An Exercise in Verifying Sequential Programs with VerCors

Sebastiaan J.C. Joosten
Wytse Oortwijn
Mohsen Safari
Marieke Huisman
University of Twente, The Netherlands
{s.j.c.joosten,w.h.m.oortwijn,m.safari,m.huisman}@utwente.nl

Abstract

Society nowadays relies heavily on software, which makes verifying the correctness of software crucially important. Various verification tools have been proposed for this purpose, each focusing on a limited set of tasks, as there are many different ways to build and reason about software. This paper discusses two case studies from the VerifyThis2018 verification competition, worked out using the VerCors verification toolset. These case studies are sequential, while VerCors specialises in reasoning about parallel and concurrent software. This paper elaborates on our experiences of using VerCors to verify sequential programs. The first case study involves specifying and verifying the behaviour of a gap buffer; a data-structure commonly used in text editors. The second case study involves verifying a combinatorial problem based on Project Euler problem #114. We find that VerCors is well capable of reasoning about sequential software, and that certain techniques to reason about concurrency can help to reason about sequential programs. However, the extra annotations required to reason about concurrency bring some specificational overhead.

ACM Reference Format:
Sebastiaan J.C. Joosten, Wytse Oortwijn, Mohsen Safari, and Marieke Huisman. 2018. An Exercise in Verifying Sequential Programs with VerCors. In *FTfJP (ISSTA Companion/ECOOP Companion'18), Amsterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236454.3236479

1 Introduction

Now that society relies on software, verification of software is of crucial importance. Different tools help in software verification, and many of those have been created with only a limited set of verification tasks and targets in mind. This is a problem for both developers and users of verification tools: users risk investing effort in learning and using a tool that does not enable them to verify the code, or properties of that code, that they are most interested in. Developers risk spending time on improving parts of the tool that have no real impact for users. Case studies in such tools help bridge the gap between users and developers, by finding blind-spots: programs that turn out to be hard to verify because tools lack support for it. This case study works on problems created by people who are not the tool developers, making the problem particularly well suited for taking us out of our comfort zone.

To support developers in verifying their software, several verification tools have been created. These tools allow the programmer to annotate their code with specifications, expressing the intended behaviour of the program, so that the tool can mechanically verify that the program behaves as specified. In 2012, the VerifyThis competition was held to test these tools, resulting in a special issue of case studies [9]. Participating teams used various tools, including KeY [1], VeriFast [10], KIV [7], Why3 [8], GNATprove (based on Why3), AutoProof [13], and VerCors [3]. In our case study, we use a tool for verification of concurrent software: VerCors [3].

As VerCors is intended to reason about concurrent software, it has been used to verify different kinds of concurrent code. This includes several Java control structures such as different types of locks as well as (lock-free) concurrent collections [2–5, 12]. In contrast to the work mentioned in this paragraph, we focus on verification of sequential programs.

This paper discusses two sequential case studies, which were given as verification challenges as part of the VerifyThis competition in 2018. Even though sequential verification may be considered a special case of concurrent verification, the techniques to reason about concurrency are generally different from sequential verification techniques, especially when applied in a tool. All of the authors are involved in the development of VerCors, but took on the role of users during the competition. The purpose of these case studies

ISSTA/ECOOP, , Amsterdam, Netherlands

ACM ISBN 978-1-4503-5939-9/18/07…$15.00
https://doi.org/10.1145/3236454.3236479

is to address blind-spots that arise because VerCors is only developed with concurrency in mind.

The case-study has some surprising results. Unsurprisingly, using the wrong tool for the job hinders the verification effort. However, this effect is very limited, and most of the verification is surprisingly similar as to how it would be for sequential verification. Moreover, some of the features intended for parallelism actually helped verification of sequential programs: In the second example we used the parallel-block construct in VerCors to prove quantified facts about sequential programs. We give recommendations on how VerCors should be improved, and highlight some features of VerCors that might be useful for other programs.

This paper is structured as follows. Section 2 elaborates on the verification capabilities and workflow of VerCors. Section 3 presents the first case study: the behaviour of a *gap buffer*, a data structure often used in e.g. text editors. Section 4 discusses our second case study, which involves counting how many sequences of black and red tiles of length 50 can be constructed, given some restrictions on their construction (based on Project Euler problem #114). Section 5 concludes by discussing the lessons learned during the verification of the VerifyThis2018 challenges and gives future work.

