predicates:

- The $E \overset{\pi}{\hookrightarrow}_{\mathsf{n}} E'$ predicate is the standard points-to predicate from CSL. It gives write permission to the heap location expressed by E in case $\pi = 1$, and gives read access in case $\pi \in (0, 1]$. This predicate also represents the knowledge that the heap contains the value expressed by E' at location E.
- The *process points-to predicate* $E \overset{\pi}{\hookrightarrow}_{\mathsf{p}} E'$ is similar to $\overset{\pi}{\hookrightarrow}_{\mathsf{n}}$, but indicates that the heap location at E is *bound* by an abstract model. Since all changes to this heap location must be captured by the model, the $\overset{\pi}{\hookrightarrow}_{\mathsf{p}}$ predicate *only* gives read permission to E, even when $\pi = 1$.

– The *action points-to predicate* $E \overset{\pi}{\hookrightarrow}_{\mathsf{a}} E'$ gives read- or write access to the heap location E in the context of an **action** block. As a precondition, **action** blocks require $\overset{\pi}{\hookrightarrow}_{\mathsf{p}}$ predicates for all heap locations that are accessed in their body. These predicates are then converted to $\overset{\pi}{\hookrightarrow}_{\mathsf{a}}$ predicates, which give reading permission if $\pi \in (0, 1]$, and writing permission if $\pi = 1$.

All three points-to ownership predicates can be split and merged along the associated fractional permission, to be distributed among concurrent threads:

$$E \overset{\pi_1+\pi_2}{\hookrightarrow}_t E' \;\Leftrightarrow\; E \overset{\pi_1}{\hookrightarrow}_t E' * E \overset{\pi_2}{\hookrightarrow}_t E' \qquad \text{for } t \in \{\mathsf{n}, \mathsf{p}, \mathsf{a}\}$$

Essentially, three different predicates are needed to ensure soundness of the verification approach. When a heap location ℓ becomes bound by an abstract model, its $\ell \overset{\pi}{\hookrightarrow} E$ predicate is converted to an $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{p}} E$ predicate in the program logic. As an effect, the value at ℓ cannot just be changed, since the $\overset{\pi}{\hookrightarrow}_{\mathsf{p}}$ predicate does not permit writing to ℓ (even when $\pi = 1$). However, the value at ℓ *can* be changed in the context of an action block, as the rule for action blocks in our program logic converts all affected $\overset{\pi}{\hookrightarrow}_{\mathsf{p}}$ predicates to $\overset{\pi}{\hookrightarrow}_{\mathsf{a}}$ predicates, and $\overset{\pi}{\hookrightarrow}_{\mathsf{a}}$ again allows heap writes. The intuition is that, by converting $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{p}} E$ predicates to $\ell \overset{\pi}{\hookrightarrow}_{\mathsf{a}} E$ predicates, all changes to ℓ *must* occur in the context of action blocks, and this allows us to describe all changes to ℓ as process algebra terms. Consequently, by reasoning over these process algebra terms, we may reason about all possible changes to ℓ, and our verification approach allows to use the result of this reasoning in the proof system.

The second main extension our program logic makes to standard CSL is the $\mathsf{Proc}_\pi(E, p, P)$ predicate, which represents the knowledge of the existence of an abstract model that: *(i)* is identified by the expression E, *(ii)* was initialised by invoking the process labelled p, and *(iii)* is described by the process term P. For brevity we omitted p from the annotations in all example programs, since this component is constant (it cannot be changed in the proof system). The third component P is the remaining process term that is to be "executed" (or "matched") by the program. The Proc_π predicates may be split and merged along the fractional permission and the process term, similar to the points-to ownership predicates:

$$\mathsf{Proc}_{\pi_1+\pi_2}(E, p, P_1 \parallel P_2) \;\Leftrightarrow\; \mathsf{Proc}_{\pi_1}(E, p, P_1) * \mathsf{Proc}_{\pi_2}(E, p, P_2)$$

4.2 Proof system

Figure 5 shows the proof rules for our model-based reasoning approach. For presentational purposes these rules are somewhat simplified: the rules [INIT], [FIN], and [ACT] require some extra side conditions that deal with process- and action arguments. We also omitted handing process arguments in [INIT]. More details on these proof rules can be found in [27].

