VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs

Petra van den Bos[1](✉) and Sung-Shik Jongmans[2,3]

[1] Formal Methods and Tools Group, University of Twente, Enschede, The Netherlands
p.vandenbos@utwente.nl

[2] Department of Computer Science, Open University, Heerlen, The Netherlands

[3] CWI, Amsterdam, The Netherlands

Abstract. We present VeyMont: a deductive verification tool that aims to make reasoning about functional correctness and deadlock freedom of parallel programs (relatively complex) as easy as that of sequential programs (relatively simple). The novelty of VeyMont is that it "inverts the workflow": it supports a new method to parallelise verified programs, in contrast to existing methods to verify parallel programs. Inspired by methods for distributed systems, VeyMont targets coarse-grained parallelism among threads (i.e., whole-program parallelisation) instead of fine-grained parallelism among tasks (e.g., loop parallelisation).

1 Introduction

Deductive verification is a classical approach to reason about functional correctness of programs. The idea is to annotate programs with logic assertions about state. A proof system can subsequently be used to statically check whether or not annotations are true (i.e., whether or not state dynamically evolves as asserted).

As multicore hardware and multithreaded software have become ubiquitous, deductive verification has been facing an elusive open problem: the approach is much harder to apply to *parallel programs* than to *sequential programs*. Towards addressing this issue, in this paper, we present **VeyMont**. It is a deductive verification tool that aims to make reasoning about functional correctness and deadlock freedom of parallel programs *as easy* as that of sequential programs. The novelty of VeyMont is that it "inverts the workflow": it supports a new method to **parallelise verified programs**, in contrast to existing methods to **verify parallel programs**. Unlike traditional model checkers, VeyMont proves properties *generally* for all (possibly infinitely many) initial values of variables, instead of *specifically* for instances. Unlike parallelising compilers, VeyMont targets *coarse-grained* parallelism among threads (i.e., whole-program parallelisation), instead of *fine-grained* parallelism among instructions (e.g., loop parallelisation).

Background. In the state-of-the-art on verification of sequential and parallel programs, typically, proof systems based on (extensions of) *Hoare logic* [4,21] and *separation logic* [40,45] are used to prove properties of annotated programs. To demonstrate the main concepts, Fig. 1 shows four functionally equivalent programs to swap the values of variables `x` and `y`:

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
M. Chechik et al. (Eds.): FM 2023, LNCS 14000, pp. 321–339, 2023.
https://doi.org/10.1007/978-3-031-27481-7_19

$\{x = v_1 \wedge y = v_2\}$
z := x ; x := y ; y := z
$\{x = v_2 \wedge y = v_1\}$

(a) Sequential program

```
1 class SeqProgram {
2   int v1, v2, x, y, z;
3
4   context ...
5   requires x == v1 && y == v2;
6   ensures  x == v2 && y == v1;
7   void swap() {
8     z = x; x = y; y = z;
9 } }
```

(b) Sequential program in VerCors

$\{x \mapsto v_1 * y \mapsto v_2 *$
$z1 \mapsto _ * z2 \mapsto _\}$
barrier b
$\{z1 \mapsto v_1 \triangleright z2 \mapsto v_2,$
$z2 \mapsto v_2 \triangleright z1 \mapsto v_1\}$ **in**

Left thread	Right thread
$\{x \mapsto v_1 * z1 \mapsto _\}$	$\{y \mapsto v_2 * z2 \mapsto _\}$
z1 := x ;	z2 := y ;
wait b ;	**wait** b ;
x := z2	y := z1
$\{x \mapsto v_2\}$	$\{y \mapsto v_1\}$

$\{x \mapsto v_2 * y \mapsto v_1\}$

$x \mapsto v$ means shared variable x is *owned* (i.e., exclusive permission to use) and has value v; "_" means "any".

$\phi_1 * \phi_2$ means memory can be *separated* into two parts s.t. ϕ_1 and ϕ_2 are true in different parts.

$\phi_1 \triangleright \chi_1, \ldots, \phi_n \triangleright \chi_n$ means: ϕ_i must be true before thread i passes the barrier; χ_i will be true after.

(c) Parallel program, using a barrier

```
1  class Channel {
2    int s, buf; // state, buffer
3
4    resource lock_invariant() =
5      Perm(s, 1) **
6      (s == 1 || s == 2 || s == 3) **
7      (s == 1 ==> Perm(buf, 1\2)) **
8      (s == 2 ==> Perm(buf, 1\2));
9
10   context Perm(buf, 1\2);
11   ensures buf == v;
12   void writeValue(int v) {
13     lock this;
14     loop_invariant ...;
15     while (s != 1) { wait this; }
16     s = 2; buf = v;
17     unlock this;
18   }
19
20   ensures Perm(buf, 1\2);
21   ensures \result == buf;
22   int readValue() { ... }
23 }
24
25 class Thread {
26   Channel a, b; int v, v_old;
27   context  Perm(v, 1);
28   context  Perm(v_old, 1\2);
29   context  Perm(a, 1\2);
30   context  Perm(b, 1\2);
31   context  Perm(b.buf, 1\2);
32   requires v == v_old;
33   requires a != null;
34   requires b != null;
35   ensures  Perm(a.buf, 1\2);
36   ensures  v_old == b.buf
37   ensures  a.buf == v;
38   void run() {
39     b.writeValue(v);
40     v = a.readValue();
41 } }
42
43 class ParProgram {
44   int v1, v2; Thread t1, t2;
45
46   context ...
47   requires t1.v == v1 && t2.v == v2;
48   ensures  t1.v == v2 && t2.v == v1;
49   void swap() {
50     fork t1; fork t2;
51     join t1; join t2;
52 } }
```

(d) Parallel program in VerCors, using channels

Fig. 1. Example of deductive verification (swapping values)

– Figure 1a shows a **sequential program**; it uses auxiliary variable `z`. The program is annotated with two assertions (in teal), expressed in Hoare logic: the *precondition* (top) specifies what must be true before the program is run; the *postcondition* (bottom) specifies what will be true after.
– Figure 1c shows a **parallel program**, with two threads; it uses a barrier `b` and auxiliary variables `z1` and `z2`. First, the "left thread" copies `x` into `z1`; next, it waits on `b` (until the "right thread" has copied `y` into `z2`); next, it copies `z2` into `x`. In parallel, the "right thread" behaves symmetrically. The barrier is crucial: without it, the threads can prematurely copy `z1` and `z2`.

The program is annotated with seven assertions (in teal), expressed in a variant of separation logic [22,23]: the "global" and "local" pre/postconditions specify the behaviour of the whole program and of the separate threads; the *barrier contract* specifies for every thread what must be true before it passes the barrier, and what will be true after (i.e., transfer of ownership and data).