2 The VerCors Toolset

VerCors is a toolset that specialises in verifying programs written in high-level languages with parallelism and concurrency features [6]. Notably, VerCors is able to reason about locks and fork/join concurrency in Java, but can also handle advanced language features such as GPU kernels with barriers and atomic operations in OpenCL, and compiler directives for automated parallelism in a subset of OpenMP for C. VerCors allows to verify race freedom, memory safety and functional correctness of these languages and uses separation logic with permission accounting as its logical foundation.

2.1 Workflow

Figure 1 gives an overview of the workflow of VerCors. VerCors takes programs as input written in high-level languages and annotated with JML-like specifications. Multiple front-end languages are supported, including (subsets of) Java, PVL, OpenCL, and OpenMP for C. PVL stands for Prototypal Verification Language and is a verification language used by the developers to prototype new verification features.

The VerCors toolchain transforms input programs and their specifications into the intermediate language of VerCors. The main goal of VerCors is to lift existing verification technology towards high-level languages and concurrency features, rather than developing new verification technology. In essence, VerCors is a set of language transformations that transform programs in this intermediate language into input for the Viper toolset [11], which is the back-end of VerCors. The Viper toolset uses the intermediate verification language Silver as its input language and allows to reason about programs with persistent mutable state, annotated with separation logic-style specifications. Viper eventually encodes the verification problem into an SMT problem.

VerCors can easily be extended with new front-end languages with parallelism or concurrency features, by translating them into the intermediate language of VerCors. Since all language transformations work on this intermediate representation, full verification support is then available for all the translated language features in the new language.

2.2 Obtaining and using VerCors

The source code of VerCors is available online[1], as well as a list of verified case studies and examples[2]. The latter webpage includes an online interface that allows to try VerCors online. The two verification challenges discussed in Sections 3 and 4 can also be found and verified online via this webpage.

3 Challenge One: Gap Buffer

The first challenge of VerifyThis2018 involves verifying four basic operations on a *gap buffer*, which is a data-structure commonly used in text editors to move the text cursor and to add or delete characters at the cursor's location.

A gap buffer is an integer array a, together with two indices $0 \leq l \leq r < a.length$, such that $a[l], \ldots, a[r]$ is a *gap*: a region of unused entries in a. The index l represents the current position of the cursor and the contents of the gap buffer is represented as $a[0], \ldots, a[l-1], a[r], \ldots, a[a.length-1]$.

3.1 Problem description

Algorithm 1 gives the implementation of four basic operations on gap buffers, namely: `left` and `right` for moving the text cursor to the left and right, respectively; `insert` for inserting a character at the position of the cursor; and `delete` for deleting the character at the cursor's position. These four operations assume the array a to be global, as well as the indices l and r. Moreover, while inserting a character with `insert` it may happen that the gap is empty, i.e. that $l = r$. In that scenario, the procedure `grow` is called on line 22, which enlarges the array a by creating a gap of size K.

The verification challenge is: verify in a modular way that the gap buffer behaves as intended with respect to the operations described in Algorithm 1. This intended behaviour should be specified in terms of a continuous representation of the buffers' content, for example as a sequence of characters.

3.2 Approach

During the VerifyThis competition we worked this example out in PVL, our prototype verification language. We chose this language because it allowed us to work more efficiently,

[1] The source code is available at: https://github.com/utwente-fmt/vercors. The examples in this paper have been verified with commit number *b2c116b*.
[2] A list of case studies and verified examples is available at: http://ctit-vm2.ewi.utwente.nl.

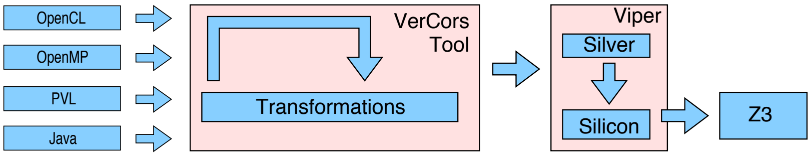

Figure 1. The workflow of the VerCors toolset.

```
void left() {
  if (l ≠ 0) {
    l := l − 1;
    r := r − 1;
    a[r] := a[l];
  }
}
void right() {
  if (r ≠ a.length − 1) {
    a[l] := a[r];
    l := l + 1;
    r := r + 1;
  }
}
void delete() {
  if (l ≠ 0) {
    l := l − 1;
  }
}
void insert(int c) {
  if (l = r) {
    grow();
  }
  a[l] := c;
  l := l + 1;
}
void grow() {
  int n := a.length;
  int[] b := new int[n + K];
  for (i := 0 to l) {
    b[i] := a[i];
  }
  for (i := r to n) {
    b[i + K] := a[i];
  }
  r := r + K;
  a := b;
}
```

Algorithm 1: The basic operations on gap buffers.

given the time limit of 90 minutes. VerCors also allows program verification in Java or C, and this example could have been worked out in either of those languages as well.