The [ASS] rule allows reading from the heap, which can be done with any points-to permission predicate (that is, $\overset{\pi}{\hookrightarrow}_t$ for any permission type t). Writing to shared memory is only allowed by [MUT] with a *full* permission predicate that

$$\frac{x \notin \mathsf{fv}(E, E')}{\vdash \{\mathcal{P}[x/E'] \wedge E \overset{\pi}{\hookrightarrow}_t E'\}\, x := [E]\, \{\mathcal{P} \wedge E \overset{\pi}{\hookrightarrow}_t E'\}} \text{ [ASS]}$$

$$\frac{t \neq \mathsf{p}}{\vdash \{E \overset{1}{\hookrightarrow}_t -\}\, [E] := E'\, \{E \overset{1}{\hookrightarrow}_t E'\}} \text{ [MUT]}$$

$$\frac{B = \mathsf{precondition}(p) \qquad P = \mathsf{body}(p)}{\begin{array}{c}\vdash \{*_{i=0..n} E_i \overset{1}{\hookrightarrow}_\mathsf{n} E_i' * B\} \\ \textbf{model}\ m := \textbf{init}\ p()\ \textbf{over}\ E_0, \ldots, E_n \\ \{*_{i=0..n} E_i \overset{1}{\hookrightarrow}_\mathsf{p} E_i' * \mathsf{Proc}_1(m, p, P)\}\end{array}} \text{ [INIT]}$$

$$\frac{\mathsf{locations}(m) = (E_0, \ldots, E_n) \qquad B = \mathsf{postcondition}(p)}{\vdash \{*_{i=0..n} E_i \overset{1}{\hookrightarrow}_\mathsf{p} E_i' * \mathsf{Proc}_1(m, p, \varepsilon)\}\ \textbf{finish}\ m\ \{*_{i=0..n} E_i \overset{1}{\hookrightarrow}_\mathsf{n} E_i' * B\}} \text{ [FIN]}$$

$$\frac{\begin{array}{c}\mathsf{accessedlocs}(S) = (E_0, \ldots, E_n) \qquad B_1 = \mathsf{guard}(a) \qquad B_2 = \mathsf{effect}(a) \\ \vdash \{*_{i=0..n} E_i \overset{1}{\hookrightarrow}_\mathsf{a} E_i' * B_1\}\, S\, \{*_{i=0..n} E_i \overset{1}{\hookrightarrow}_\mathsf{a} E_i'' * B_2\}\end{array}}{\begin{array}{c}\vdash \{*_{i=0..n} E_i \overset{1}{\hookrightarrow}_\mathsf{p} E_i' * \mathsf{Proc}_\pi(m, p, a(\overline{E}) \cdot P) * B_1\} \\ \textbf{action}\ m.a(\overline{E})\ \{\ S\ \} \\ \{*_{i=0..n} E_i \overset{1}{\hookrightarrow}_\mathsf{p} E_i'' * \mathsf{Proc}_\pi(m, p, P) * B_2\}\end{array}} \text{ [ACT]}$$

Fig. 5: The simplified proof rules of all model-related specification constructs.

is *not* of type p; if the targeted heap location is bound by an abstract model, then all changes must be done in an action block (see the [ACT] rule). [INIT] handles the initialisation of a model, which on the specification level converts all affected $\overset{1}{\hookrightarrow}_\mathsf{n}$ predicates to $\overset{1}{\hookrightarrow}_\mathsf{p}$ and produces a *full* Proc_1 predicate. [FIN] handles model finalisation: it requires a fully executed Proc_1 predicate (holding the process ε) and converts all affected $\overset{1}{\hookrightarrow}_\mathsf{p}$ predicates back to $\overset{1}{\hookrightarrow}_\mathsf{n}$. Finally, [ACT] handles action blocks. If a proof can be derived for the body S of the action block that: *(i)* respects the guard and effect of the action, and *(ii)* with the $\overset{1}{\hookrightarrow}_\mathsf{p}$ predicates of all heap locations accessed in S converted to $\overset{1}{\hookrightarrow}_\mathsf{a}$, then a similar proof can be established for the entire action block. Observe that [ACT] requires and consumes the matching action call in the process term.