– To offer also a more practical perspective, Fig. 1b and Fig. 1d show excerpts of **the same programs, but represented in the input format of *VerCors*** [9,11], a state-of-the-art deductive verifier. Keywords `requires`, `ensures`, and `context` indicate preconditions, postconditions, and method invariants, respectively. For instance, the pre/postconditions in Fig. 1a and Fig. 1c correspond to lines 5–6 in Fig. 1b and lines 47–48 in Fig. 1d. Furthermore, an assertion of the form `Perm(`x`,`q`)` in Fig. 1d indicates the permission to write to variable x ($q = 1$) or to read from it ($q < 1$). That is, $x \mapsto v$ in Fig. 1c is written as the conjunction of `Perm(`x`,1)` and x `==` v in Fig. 1d.
  We organised the code in Fig. 1d differently from the code in Fig. 1c, as VerCors does not support such barriers. Instead we implemented a custom *channel* to transfer data/ownership between threads (lines 39–40), using VerCors's *locking* mechanism. The *lock invariant* (lines 4–8) specifies what is assumed upon acquiring, and asserted upon releasing, an object's lock.

Open Problem. Based on Fig. 1, we make two observations:

– Fig. 1a and Fig. 1b show that deductive verification of <u>simple</u> sequential programs is <u>simple</u> (i.e., relatively little effort to annotate).
– However, Fig. 1c and Fig. 1d show that deductive verification of corresponding parallel programs is surprisingly <u>hard</u> (i.e., relatively big effort).
  Moreover, while VerCors automatically checks the truth of the annotations (advantage relative to pen-and-paper proofs), manually writing these annotations can be burdensome, as seen by comparing Fig. 1c and Fig. 1d. Specifically, the "local" pre/postconditions of the "left thread" in Fig. 1c are more concise than those for method `run` in class `Thread` in Fig. 1d.

Thus, in existing approaches, verification of parallel programs is substantially more laborious than that of sequential programs; *already in theory, using pen and paper, but*—paradoxically—*sometimes more so in practice, using tool support.* We illustrate these findings with the simplest non-trivial example we could think of. This problem is only aggravated as the complexity of the programs increases.

Essentially, the reason why annotations of parallel programs are complicated, is because synchronisation (of data accesses/mutations) among threads needs to be specified explicitly with permissions. This is already non-trivial when using the high-level barrier in Fig. 1c (writing the barrier contract); getting synchronisation among threads right costs even more intellectual effort when we are forced to implement the custom channels in Fig. 1d using lower-level locks (VerCors does not have such built-in barriers). In the sequential programs in Fig. 1a and Fig. 1b, we need not worry about synchronisation among threads at all; this is the level of simplicity that VeyMont aims to provide (e.g., we added support in VeyMont to auto-generate permissions, so the user needs not write them).

Contributions. Existing methods (e.g., [12,15,22,23,29,39,41,42]) and tools (e.g. [11,27,52]) for deductive verification of parallel programs have this workflow:

Step 1: Parallelise a sequential program. **Step 2:** Verify it.
(Or write a parallel program from scratch.)

However, step 2 requires significant extra, and non-trivial, annotation effort. As demonstrated above, this makes deductive verification of parallel programs much harder than that of sequential programs. To address this issue, we are developing a new method and tool that have an "inverted workflow":

Step 1: Verify a *sequential-ish* program. **Step 2:** Parallelise it.

The idea behind sequential-ish programs is that they have sequential syntax and sequential axiomatic semantics (i.e., proof system), but parallel operational semantics. That is, they look and feel as sequential programs, but they are run as parallel programs. More concretely, the user uses Hoare logic to annotate a sequential-ish program P_{seq}—without worrying about synchronisation—after which a *functionally correct, deadlock-free* parallel program P_{par} is generated:

- "Functionally correct" means that if the precondition of P_{seq} holds in the initial state of P_{par}, then the postcondition of P_{seq} holds in the final state of P_{par} (i.e., functional correctness of P_{seq} is preserved in P_{par}).
- "Deadlock free" means that threads do not get stuck waiting on each other, e.g. because two threads are both reading from a channel but expect the other thread to write. No additional manual annotations are needed.

In a previous paper [30], we presented the theoretical foundations of this new method and its "inverted workflow", targetting coarse-grained parallelism among threads (inspired by distributed systems). In this paper, we present the first deductive verification tool that supports it. The novel contributions are:

1. We designed and implemented VeyMont: it accepts an annotated sequential-ish program as input and offers a functionally correct, deadlock-free parallel program in Java as output. Section 2 and Sect. 3 provide an overview of the workflow and features of VeyMont, by example; Sect. 4 contains details.
2. We evaluated VeyMont along two dimensions. As case studies in applicability, we used VeyMont to verify and parallelise sequential-ish versions of distributed algorithms. As case studies in efficiency, we used VeyMont to produce parallel programs in Java that have comparable performance to third-party reference implementations. Section 5 describes our findings.

The artifact for reproducing the experiments of this paper is available at [51].

Related Work. Existing tools for deductive verification of parallel programs include Frama-C [5], KeY-ABS [20], VeriFast [27], and Gobra [52]. However, these tools verify parallel programs, whereas VeyMont parallelises verified programs.

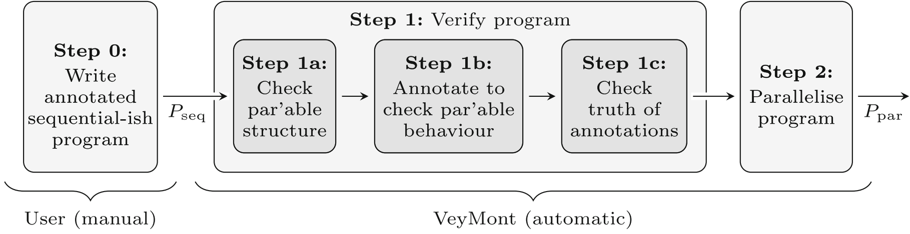

Fig. 2. "Inverted workflow" using VeyMont

The "inverted workflow"—*verify first, parallelise second*—of the method supported by VeyMont is strongly inspired by the methods of *choreographic programs* [17,18] and *multiparty session types* [24] for construction/analysis of deadlock-free distributed systems. The idea behind those methods is: first, to implement/ specify distributed systems as *choreographies/global types* (cf. sequential-ish programs); second, to generate sets of *processes/local types* (cf. parallel programs with threads) that are formally guaranteed to be deadlock-free. Existing tools that support these methods include Chor [17], Scribble [25] and its dialects [19,35,36,46], Pabble [37], and ParTypes [34]. However, these tools offer deadlock freedom, but *not* functional correctness; VeyMont offers both.

The literature on *parallelising compilers* that target fine-grained parallelism among tasks is rich (e.g., loop parallelisation [2,13,16,33,38,49]) and goes back to the 1970 s [31]. In contrast, VeyMont is a *parallelising verifier* that targets coarse-grained parallelism among threads (i.e., whole-program parallelisation). We discuss the integration of fine-grained parallelism into VeyMont in Sect. 6.