Our general approach is to represent the buffer's content as a sequence xs of integers and to verify the following:

- After calling `left`, `right` and `grow` the buffer's content is still represented by xs.
- After calling `delete` the buffer's content is represented by $xs[..l] + xs[l+1..]$, provided that a delete was possible, with l the cursor location after the call to `delete`[3].
- After calling `insert` the buffer's content is represented by $xs[..l–1]+\{c\}+xs[l–1..]$, with c the inserted character and l the cursor location after the call to `insert`.

The sequence slicing notations $xs[n..]$ and $xs[..n]$ are currently not natively supported by VerCors. Instead, we implemented these using two auxiliary operations over sequences, $\texttt{Skip}(xs, i)$ and $\texttt{Take}(xs, i)$, to skip and take the first i entries of the given sequence xs, respectively. We additionally needed to prove some lemmas to relate the length and contents of xs to $\texttt{Skip}(xs, i)$ and $\texttt{Take}(xs, i)$ for any index i. The slicing notations are however used in the remainder of this section for presentational convenience.

$$\begin{aligned}
&\textbf{resource}\ \mathsf{Represents}(\textbf{seq<int>}\ xs) \triangleq \\
&\quad \mathsf{Perm}(l, \tfrac{2}{3}) \ast\ast \mathsf{Perm}(r, \tfrac{2}{3}) \ast\ast \mathsf{Perm}(a, \tfrac{2}{3}) \ast\ast a \neq \mathsf{null} \ast\ast && (1)\\
&\quad 0 \leq l \leq r < a.length \ast\ast |xs| = a.length - (r - l) \ast\ast && (2)\\
&\quad (\forall^{*} i\,.\,0 \leq i < a.length \Rightarrow \mathsf{Perm}(a[i], 1)) \ast\ast && (3)\\
&\quad (\forall i\,.\,0 \leq i < l \Rightarrow a[i] = xs[i]) \ast\ast && (4)\\
&\quad (\forall i\,.\,r \leq i < a.length \Rightarrow a[i] = xs[i - (r - l)]); && (5)
\end{aligned}$$

Figure 2. The definition of the Represents predicate.

3.3 Solution

Algorithm 2 shows the annotated version of the gap buffer implementation, with the annotated specifications displayed in purple. The presented annotations are somewhat simplified for the sake of brevity; the full version can be seen and verified using the online interface of VerCors.

Recall that VerCors uses separation logic with permission accounting as its logical foundation, to enable reasoning over concurrent software. Therefore, annotations expressing *ownership* are required in addition to the annotations that express functional correctness. Ownership is specified via predicates of the form $\mathsf{Perm}(s, \pi)$, with s a shared-memory location (e.g. a class field) and π a *fractional permission* in the range $(0..1]$. Any ownership predicate with $\pi = 1$ expresses *write access* for the location s; whereas $\pi < 1$ expresses *read access* to s. Soundness of the underlying logic ensures that the total sum of permissions for any shared-memory location does not exceed 1, implying race freedom and memory safety.

In addition to these ownership predicates, the $\ast\ast$ connective is used in the annotated program, which is the *separating conjunction* from separation logic. A formula of the form $P \ast\ast Q$ expresses *disjointness* of the ownership expressed by P and Q, read as *"P and separately Q"*. The separating conjunction allows ownership predicates to be split and merged:

$$\mathsf{Perm}(s, \pi_1 + \pi_2) \Leftrightarrow \mathsf{Perm}(s, \pi_1)\ \ast\ast\ \mathsf{Perm}(s, \pi_2)$$

We define and use an auxiliary predicate $\mathsf{Represents}(xs)$ to specify the intended behaviour of the gap buffer using a sequence xs that represents the buffer's content. Figure 2 shows the definition of Represents. Notably, line (1) asserts read access for l, r and a, so that the remaining definition of Represents may read from these locations. The fractions