5 Applications of the Logic

In this section we apply our approach on two more verification problems: *(i)* a concurrent program in which multiple threads increase a shared counter by one (see Section 5.1); and *(ii)* verifying control-flow properties of a fine-grained lock implementation (see Section 5.2). Also some interesting variants on these problems are discussed. For example *(i)* we verify the functional property that, after the program terminates, the correct value has been calculated. For *(ii)* we

```
1  int counter;
2
3  void worker() {
4  |  atomic {
5  |  |  counter := counter + 1;
6  |  }
7  }
```

```
8  void program(int n) {
9  |  counter := n;
10 |  handle t1 = fork worker();
11 |  handle t2 = fork worker();
12 |  join t1;
13 |  join t2;
14 }
```

Fig. 6: The concurrent counting example program, where two threads forked by `program` increment the shared integer *counter*.

verify that clients of the lock adhere to the intended locking protocol and thereby avoid misusing the lock.

5.1 Concurrent counting

Our second example considers a *concurrent counter*: a program where two threads concurrently increment a common shared integer. The basic algorithm is given in Figure 6. The goal is to verify that `program` increments the original value of *counter* by two, given that it terminates. However, providing a specification for `worker` can be difficult, since no guarantees to the value of *counter* can be given after termination of `worker`, as it is used in a concurrent environment.

Existing verification approaches for this particular example [8] mostly require *auxiliary state*, a form of *rely/guarantee reasoning*, or, more recently, *concurrent abstract predicates*, which may blow-up the amount of required specifications and are not always easy to use. We show how to verify the program of Figure 6 via our model-based abstraction approach. Later, we show how our techniques may be used on the same program but generalised to n threads.

Our approach is to protect all changes to *counter* by a process that we name parincr. The parincr process is defined as the parallel composition incr $\parallel$ incr of two processes that both execute the incr action once. Performing incr has the effect of incrementing *counter* by one. From a process algebraic point of view it is easy to see that parincr satisfies its contract: every possible trace of parincr indeed has the effect of increasing *counter* by two, and this can automatically be verified. We use this result in the verification of `program` by using model-based reasoning. In particular, we may instantiate parincr as a model m, split along its parallel composition, and give each forked thread a fraction of the splitted Proc predicate. The interface specification of the `worker` procedure thus becomes:

$$\{\mathsf{Proc}_\pi(m, \mathsf{incr})\}\ \texttt{worker}(m)\ \{\mathsf{Proc}_\pi(m, \varepsilon)\}$$

An annotated version of the concurrent counting program is presented in Figure 7. The **atomic** statement is used as a construct for statically-scoped locking; for simplicity we assume that writing permissions for *counter* are maintained by its resource invariant. Indeed, by showing that both threads execute the incr action, the established result of incrementing *counter* by 2 can be concluded.

```
1  int counter;
2
3  effect counter = old(counter) + 1;
4  action incr;
5
6  ensures counter = old(counter) + 2;
7  process parincr() := incr ‖ incr;
8
9  requires Proc_π(m, incr);
10 ensures Proc_π(m, ε);
11 void worker(model m) {
12 |  atomic {
13 |  |  action m.incr {
14 |  |  |  counter := counter + 1;
15 |  |  }
16 |  }
17 }
18
19 ensures counter = c + 2;
20 void program(int c) {
21 |  counter := c;
22 |  model m := parincr();
23 |  handle t1 := fork worker(m);
24 |  handle t2 := fork worker(m);
25 |  join t1;
26 |  join t2;
27 |  finish m;
28 }
```

Fig. 7: Definition of the parincr process that models two concurrent threads performing an atomic incr action, and the required annotations for `worker` and `program`.

```
1  requires n ≥ 0;
2  requires Proc_π(m, parincr(n));
3  ensures Proc_π(m, ε);
4  void spawn(model m, int n) {
5  |  if (n > 0) {
6  |  |  handle t := fork worker(m);
7  |  |  spawn(m, n − 1);
8  |  |  join t;
9  |  }
10 }
11
12 requires n ≥ 0;
13 ensures counter = c + n;
14 void program(int c, int n) {
15 |  counter := c;
16 |  model m := parincr(n);
17 |  spawn(m, n);
18 |  finish m;
19 }
```

Fig. 8: Generalisation of the concurrent counting verification problem, where `program` forks n threads using the recursive `spawn` procedure. Each thread executes the `worker` procedure and therewith increments the value of *counter* by one.