2 Overview of VeyMont – The "Inverted Workflow"

Figure 2 visualises the "inverted workflow" of the method supported by VeyMont.

Step 0: The user writes a sequential-ish program P_{seq} in VeyMont's input language μPVL (core fragment of VerCors's language PVL [50]). This is a programming/assertion language that combines object-oriented sequential programs with Hoare logic assertions (similar to sequential Java, enriched with JML [32]).

For instance, Fig. 3a shows a sequential-ish program in µPVL (cf. Figure 1d). It is split into two parts: fields `s1` and `s2` of class `SeqProgram` define the *data* (lines 1–12), while method `run` defines the *sequence of operations* (lines 16–21). The precondition of `run` is trivial (line 13); the postcondition uses the `\old` predicate for the old values of `s1.v` and `s2.v` at the start of `run` (lines 14–15). As `s1.v` and `s2.v` are initialised to `x` and `y` (lines 12–13), which are free program arguments (line 9), all possible initial values of `s1.v` and `s2.v` are quantified over.

Step 1a: VeyMont checks whether or not P_{seq} has a *parallelisable* ("par'able") *structure*. This is a set of syntactic conditions, beyond µPVL's grammar, that P_{seq} must meet to be able to generate a grammatical parallel program (step 2).

```
 1 class Storage {     // The name of this class is
 2   int v, temp;      // specifically chosen to
 3   Storage(int v) { // clarify VeyMont. Generally,
 4     this.v = v;     // it can be anything.
 5 } }
 6
 7 class SeqProgram { // The name of this class is
 8   Storage s1, s2;  // always mandatory (Sect. 4).
 9   SeqProgram(int x, int y) {
10     s1 = new Storage(x);
11     s2 = new Storage(y);
12   }
13   requires true;
14   ensures s1.v == \old(s2.v);
15   ensures s2.v == \old(s1.v);
16   void run() {
17     s1.temp = s1.v;
18     s2.temp = s2.v;
19     s1.v = s2.temp;
20     s2.v = s1.temp;
21 } }
```

(a) Sequential-ish program in μPVL – *input*

```
 1 class s1Thread extends Thread {
 2   Storage s1;
 3   IntegerChannel s1_s2;
 4   IntegerChannel s2_s1;
 5
 6   s1Thread(int x,
 7     IntegerChannel s1_s2,
 8     IntegerChannel s2_s1) {
 9
10     s1 = new Storage(x);
11     s1_s2 = s1_s2;
12     s2_s1 = s2_s1;
13   }
14
15   public void run() {
16     s1.temp = s1.v;
17     s1.v = s2_s1.read();
18     s1_s2.write(s1.temp);
19 } }
20 class s2Thread extends Thread {
21
22   ... // similar to lines 2-13
23
24   public void run() {
25     s2.temp = s2.v;
26     s2_s1.write(s2.temp);
27     s2.v = s1_s2.read();
28 } }
29
30 class ParProgram {
31   ParProgram(int x, int y) {
32     IntegerChannel s1_s2 = ...;
33     IntegerChannel s2_s1 = ...;
34     new s1Thread(
35       x, s1_s2, s2_s1).start();
36     new s2Thread(
37       y, s1_s2, s2_s1).start();
38 } }
```

(b) Parallel program in Java, excerpt – *output*

Fig. 3. Example of VeyMont (swapping values)

Step 1b: VeyMont generates annotations for P_{seq}—in addition to those the user has written in step 0—to be able to check that it has *parallelisable behaviour* (step 1c). This is a set of semantic conditions, encoded as logic assertions, that P_{seq} must meet to guarantee that functional correctness of P_{seq} will be preserved.

Step 1c: VeyMont checks the truth of the annotations in P_{seq}, using the state-of-the-art *VerCors–Viper–Z3* tool stack [9,11]. If so, P_{seq} is guaranteed to be functionally correct (the user's annotations; step 0), functional correctness is guaranteed to be preserved through parallelisation (VeyMont's annotations; step 1b), and parallelisation does not introduce deadlocks.

Step 2: VeyMont generates a parallel program P_{par} in Java. Step 1a guarantees that P_{seq} is parallelisable; steps 1b–1c and the theoretical foundations of VeyMont guarantee that P_{par} is functionally correct and deadlock-free [30].

For instance, Fig. 3b shows an excerpt of the parallel program generated for the sequential-ish program in Fig. 3a. The idea is to parallelise coarse-grained, at the level of granularity *of top-level fields*. For every field $f \in \{\texttt{s1}, \texttt{s2}\}$ of class `SeqProgram` in Fig. 3a, there is a corresponding subclass f`Thread` of class `Thread`

```
1  class Player {
2    int m, n;
3    int[][] grid;
4    Move move;
5
6    ...
7
8    ensures (\forall int i = 0..m;
9               (\forall int j = 0..n;
10                grid[i][j] == 0));
11   ensures ...
12   Player(int m, int n, ...) {
13     this.grid = new int[m][n];
14     ...
15   }
16
17  requires ...
18   ensures 0 <= move.x && move.x < m;
19   ensures 0 <= move.y && move.y < n;
20   ensures grid[move.x][move.y] == 0;
21   ensures ...
22   void think();
23
24  requires ...
25  requires 0 <= move.x && move.x < m;
26  requires 0 <= move.y && move.y < n;
27  requires grid[move.x][move.y] == 0;
28   ensures ...
29   void play();
30 }

31 class Move {
32   int x, y, t;
33   ...
34 }

35 class SeqProgram {
36   Player p1, p2;
37
38   ...
39
40   ensures eq_grids(p1, p2);
41   SeqProgram(int m, int n) {
42     p1 = new Player(m, n, ...);
43     p2 = new Player(m, n, ...);
44   }
45
46   context eq_grids(p1, p2);
47   void turn1() {
48     p1.think();
49     p1.play();
50     p2.think(); // in the background
51     p2.move = p1.move.clone();
52     p2.play(); // to update
53   }
54
55   ...
56
57   context eq_grids(p1, p2);
58   void run() {
59     loop_invariant eq_grids(p1, p2);
60     while (p1.inPlay && p2.inPlay) {
61       turn1();
62       if (p1.inPlay && p2.inPlay) {
63         turn2();
64 } } } }
```

Fig. 4. Another example of VeyMont (tic–tac–toe on an arbitrary $m \times n$ grid)

in Fig. 3b (which defines a Java thread); this subclass alone is responsible for managing the data of f and performing its operations in class `ParProgram`.

f`Thread` has three fields: the `Storage` that it is responsible for, and `Channels` to explicitly transfer data between `Storages`. Meanwhile, method `run` of f`Thread` defines the operations that it needs to perform, derived from method `run` of class `SeqProgram`: if only f occurs in an assignment in `run` of `SeqProgram`, then the assignment is copied into `run` of f`Thread`, verbatim (e.g., line 17 in Fig. 3a, line 16 in Fig. 3b); alternatively, if also $g \in \{\mathtt{s1}, \mathtt{s2}\} \setminus \{f\}$ occurs in the assignment, then an explicit data transfer between `Storages` is introduced (i.e., fThread is forbidden to use data of gThread directly). Transfers are *synchronous*: method `read` blocks until method `write` is called, and vice versa. In this way, `Channels` are an alternative synchronisation mechanism to the barrier in Fig. 1c.

Generally, explicit data transfers are the only form of synchronisation that VeyMont needs to introduce to guarantee functional correctness (given step 1c). Specifically, the values of `s1.v` and `s2.v` are swapped in `run` of `ParProgram`, just as asserted by the postcondition of `run` of `SeqProgram`. Finally, we note that `ParProgram` really is parallel: lines 16 and 25 can be executed simultaneously.

3 Overview of VeyMont – More Features

To demonstrate some more features of µPVL/VeyMont, Fig. 4 shows an excerpt of another sequential-ish program in µPVL. Two threads—implicitly declared in

top-level fields `p1` and `p2` of class `SeqProgram`—take turns to simulate a game of tic–tac–toe on an arbitrary $m \times n$ grid (i.e., beyond 3×3, all possible grid sizes are quantified over). Each thread has its own copy of the grid; when a move is made, the active thread informs the passive thread accordingly, so the passive thread can update its grid to match. In the active thread's turn, the passive thread can "think ahead" to ponder its next move. This makes the program really parallel.

We highlight the noteworthy features, as supported by μPVL/VeyMont:

- **Turing completeness:** Method `run` of class `SeqProgram` shows that μPVL has *if/while-statements*. This is actually significant: automatically parallelising *the conditions of* if/while-statements, while guaranteeing functional correctness and deadlock freedom, has been a key challenge in developing VeyMont's theoretical foundations [30]. It is also a reason why VeyMont needs to check if a sequential-ish program has parallelisable behaviour in steps 1b–1c.
- **Data structures:** The fields of class `Player` show that μPVL/VeyMont has *multidimensional arrays* (field `grid`) and *nesting* of classes (field `move`).
- **Trusted code:** Methods `think` and `play` of class `Player` show that μPVL/VeyMont has *abstract methods*: they have a specification (precondition and postcondition), but no implementation (method body). This allows the user to integrate external trusted code into parallel programs generated by VeyMont. If the trusted code truly implements the specification (proved using VeyMont, or proved using a different tool, or estimated with code reviews, etc.), then functional correctness and deadlock freedom are guaranteed.

An excerpt of the parallelisation generated by VeyMont appears in Sect. A.

4 Design and Implementation

VeyMont has five main components, each of which enables a (sub)step in Fig. 2.

4.1 Parser (Step 1a)

The first main component of step 1a is a *parser* for μPVL. It accepts sequential-ish programs that comply with the grammar in Fig. 5. We split the grammar into an "external fragment" and an "internal fragment". The difference is that the internal fragment supports more complicated assertions, which the user should never write manually; instead, they are always inserted by VeyMont automatically (step 1b; Sect. 4.4). Regarding the external fragment:

- **Basic notation:** Let n range over class names, f over field names, m over method names, and x over variable names. We write $\tilde{\square}$ to mean a list of $\square$s.
- **Programs, classes, fields, methods, annotations:** A *program* P consists of a list of classes. A *class* C consists of a name, a list of fields, and a list of methods, including a constructor that has the same name as the class. A *method* M consists of a list of annotations (contract), a list of variable names

$$
\begin{array}{rcl}
P & ::= & \tilde{C} \\
C & ::= & \texttt{class}\ n\ \texttt{\{}\ \tilde{F}\ \tilde{M}\ \texttt{\}} \\
F & ::= & f \\
M & ::= & \tilde{A}\ m\texttt{(}\tilde{x}\texttt{)}\ \texttt{\{}\ \tilde{S}\ \texttt{\}}\ \mid \\
 & & \tilde{A}\ m\texttt{(}\tilde{x}\texttt{)} \\
A & ::= & \texttt{requires}\ B\texttt{;}\ \mid \\
 & & \texttt{ensures}\ B\texttt{;}\ \mid \\
 & & \texttt{context}\ B\texttt{;}
\end{array}
\qquad
\begin{array}{rcl}
S & ::= & \texttt{assert}\ B\texttt{;}\ \mid\ X\ \texttt{=}\ E\texttt{;}\ \mid\ E\texttt{.}m\texttt{(}\tilde{E}\texttt{);}\ \mid \\
 & & \texttt{if (}B\texttt{) \{}\ \tilde{S}_1\ \texttt{\} else \{}\ \tilde{S}_2\ \texttt{\}}\ \mid \\
 & & \texttt{loop_invariant}\ B_1\texttt{; while (}B_2\texttt{) \{}\ \tilde{S}\ \texttt{\}} \\
X & ::= & x \mid f \mid \texttt{this.}f \mid X\texttt{.}f \mid X\texttt{[}E\texttt{]} \\
E & ::= & X \mid \texttt{this} \mid \texttt{null} \mid B \mid \texttt{0} \mid \texttt{1} \mid E_1\ \texttt{+}\ E_2 \mid \cdots \mid \\
 & & \texttt{new}\ n\texttt{(}\tilde{E}\texttt{)} \mid E\texttt{.}f \mid E\texttt{.}m\texttt{(}\tilde{E}\texttt{)} \mid \texttt{new [}E\texttt{]} \mid E_1\texttt{[}E_2\texttt{]} \\
B & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{!}B \mid B_1\ \texttt{\&\&}\ B_2 \mid B_1\ \texttt{==>}\ B_2 \mid \\
 & & \texttt{(\textbackslash forall}\ x\ \texttt{=}\ E_1\texttt{..}E_2\texttt{;}\ B\texttt{)} \mid E_1\ \texttt{==}\ E_2 \mid \cdots
\end{array}
$$

(a) External fragment, for the user in step 0

$$
B\ ::=\ \cdots \mid \texttt{Perm(}X\texttt{,}q\texttt{)} \mid B_1\ \texttt{**}\ B_2 \mid \texttt{(\textbackslash forall*}\ x\ \texttt{=}\ E_1\texttt{..}E_2\texttt{;}\ B\texttt{)}
$$

(b) Internal fragment, for VeyMont in step 1b

Fig. 5. Grammar of μPVL (types omitted for simplicity)

(formal parameters), and an optional list of statements (body). A method without a body is abstract (for external trusted code). An *annotation* A is a precondition, a postcondition, or a method invariant.

– **Statements, variables, expressions:** A *statement* S is an assertion, an assignment, a method call, a conditional choice, or a conditional loop. A *variable* X is a variable name, a (qualified) field name, or a (qualified) array cell. An *expression* E is a variable, a self reference, a null reference, a Boolean expression, a primitive value/operation, an object constructor call/ field access/method call, or an array constructor call/cell access. In Boolean expressions, light grey shading indicates that implication and quantification can be used only in annotations, assertions, and loop invariants.