Generalised concurrent counting. The interface specification of worker is generic enough to allow a generalisation to n threads. Instead of the parincr process as presented in Figure 7 one could consider the following process, which essentially encodes the process "incr ‖ ⋯ ‖ incr" (n times) via recursion:

requires $n \geq 0$;
ensures *counter* = **old**(*counter*) + n;
process parincr(**int** n) := **if** $n > 0$ **then** incr ‖ parincr($n - 1$) **else** ε;

Figure 8 shows the generalised version of the concurrent counting program, in which we reuse the incr action and the `worker` procedure from Figure 7. Here `program` takes an extra parameter n that determines the number of threads to be spawned. The `spawn` procedure has been added to spawn the n threads. This procedure is recursive to match the recursive definition of the parincr(n) process.

Again, each thread executes the `worker` procedure. We verify that after running `program` the value of *counter* has increased by n.

On the level of processes we may automatically verify that each trace of the process $\mathsf{parincr}(n)$ is a sequence of n consecutive incr actions. As a consequence, from the effects of incr we can verify that $\mathsf{parincr}(n)$ increases *counter* by n. On the program level we may verify that $\texttt{spawn}(m, n)$ fully executes according to the $\mathsf{parincr}(n)$ process. To clarify, on line 6 the definition of $\mathsf{parincr}(n)$ can be unfolded to $\mathsf{incr} \parallel \mathsf{parincr}(n-1)$ and can then be split along its parallel composition. Then the forked thread receives incr and the recursive call to `spawn` receives $\mathsf{parincr}(n-1)$. After calling **join** on line 8, both the call to `worker` and the recursive call to `spawn` have ensured completing the process they received, thereby leaving the (merged) process $\varepsilon \parallel \varepsilon$, which can be rewritten to ε to satisfy the postcondition of `spawn`. As a result, after calling **finish** on line 18 we can successfully verify that *counter* has indeed been increased by n.

Unequal concurrent counting. One could consider an interesting variant on the two-threaded concurrent counting problem: one thread performing the assignment "$\mathit{counter} = \mathit{counter} + v$" for some integer value v, and the other thread concurrently performing "$\mathit{counter} = \mathit{counter} * v$". Starting from a state where $\mathit{counter} = c$ holds for some c, the challenge is to verify that after running the program we either have $\mathit{counter} = (c + v) * v$ or $\mathit{counter} = (c * v) + v$.

This program can be verified using our model-based approach (without requiring for example auxiliary state) by defining corresponding actions for the two different assignments. The global model is described as the process $\mathsf{count}(\textbf{int}\ n) := \mathsf{plus}(n) \parallel \mathsf{mult}(n)$, where the action $\mathsf{plus}(n)$ has the effect of incrementing *counter* by n and $\mathsf{mult}(n)$ has the effect of multiplying *counter* by n. The required program annotations are then similar to the ones used in Figure 7.

All three variants on the concurrent counting problem can be automatically verified using the VerCors toolset.

5.2 Lock specification

The third example demonstrates how our approach can be used to verify control-flow properties of programs, in this case the *compare-and-swap lock implementation* that is presented in the Concurrent Abstract Predicates (CAP) paper [9]. The implementation is given in Figure 9. The $\texttt{cas}(x, c, v)$ operation is the *compare-and-swap* instruction, which atomically updates the value of x by v if the old value at x is equal to c, otherwise the value at x is not changed. A Boolean result is returned indicating whether the update to x was successful.

In particular, model-based reasoning is used to verify that the clients of this lock adhere to the intended locking protocol: clients may only successfully acquire the lock when the lock was unlocked and vice versa. Stated differently, we verify that clients may not acquire (nor release) the same lock successively.

The process algebraic description of the locking protocol is a composition of two actions, named acq and rel, that model the process of acquiring and releasing

```
1  bool flag := false;
2
3  void acquire() {
4  |  bool b := false;
5  |  while ¬b {
6  |  |  b := cas(flag, false, true);
7  |  }
8  }
9
10 void release() {
11 |  atomic { flag := false; }
12 }
```

Fig. 9: Implementation of a simple locking system.