Regarding the internal fragment, let q range over "fractions" between 0 (exclusive) and 1 (inclusive). Effectively, the grammar of Boolean expressions in Fig. 5a is extended to the grammar of *permission-based, concurrent separation logic* [12,14] in Fig. 5b to support ownership-like assertions for mutable data. That is, `Perm(`X`,`q`)` indicates that an annotated piece of code has read permission for X (if $0 < q < 1$) or read+write permission (if $q = 1$); the sum of different fractions for the same variable can never exceed 1. Operators `**` and `\forall*` are the standard separating conjunction and separating quantification in separation logic [40,45]. Regarding notation, `requires Perm(`X_1`,`q_1`) ** Perm(`X_2`,`q_2`);` is equivalent to `requires Perm(`X_1`,`q_1`); requires Perm(`X_2`,`q_2`);`.

Remark 1. μPVL is also statically typed, but as type checking is not a contribution of this paper, we omit types to keep the presentation of μPVL concise.

4.2 Linter (Step 1a)

The second main component of step 1a is a *linter*. It checks if P_{seq} has a parallelisable structure. This is needed for applying the transformation rules in step 2 (Sect. 4.5). The linter checks the following syntactic conditions:

1. P_{seq} has a class `SeqProgram` that consists of k fields $(f_1, \ldots, f_k)$, a constructor (m_1), a main method `run` (m_2), and any auxiliary methods $(m_3, \ldots, m_l)$. All fields are instances of classes; all methods (except the constructor) are parameterless and non-recursive. The constructor initialises all fields.
2. For every assignment in $m_2, \ldots, m_l$: (a) the left-hand side is of the form $f_i.X$; (b) at most one field f_j occurs in the right-hand side. For instance, `s1.x = 5` and `s2.y = s1.x + 4` and `s1.a.b.c = 5` are fine; `s1.x = s1.x + s2.y` is not.
3. For every if/while-statement in $m_2, \ldots, m_l$: (a) the condition is of the form E_1 `&& ... &&` E_k; (b) f_i is the only field that occurs in every E_i. For instance, `s1.x == 5 && s2.y == 9` is fine; `s1.x + 4 == s2.y` is not.
4. For every method call on field f_i in $m_2, \ldots, m_l$: f_i is the only field that occurs in the arguments. For instance, `s1.foo(s1.x)` is fine; `s1.foo(s2.y)` is not.

These syntactic conditions constrain only class `SeqProgram` (i.e., structural parallelisability depends only on `SeqProgram`). Other classes in P_{seq} are unrestricted.

Remark 2. In our experience (e.g., Sect. 5), conditions 1–4 are straightforward to meet. Notably, many potential violations can be fixed using auxiliary fields. For instance, `s1.x = s1.x + s2.y` violates condition 2, but it can be rewritten to `s1._y = s2.y; s1.x = s1.x + s1._y`, which is functionally equivalent. Similarly, `if (s1.x + 4 == s2.y) { ... }` violates condition 3, but it can be rewritten to:

```
s1._y = s2.y; s2._x = s1.x; if (E1 && E2) { ... }
```

with E_1 = `s1.x + 4 == s1._y` and E_2 = `s2._x + 4 == s2.y`. (In these examples, `s1._y` and `s2._x` are fresh.) A *complete formal characterisation* of the class of sequential-ish programs that can be rewritten in this way, including a mechanical procedure to automatically perform the necessary rewrites to meet the conditions, is still an open problem.

Remark 3. The conditions checked by the linter result from our design decision to target coarse-grained parallelism (i.e., every top-level field of `SeqProgram` is turned into a separate thread in step 2; Sect. 4.5) instead of fine-grained (e.g., loop parallelisation). We discuss their combination in Sect. 6.

4.3 Annotator (Step 1b)

The main component of step 1b is an *annotator*. It inserts additional annotations into the input program to be able to check if P_{seq} has parallelisable behaviour (step 1c; Sect. 4.4), in terms of two properties:

i. *Alias freedom.* For every piece of mutable data in P_{seq} (object fields and array cells), VeyMont inserts ownership-like assertions to specify that it cannot be aliased. As a result, the threads of P_{par} will operate on disjoint fragments of memory, so data races are avoided.

	sequential-ish	→	parallel: `s1Thread`	parallel: `s2Thread`
store	`s1.x = 5;`	→	`s1.x = 5;`	`/* skip */`
transfer	`s2.y = s1.x + 4;`	→	`s1_s2.write(s1.x + 4);`	`s2.y = s1_s2.read();`
if/while	`if (s1.x == 5 &&`	→	`if (s1.x == 5`	`if (/* skip */`
	`s2.y == 9) {`	→	`/* skip */) {`	`s2.y == 9) {`
call	`s1.foo(s1.x); }`	→	`s1.foo(s1.x); }`	`/* skip */ }`

Fig. 6. Summary of transformation rules for statements, by example

Example 1. VeyMont amends the constructor of class `Storage` in Fig. 3:

```
ensures Perm(v,1) ** Perm(temp,1);
Storage(int v_init) { ... }
```

VeyMont amends the methods of class `SeqProgram`, too:

```
ensures B_own
SeqProgram(int v) { ... }

context B_own
void run() { ... }
```

where
B_{own} = `Perm(s1, 1) ** Perm(s2, 1) **`
`Perm(s1.v, 1) ** Perm(s2.v, 1) **`
`Perm(s1.temp, 1) ** Perm(s2.temp, 1)`

The key idea is to assert write permissions of 1, for all data, everywhere. As the sum of fractional permissions can never exceed 1, there can be no aliases.

ii. *Branch unanimity.* For every condition of the form E_1 `&& ... &&` E_k of if/while-statements in methods $m_2, \dots, m_l$ of class `SeqProgram`, VeyMont inserts an assertion of the form E_1 `==` E_2 `&& ... &&` E_{k-1} `==` E_k (i.e., $\forall_{0 \leq i < j \leq k}$ E_i `==` E_j) to specify that, when $E_1, \dots, E_k$ are evaluated, they are all equivalent. This implies that the threads of P_{par} all choose the same branch.

Example 2. VeyMont amends the while-statement in method `run` in Fig. 4:

```
loop_invariant eq_grids(p1, p2);
loop_invariant p1.inPlay == p2.inPlay;
while (p1.inPlay && p2.inPlay) { ... }
```

Alias freedom and branch unanimity are sufficient to guarantee that functional correctness is preserved through parallelisation, and that parallelisation does not introduce deadlocks [30]; we clarify the importance of the latter after having discussed parallelisation (step 2; Sect. 4.5).

4.4 VerCors (Step 1c)

The main component of step 1c is the VerCors–Viper–Z3 tool stack [9,11] (whose language, PVL, is a superset of μPVL). To check that P_{seq} is functionally correct and has parallelisable behaviour, it verifies the truth of the user's annotations (step 0) and VeyMont's (step 1b).

4.5 Code Generator (Step 2)

The main component of step 2 is a *code generator* into Java. Non-`SeqProgram` classes in P_{seq} are copied to P_{par}, while `SeqProgram` is parallelised into classes

```
1 b.x = a.x;
2 a.y = b.y;
3 if (a.x >= a.y &&
4     b.x <= b.y) {
5   a.z = b.z;
6 } else {
7   b.z = a.z;
8 }
```

(a) Sequential-ish

```
1 a_b.write(a.x);
2 a.y = b_a.read();
3 if (a.x >= a.y
4     /* skip */) {
5   a.z = b_a.read();
6 } else {
7   a_b.write(a.z);
8 }
```

(b) Parallel: `aThread`

```
1 b.x = a_b.read();
2 b_a.write(b.y);
3 if (/* skip */
4     b.x <= b.y) {
5   b_a.write(b.z);
6 } else {
7   b.z = a_b.read();
8 }
```

(c) Parallel: `bThread`

Fig. 7. Example of a sequential-ish program whose parallelisation can deadlock. VeyMont statically detects this and reports an error instead.

f_1`Thread`, ..., f_k`Thread` (each f_i is a field of `SeqProgram`), and class `ParProgram` for forking. The methods of each f_i`Thread` are derived from methods $m_2, \ldots, m_l$ of `SeqProgram`, by applying the transformations in Fig. 6 to every statement S:

- If S is an assignment, then *due to condition 2 of the linter* (Sect. 4.2), S contains the field either of one thread ("store") or of two threads ("transfer"). In the former case, S is added to the thread; in the latter case, a `write`/`read` on a `Channel` are added to the threads. Nothing is added to other threads.
- If S is an if/while-statement, then *due to condition 3 of the linter*, for every thread, S contains a corresponding subcondition. An if/while-statement with exactly that corresponding subcondition is added to every thread.
- If S is a call, then *due to condition 4 of the linter*, S contains the field of one thread. S is added to that thread. Nothing is added to other threads.

The theoretical foundations of our method ensure that if steps 1a, 1b, and 1c have succeeded, then the transformation rules in Fig. 6 indeed result in a functionally correct, deadlock-free parallel program [30].

Remark 4. To illustrate the importance of branch unanimity (Sect. 4.3) to guarantee that parallelisation does not introduce deadlocks, Fig. 7 shows a sequential-ish program (i.e., the body of method `run` of class `SeqProgram` with top-level fields `a` and `b`). This program meets the conditions of the linter, so it has a parallelisable structure; its parallelisation consists of `aThread` and `bThread`.

However, whether or not `aThread` and `bThread` can deadlock crucially depends on the initial values of `a.x` and `b.y` (intentionally omitted from Fig. 7):

- If `a.x` and `b.y` are initially equal, then branch unanimity is satisfied (no deadlock): after the first two assignments, `a.x >= a.y` and `b.x <= b.y` are both true. Subsequently, `aThread` and `bThread` both enter their then-branches, so `aThread` reads and `bThread` correspondingly writes.
  Thus, VeyMont (step 1c) reports no error when `a.x == b.y` initially.
- If `a.x` and `b.y` are initially unequal, then branch unanimity is violated (deadlock): `a.x >= a.y` and `b.x <= b.y` are either true and false, or false and true. In the former case, `aThread` enters its then-branch, but `bThread` enters its else-branch. At this point, `aThread` and `bThread` both expect to read, but neither one of them will write, so they are stuck forever.
  Thus, VeyMont (step 1c) reports an error when `a.x != b.y` initially.

We note that VeyMont guarantees deadlock freedom, but not starvation freedom: at any point in time, either all threads have terminated, or at least one thread is still running, modulo exceptions (e.g., division by zero).

5 Evaluation

Applicability. We used VeyMont to verify and parallelise sequential-ish programs for three classical distributed algorithms, for various numbers of threads n:

- In **two-phase commit** (2PC) [47], 1 Client and $n-1$ Servers cooperate to fulfil a joint query in a distributed database. First, the Client shares the query with the Servers. Next, the Servers locally run the query and report success/failure back to the Client. Only if all Servers succeeded will the Client instruct them to commit, and otherwise to abort. We successfully verified that the Clients consistently commit, for $n \in \{3, 5, 8, 12, 17\}$.
- In anonymous **election** (probabilistic version of Peleg's algorithm [43] in the style of Itai and Rodeh [26]), n symmetric threads try to elect a unique leader among them. The algorithm proceeds in rounds. In every round, every thread picks a random number from some fixed range (trusted code) and shares it with every other thread. If there is a unique highest number, then the thread that picked it declares itself the leader; otherwise, another round ensues. We verified that a unique leader is elected upon termination, for $n \in \{3, 5, 8\}$.
- In **consensus** [6], n symmetric threads try to reach agreement about a common value. First, the threads share their locally preferred values. Next, every thread computes the globally preferred value (by majority); this becomes the common value. The complication is that *threads can fail*: nondeterministically (abstract methods), they can share the wrong locally preferred value and/or compute the wrong globally preferred value. We successfully verified that all threads set the right globally preferred value when the number of failures is at most $\lfloor n/4 \rfloor$, for $n \in \{3, 5\}$; this is a classical result.

As a proxy of effort, Fig. 8 shows ratios of numbers of annotations ("spec") vs. program elements ("impl"). They are *below* 1; by comparison, Wolf et al. [52] recently report ratios of 2.69–3.16 to deductively verify parallel programs using a tool based on traditional methods. This is first evidence that VeyMont indeed significantly reduces the annotation burden.

Figure 8 also presents the mean run times (of 30 runs) of VeyMont for step 1c and in total (using: Intel i7-8569U CPU with 4 physical/4 virtual cores at 2.8 GHz; 16 GB memory). We can make two main observations. First, the run times are dominated by step 1c (actual verification). For instance, step 1c consumes $\frac{6.8}{8.0} = 85\%$ of the run time for 2PC ($n = 3$) and as much as $\frac{62.2}{63.9} = 97\%$ for 2PC ($n = 17$). Second, parallelisation itself is relatively cheap. For instance, it takes less than 1.2 seconds for 2PC ($n = 3$) and less than 1.7 seconds for 2PC ($n = 17$).

	$\frac{\textbf{spec}}{\textbf{impl}}$	**1c**	**total**
2PC (n=3)	32/75	6.8	8.0
2PC (n=5)	42/91	8.9	10.1
2PC (n=8)	57/115	13.9	15.2
2PC (n=12)	77/147	26.4	27.8
2PC (n=17)	102/187	62.2	63.9

	$\frac{\textbf{spec}}{\textbf{impl}}$	**1c**	**total**
election (n=3)	22/49	7.2	8.5
election (n=5)	28/69	14.3	16.2
election (n=8)	37/99	61.5	65.4
consensus (n=3)	72/73	9.8	11.3
consensus (n=5)	90/104	41.0	42.5

Fig. 8. Case studies in applicability: ratio of number of annotations vs. program elements ($\frac{\textbf{spec}}{\textbf{impl}}$) and mean VeyMont run times in seconds (**1c**, **total**). Program elements are: class headers, fields, method headers, and statements.