the lock, respectively. A third action named done is used to indicate that the lock is no longer used and can thus be destroyed. We use this process as a model to protect changes to the shared variable $flag$, so that all changes to $flag$ must either happen as an acq or as a rel action. The acq action may be performed only if $flag$ is currently false and has the effect of setting $flag$ to true. The rel action simply has the effect of setting $flag$ to false, whatever the current value of $flag$ (therefore rel does not need a guard). The locking protocol is defined by the processes $\mathsf{Locked}() := \mathsf{rel} \cdot \mathsf{Unlocked}()$ and $\mathsf{Unlocked}() := \mathsf{acq} \cdot \mathsf{Locked}() + \mathsf{done}$. This allows us to use the following interface specifications for the `acquire` and `release` procedures (with m a global identifier of an initialised model):

$$\{\mathsf{Proc}_\pi(m, \mathsf{Unlocked}())\}\ \texttt{acquire}()\ \{\mathsf{Proc}_\pi(m, \mathsf{Locked}())\}$$
$$\{\mathsf{Proc}_\pi(m, \mathsf{Locked}())\}\ \texttt{release}()\ \{\mathsf{Proc}_\pi(m, \mathsf{Unlocked}())\}$$

Specification-wise, clients of the lock may only perform `acquire` when they have a corresponding process predicate that is in an "Unlocked" state (and the same holds for `release` and "Locked"), thereby enforcing the locking protocol (i.e. the process only allows traces of the form: $\mathsf{acq}, \mathsf{rel}, \mathsf{acq}, \mathsf{rel}, \cdots$). The `acquire` procedure performs the acq action via the `cas` operation: one may define `cas` to update $flag$ as an acq action. Moreover, since `cas` is an atomic operation, it can get all necessary ownership predicates from the resource invariant inv. Furthermore, calling `destroy`() corresponds to performing the done action on the process algebra level, which may only be done in the "Unlocked" state.

The full annotated lock implementation is presented in Figure 10. The `init` and `destroy` procedures have been added to initialise and finalise the lock and thereby to create and destroy the corresponding model. The `init` consumes write permission to $flag$, creates the model, and transfers the converted write permission into the resource invariant inv. Both the atomic block (on line 28) and the `cas` operation (on line 21) make use of inv to get permission to change the value of $flag$ in an action block. The `cas` operation on line 21 performs the acq action internally, depending on the success of the compare-and-swap (indicated by its return value). This is reflected upon in the loop invariant. The `destroy` procedure has the opposite effect of `init`: it consumes the (full) Proc predicate (in state "Unlocked"), destroys the model and the associated resource invariant, and gives back the converted write permission to $flag$.

In the current presentation, `init` returns a single Proc predicate in state Unlocked, thereby allowing only a single client. This is however not a limita-

```
1  bool flag;
2  model m;
3
4  resource inv := flag ↪¹ₚ −;
5
6  guard ¬flag; effect flag; action acq;
7  effect ¬flag; action rel;
8
9  process Unlocked() := acq · Locked();
10 process Locked() :=
11     rel · Unlocked() + done;
12
13 requires Proc_π(m, Unlocked());
14 ensures Proc_π(m, Locked());
15 void acquire() {
16 |   bool b := false;
17 |   loop-inv ¬b ⇒
18 |        Proc_π(m, acq · Locked());
19 |   loop-inv b ⇒ Proc_π(m, Locked());
20 |   while ¬b {
21 |   |   b := cas(flag, false, true);
22 |   }
23 }
24
25 requires Proc_π(m, Locked());
26 ensures Proc_π(m, Unlocked());
27 void release() {
28 |   atomic inv {
29 |   |   action m.rel { flag := false; }
30 |   }
31 }
32
33 requires flag ↪¹ₙ −;
34 ensures Proc₁(m, Unlocked());
35 void init() {
36 |   flag := false;
37 |   m := model Unlocked();
38 |   init inv;
39 }
40
41 requires Proc₁(m, Unlocked());
42 ensures flag ↪¹ₙ −;
43 void destroy() {
44 |   action m.done { }
45 |   destroy inv;
46 |   finish m;
47 }
```

Fig. 10: The annotated implementation of the simple fine-grained locking system.

tion: to support two clients, `init` could alternatively initialise and ensure the Unlocked() ∥ Unlocked() process. Furthermore, to support n clients (or a dynamic number of clients), `init` could apply a construction similar to the one used in the generalised concurrent counting example (see Section 5.1).