Efficiency. We compared the performance of VeyMont-generated parallel programs in Java with third-party reference implementations. Our aim was to study if the synchronisation mechanism in generated parallel programs is sufficiently lightweight to be competitive. We use different programs than above, as no third-party reference implementations were available for 2PC/election/consensus.

We took the following approach. First, we selected two parallel programs from the *CLBG database* [1]: `binary-trees` (parallel tree walk) and `k-nucleotide` (parallel pattern matching of molecule sequences against a DNA string). Next, for each program: **(1)** we extracted the data sharing patterns among threads in the CLBG reference implementation and wrote them as a sequential-ish program in μPVL; **(2)** we "completed" the sequential-ish program by adding abstract methods to represent all purely sequential computations; **(3)** we generated parallel programs in Java using VeyMont; **(4)** we concretised the abstract methods in Java with trusted sequential CLBG code; **(5)** we ran the CLBG version and the VeyMont version to compare performances, using CLBG-standardised input, with various numbers of threads. We note that we did not prove functional correctness; this is beyond the scope of these performance comparisons.

We ran the resulting executables on three different machines: *Cartesius* (Intel E5-2690 v3 CPU with 16 physical cores), *MacBook* (Intel i7-8569U CPU with 4 physical/4 virtual cores), and *VM* [28] (1 virtual core). Figure 9 show our results as *speed-ups* of VeyMont versions relative to CLBG versions, computed as $\frac{\mu_{\text{CLBG}}}{\mu_{\text{VeyMont}}}$, where μ_{VeyMont} and μ_{CLBG} are the mean run times (of 100 runs) of a VeyMont and a CLBG version; $\frac{\mu_{\text{CLBG}}}{\mu_{\text{VeyMont}}} < 1$ means that a VeyMont version was slower.

We can make two main observations. First, although the VeyMont versions tend to be somewhat slower than the CLBG versions, the slowdown is generally less than 10%. We conjecture that there is a substantial class of programs for which a 10% slowdown is a fine price for better verifiability of functional correctness and deadlock freedom. Second, different machines exhibit different performance; a deeper study is needed to understand what exactly causes this.

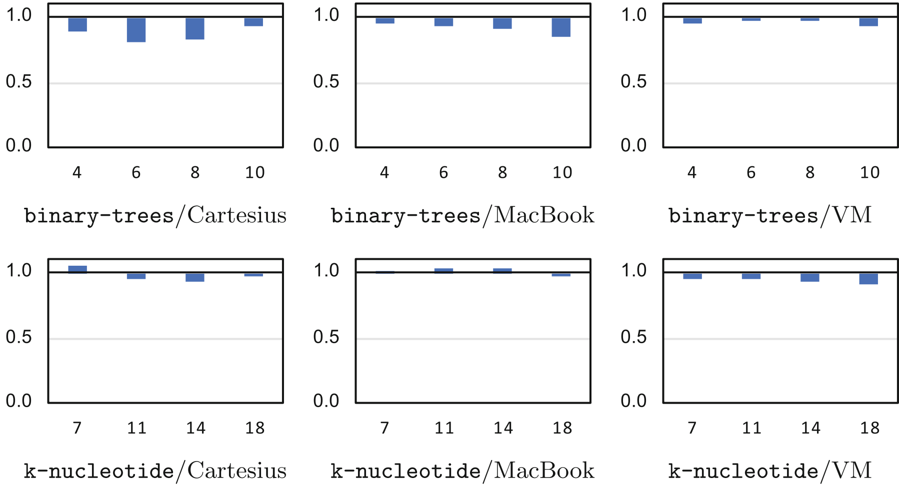

Fig. 9. Case studies in efficiency: the x-axis indicates the number of threads; the y-axis indicates the speed-up of VeyMont versions relative to CLBG versions.

6 Future Work

We presented VeyMont: a deductive verification tool that aims to make reasoning about functional correctness and deadlock freedom of parallel programs (relatively complex) as easy as that of sequential programs (relatively simple).

Our most-wanted feature for VeyMont is to support *parametrisation* (e.g., election generically for n threads instead of specifically for $3, 5, 8, \ldots$). However, parametrised verification is known to be undecidable in general [3,48]. The study of this topic (e.g., identification of decidable fragments) has become a research area of its own over the past decade; the book by Bloem et al. gives an extensive overview [7,8]. Thus, an extension of VeyMont to support parametrisation is highly non-trivial. It is our main direction for future work.

Other future work pertains to a relaxation of alias freedom and branch unanimity in the theoretical foundations of VeyMont [30]. Such a relaxation allows VeyMont to be more flexible about read/write permissions (e.g., improve support for read-only shared arrays), but maintaining the same strong guarantees.

Inspired by methods for distributed systems, VeyMont targets coarse-grained parallelism among threads (i.e., whole-program parallelisation) instead of fine-grained parallelism among tasks (e.g., loop parallelisation). We are keen to explore the combination of both approaches. A first step would be to mix VeyMont with the VerCors-based work of Blom et al. [10] on verification of loop parallelisation. Beyond that, it is interesting to extend VeyMont with complementary techniques. For instance, Raza et al. [44] developed a technique to infer dependencies among statements in sequential programs to allow their parallel execution (like us), but at the level of tasks (unlike us). Their technique and ours have different strengths: we can split the conditions of if/while-statements across

separate threads, which Raza et al. cannot (they assume indivisible conditions); conversely, Raza et al. can parallelise recursive divide-and-conquer algorithms in separate tasks, which we cannot (we assume fixed numbers of processes).

Finally, more on the engineering side, we are also keen to investigate to what extent alternative deductive verification back-ends instead of VerCors can offer value both to users (e.g., faster verification) and to researchers (i.e., in principle, any deductive verification tool for sequential programs can be combined with VeyMont's method to reason about functional correctness of parallel programs).