Reentrant locking. The process algebraic description of the locking protocol can be upgraded to describe a *reentrant lock*: a locking system where clients may `acquire` and `release` multiple times in succession. A reentrant lock that is acquired n times by a client must also be released n times before it is available to other clients. Instead of using the Locked and Unlocked processes, the reentrant locking protocol is described by the following process (with $n \geq 0$):

$$\textbf{process}\ \mathsf{Lock}(\textbf{int}\ n) := \mathsf{acq} \cdot \mathsf{Lock}(n+1) + (\textbf{if}\ n > 0\ \textbf{then}\ \mathsf{rel} \cdot \mathsf{Lock}(n-1))$$

Rather than describing the lock state as a Boolean flag, like done in the single-entrant locking example, the state of the reentrant lock can be described as a *multiset* containing thread identifiers. In that case, acq and rel protect all changes made to the multiset in order to enforce the locking protocol described

by Lock. The interface specifications of `acquire` and `release` then become:

$$\{\mathsf{Proc}_\pi(m, \mathsf{Lock}(n))\}\ \texttt{acquire}()\ \{\mathsf{Proc}_\pi(m, \mathsf{Lock}(n+1))\}$$
$$\{\mathsf{Proc}_\pi(m, \mathsf{Lock}(n)) \wedge n > 0\}\ \texttt{release}()\ \{\mathsf{Proc}_\pi(m, \mathsf{Lock}(n-1))\}$$

Moreover, the $\mathsf{Lock}(n)$ process could be extended with a done action to allow the reentrant lock to be destroyed. The done action should then only be allowed when $n = 0$. Both the simple locking implementation and the reentrant locking implementation have been automatically verified using the VerCors toolset.

5.3 Other verification examples

This section demonstrated the use of process algebraic models in three different verification examples, as well as some interesting variants on them. We showed how model-based reasoning can be used as a practical tool to verify different types of properties that would otherwise be hard to verify, *especially with an automated tool*. We considered *data properties* in the parallel GCD and the concurrent counting examples, and considered *control-flow* properties in the locking examples. Moreover, we showed how to use the model-based reasoning approach in environments with a dynamic number of concurrent threads.

Our approach can also be used to reason about *non-terminating* programs. Notably, a *no-send-after-read* verification example is available that addresses a commonly used security property: if confidential data is received by a secure device, it will not be passed on. The concrete send- and receive behaviour of the device can be abstracted by send and recv actions, respectively. Receiving confidential information is modelled as the clear action. Essentially, we show that after performing a clear action the device can no longer perform send's.

6 Related Work

The abstraction technique proposed in this paper allows reasoning about functional behaviour of concurrent, possibly non-terminating programs. A related approach is (impredicative) Concurrent Abstract Predicates (CAP)[9,25], which also builds on CSL with permissions. In the program logic of CAP, *regions* of memory can be specified as being *shared*. Threads must have a consistent view of all shared regions: all changes must be specified as *actions* and all shared regions are equipped with a set of possible actions over their memory. Our approach uses process algebraic abstractions over shared memory in contrast to the shared regions of CAP, so that all changes to the shared memory must be captured as process algebraic actions. We mainly distinguish in the use of *process algebraic reasoning* to verify properties that could otherwise be hard to verify, and in the capability of doing this mechanically by providing tool support.

Other related approaches include TaDA [7], a program logic that builds on CAP by adding a notion of *abstract atomicity* via Hoare triples for atomic operations. CaReSL [26] uses a notion of shared regions similar to CAP, but uses

tokens to denote ownership. These tokens are used to transfer ownership over resources between threads. Iris [17,18] is a reasoning framework that aims to provide a comprehensive and simplified solution for recent (higher-order) concurrency logics. Sergey et al. [24] propose *time-stamped histories* to capture modifications to the shared state. Our approach may both capture and model program behaviour and benefits from extensive research on process algebraic reasoning [12]. Moreover, the authors provide a *mechanised* approach to interactively verify full functional correctness of concurrent programs by building on CSL [23]. Popeea and Rybalchenko [21] combine abstraction refinement with rely-guarantee reasoning to verify termination of multi-threaded programs.

In the context of verifying distributed systems, Session Types [15] describe communication protocols between processes [14]. However, our approach is more general as it allows describing any kind of behaviour, including communication behaviour between different system components.

7 Conclusion

This paper addresses thread-modular verification of possibly non-terminating concurrent programs by proposing a technique to abstract program behaviour using process algebras. A key characteristic of our approach is that properties about programs can be proven by analysing process algebraic program abstractions and by verifying that programs do not deviate from these abstractions. The verification is done in a thread-modular way, using an abstraction-aware extension of CSL. This paper demonstrates how the proposed technique provides an elegant solution to various verification problems that may be challenging for alternative verification approaches. In addition, the paper contributes tool support and thereby allow mechanised verification of the presented examples.

Future work. We are currently working on mechanising the formalisation and the soundness proof of the proposed technique using Coq. At the moment, verification at the process algebra level is non-modular. As a next step, we plan to achieve modularity at this level as well, by combining our approach with rely-guarantee [16] and deny-guarantee reasoning [10]. We also plan to investigate how to mix and interleave abstract and concrete reasoning. In the current set up, reasoning is done completely at the level of the abstraction. If this part of the program is used as a component in a larger program, we plan to investigate how the verification results for the components can be used to reason about the larger program, if reasoning about the larger program is not done at this level of abstraction. Finally, in a different direction, we plan to extend the abstraction technique to reason about distributed software. For example, abstractions may be used to capture the behaviour of a single actor/agent as a process term, allowing process algebraic techniques such as [20] to be used for further verification.

Acknowledgements. This work is partially supported by the ERC grant 258405 for the VerCors project and by the NWO TOP 612.001.403 project VerDi.

References

1. A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multithreaded Java programs. *LMCS*, 11(1), 2015.
2. S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *iFM*, LNCS. Springer, 2017.
3. S. Blom and M. Huisman. The VerCors tool for verification of concurrent programs. In *FM*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
4. S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. History-based verification of functional behaviour of concurrent programs. In *SEFM*, volume 9276 of *LNCS*. Springer, 2015.
5. R. Bornat, C. Calcagno, P.W. O'Hearn, and M.J. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270, 2005.
6. J. Boyland. Checking interference with fractional permissions. In *SAS*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
7. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A Logic for Time and Data Abstraction. In *ECOOP*, volume 8586 of *LNCS*, pages 207–231. Springer, 2014.
8. P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Steps in Modular Specifications for Concurrent Modules. In *MFPS*, EPTCS, pages 3–18, 2015. doi:10.1016/j.entcs.2015.12.002.
9. T. Dinsdale-Young, M. Dodds, P. Gardner, M.J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.
10. M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-Guarantee Reasoning. In *ESOP*, volume 5502 of *LNCS*, pages 363–377. Springer, 2009.
11. A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *APLAS*, volume 4807 of *LNCS*, pages 19–37. Springer, 2007.
12. J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.
13. J.F. Groote and A. Ponse. The syntax and semantics of mCRL. In *Algebra of Communicating Processes*, pages 26–62. Springer, 1995.
14. K. Honda, E. Marques, F. Martins, N. Ng, V. Vasconcelos, and N. Yoshida. Verification of MPI Programs using Session Types. In *EuroMPI*, volume 7940 of *LNCS*, pages 291–293. Springer, 2012. doi:10.1007/978-3-642-33518-1_37.
15. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284. ACM, 2008.
16. C.B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.
17. R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*, pages 637–650. ACM, 2015.
18. R. Krebbers, R. Jung, A. Bizjak, J. Jourdan, D. Dreyer, and L. Birkedal. The Essence of Higher-Order Concurrent Separation Logic. In *ESOP*, volume 10201 of *LNCS*, pages 696–723. Springer, 2017.
19. P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.
20. J. Pang, J. van de Pol, and M. Espada. Abstraction of Parallel Uniform Processes with Data. In *SEFM*, pages 14–23. IEEE, 2004.

21. C. Popeea and A. Rybalchenko. Compositional Termination Proofs for Multi-threaded Programs. In *TACAS*, volume 7214 of *LNCS*, pages 237–251. Springer, 2012.
22. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.
23. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM, 2015.
24. I. Sergey, A. Nanevski, and A. Banerjee. Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity. In *ESOP*, volume 9032 of *LNCS*, pages 333–358. Springer, 2015.
25. K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *ESOP*, volume 8410 of *LNCS*, pages 149–168. Springer, 2014.
26. A. Turon, D. Dreyer, and L. Birkedal. Unifying Refinement and Hoare-style Reasoning in a Logic for Higher-Order Concurrency. In *ICFP*, pages 377–390, 2013.
27. M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying Functional Behaviour of Concurrent Programs*. PhD thesis, University of Twente, 2015. doi:10.3990/1.9789036539241.