A Appendix: Parallelisation of Tic-Tac-Toe

The following listing shows the two threads for top-level fields `p1` and `p2` in the parallelisation of the sequential-ish program in Fig. 4, generated by VeyMont (functionally correct and deadlock-free). We note that `p1Thread` and `p2Thread` have "opposite" behaviour in their methods `turn1` and `turn2`.

```
1  class p1Thread extends Thread {
2    Player p1;
3    MoveChannel p1_p2;
4    MoveChannel p2_p1;
5
6    p1Thread(int m, int n,
7      MoveChannel p1_p2,
8      MoveChannel p2_p1) {
9
10     this.p1 = new Player(m, n, ...);
11     this.p1_p2 = p1_p2;
12     this.p2_p1 = p2_p1;
13   }
14
15   void turn1() {
16     p1.think();
17     p1.play();
18     p1_p2.write(p1.move.clone());
19   }
20
21   void turn2() {
22     p1.think(); // in the background
23     p1.move = p2_p1.read();
24     p1.play(); // to update
25   }
26
27   public void run() {
28     while(p1.inPlay) {
29       turn1();
30       if (p1.inPlay) {
31         turn2();
32 } } } }
```

```
33 class p2Thread extends Thread {
34   Player p2;
35   MoveChannel p1_p2;
36   MoveChannel p2_p1;
37
38   p2Thread(int m, int n,
39     MoveChannel p1_p2,
40     MoveChannel p2_p1) {
41
42     this.p2 = new Player(m, n, ...);
43     this.p1_p2 = p1_p2;
44     this.p2_p1 = p2_p1;
45   }
46
47   void turn1() {
48     p2.think(); // in the background
49     p2.move = p1_p2.read();
50     p2.play(); // to update
51   }
52
53   void turn2() {
54     p2.think();
55     p2.play();
56     p2_p1.write(p2.move.clone());
57   }
58
59   public void run() {
60     while(p2.inPlay){
61       turn1();
62       if (p2.inPlay) {
63         turn2();
64 } } } }
```

The remaining classes that are part of the parallelisation are:

- `ParProgram`: This class is responsible for creating channels and starting the threads. It is very similar to class `ParProgram` in Fig. 3b
- `Player`, `Move`: These classes are straightforward Java versions of the μPVL versions in Fig. 4.

References

1. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
2. Aiken, A., Nicolau, A.: Optimal loop parallelization. In: PLDI, pp. 308–317. ACM (1988)
3. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. **22**(6), 307–309 (1986)
4. Apt, K.R., Olderog, E.-R.: Fifty years of Hoare's logic. Formal Aspects Comput. **31**(6), 751–807 (2019). https://doi.org/10.1007/s00165-019-00501-3
5. Baudin, P., et al.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021)
6. Berman, P., Garay, J.A., Perry, K.J.: Towards optimal distributed consensus (extended abstract). In: FOCS, pp. 410–415. IEEE Computer Society (1989)
7. Bloem, R., et al.: Decidability of Parameterized Verification. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, San Rafael (2015)
8. Bloem, R., et al.: Decidability in parameterized verification. SIGACT News **47**(2), 53–64 (2016)
9. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
10. Blom, S., Darabi, S., Huisman, M., Safari, M.: Correct program parallelisations. Int. J. Softw. Tools Technol. Transf. **23**(5), 741–763 (2021). https://doi.org/10.1007/s10009-020-00601-z
11. Blom, S., Huisman, M.: The VerCors tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_9
12. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL, pp. 259–270. ACM (2005)
13. Boulet, P., Darte, A., Silber, G.-A., Vivien, F.: Loop parallelization algorithms: from parallelism extraction to code generation. Parallel Comput. **24**(3–4), 421–444 (1998)
14. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
15. Brookes, S.: A semantics for concurrent separation logic. Theor. Comput. Sci. **375**(1–3), 227–270 (2007)
16. Burke, M., Cytron, R.: Interprocedural dependence analysis and parallelization. In: SIGPLAN Symposium on Compiler Construction, pp. 162–175. ACM (1986)
17. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL, pp. 263–274. ACM (2013)
18. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distributed Comput. **31**(1), 51–67 (2018)
19. Castro, D., Hu, R., Jongmans, S.S., Ng, N., Yoshida, N.: Distributed programming using role-parametric session types in Go: statically-typed endpoint APIs for dynamically-instantiated communication structures. PACMPL, **3**(POPL), 29:1–29:30 (2019)
20. Din, C.C., Tapia Tarifa, S.L., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407, pp. 217–233. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_14

21. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
22. Hobor, A., Gherghina, C.: Barriers in concurrent separation logic. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 276–296. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19718-5_15
23. Hobor, A., Gherghina, C.: Barriers in concurrent separation logic: now with tool support! Log. Methods Comput. Sci., **8**(2) (2012)
24. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284. ACM (2008)
25. Hu, R., Yoshida, N.: Hybrid session verification through endpoint API generation. In: Stevens, P., Wasowski, A. (eds.) FASE 2016. LNCS, vol. 9633, pp. 401–418. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49665-7_24
26. Itai, A., Rodeh, M.: Symmetry breaking in distributive networks. In: FOCS, pp. 150–158. IEEE Computer Society (1981)
27. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4
28. Jacobs, S., Reynolds, A.: TACAS 22 Artifact Evaluation VM - Ubuntu 20.04 LTS (2021). https://doi.org/10.5281/zenodo.5562597
29. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Trans. Program. Lang. Syst. **5**(4), 596–619 (1983)
30. Jongmans, S., van den Bos, P.: A predicate transformer for choreographies. In: ESOP 2022. LNCS, vol. 13240, pp. 520–547. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99336-8_19
31. Lamport, L.: The parallel execution of DO loops. Commun. ACM **17**(2), 83–93 (1974)
32. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006)
33. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: POPL, pp. 201–214. ACM Press (1997)
34. López, H.A., et al.: Protocol-based verification of message-passing parallel programs. In: OOPSLA, pp. 280–298. ACM (2015)
35. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: compile-time API generation of distributed protocols with refinements in F#. In: CC, pp. 128–138. ACM (2018)
36. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: CC, pp. 98–108. ACM (2017)
37. Ng, N., Yoshida, N.: Pabble: parameterised scribble. Serv. Oriented Comput. Appl. **9**(3–4), 269–284 (2015)
38. Oancea, C.E., Rauchwerger, L.: Logical inference techniques for loop parallelization. In PLDI, pp. 509–520. ACM (2012)
39. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. **375**(1–3), 271–307 (2007)
40. O'Hearn, P.: Separation logic. Commun. ACM **62**(2), 86–95 (2019)
41. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**, 319–340 (1976)
42. Owicki, S.S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Commun. ACM **19**(5), 279–285 (1976)

43. Peleg, D.: Time-optimal leader election in general networks. J. Parallel Distrib. Comput. **8**(1), 96–99 (1990)
44. Raza, M., Calcagno, C., Gardner, P.: Automatic parallelization with separation logic. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 348–362. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00590-9_25
45. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society (2002)
46. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP, volume 74 of LIPIcs, pp. 24:1–24:31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
47. Skeen, D.: Nonblocking commit protocols. In: SIGMOD Conference, pp. 133–142. ACM Press (1981)
48. Suzuki, I.: Proving properties of a ring of finite-state machines. Inf. Process. Lett. **28**(4), 213–214 (1988)
49. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In: PLDI, pp. 177–187. ACM (2009)
50. VerCors Wiki. https://github.com/utwente-fmt/vercors/wiki
51. VeyMont Artifact. https://doi.org/10.5281/zenodo.7410640
52. Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra: modular specification and verification of Go programs. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 367–379. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_17