[3] The + operator has been overloaded to represent sequence concatenation, and $\{c\}$ is the singleton sequence containing c as its value.

```
1  given seq<int> xs;
2  context Perm(l, 1/3) ** Perm(r, 1/3);
3  context Represents(xs);
4  void left() {
5    if (l ≠ 0) {
6      unfold Represents(xs);
7      l := l − 1;
8      r := r − 1;
9      a[r] := a[l];
10     fold Represents(xs);
11   }
12 }
13 given seq<int> xs;
14 context Perm(l, 1/3);
15 requires Represents(xs);
16 ensures 0 = old(l) ⇒ Represents(xs);
17 ensures 0 < old(l) ⇒ Represents(xs[..l] + xs[l+1..]);
18 void delete() {
19   if (l ≠ 0) {
20     unfold Represents(xs);
21     l := l − 1;
22     fold Represents(xs[..l] + xs[l+1..]);
23   }
24 }
25 given seq<int> xs;
26 context Perm(l, 1/4) ** Perm(r, 1/3) ** K > 0;
27 context Represents(xs);
28 ensures l < r;
29 void grow(int K) {
30   // omitted for brevity
31 }
32 given seq<int> xs;
33 context Perm(l, 1/3) ** Perm(r, 1/3) ** K > 0;
34 requires Represents(xs);
35 ensures Represents(xs[..l−1] + {c} + xs[l−1..]);
36 void insert(int c, int K) {
37   if (l = r) {
38     grow(K) with { xs := xs };
39     unfold Represents(xs);
40   }
41   else {
42     unfold Represents(xs);
43   }
44   a[l] := c;
45   l := l + 1;
46   fold Represents(xs[..l−1] + {c} + xs[l−1..]);
47 }
```

Algorithm 2: The annotated operations of the gap buffer. The contract for `right` is the same as for `left` and is therefore omitted. An annotation of the form **context** P is an abbreviation for **requires** P; **ensures** P.

$\frac{2}{3}$ are somewhat arbitrary; the key point is that we did not express write permissions, so that we can be sure that a does not change and the contracts in Algorithm 2 may also still read from l and r (e.g. at lines 16, 17, and 28). Line (3) asserts write permission for every element of the array a using a $\forall^*$ quantifier, which is the *iterated separating conjunction*. Finally, lines (4) and (5) assert that a is properly abstracted by the integer sequence xs.

The predicate Represents(xs) is used in Algorithm 2 to specify the functional behaviour of the gap buffer, and is folded and unfolded in every operation to provide access to its contents. The sequence xs is specified using a **given** annotation, stating that xs is a *ghost parameter*—an extra argument only for the sake of specification. Ghost parameters should be instantiated when calling the corresponding method, e.g. on line 38. Also note that we slightly altered the implementations of `grow` and `insert` by making the gap size K a parameter, to simplify verification. The implementation of `grow` is omitted for brevity; the annotations mostly consist of loop invariants for the two for-loops, asserting that a is correctly copied into the new array b. The detailed, fully annotated PVL version of this example can be found in the list of verified examples we maintain online.

4 Challenge Two: Colored Tiles

The second problem is to verify a program, which produces a single number as output. The program computes in how many ways one can put 50 black and red tiles in a row, satisfying the requirement that no sequence of red tiles is shorter than 3. Rather than enumerating each such sequence, the program to be verified computes a number efficiently through two nested loops. The aim of this challenge is to verify that this result corresponds to the actual number of sequences of length 50. This problem is based on Project Euler problem 114 (which simply asks for the number).

4.1 Problem description

For this paper, we use the following language: We will refer to a particular sequence of tiles as a tiling (for instance, black-black is a tiling of length 2). A tiling is valid if it does not contain a sequence of red tiles shorter than 3.

In our encoding, we use ‘*true*’ for red, and ‘*false*’ for black, and refer to true and red interchangeably. Tilings are encoded as sequences of booleans. Validity of a tiling is defined as:

$$
\begin{aligned}
\mathrm{valid}(s,n) = (s.\mathit{length} = n) \land \\
\forall i.\ 0 \le i < n \land s[i] \quad &\Rightarrow \\
((i \ge 2 \quad &\land s[i-2] \land s[i-1]) \lor \\
(1 \le i < n-1 \quad &\land s[i-1] \land s[i+1]) \lor \\
(i \le n-2 \quad &\land s[i+1] \land s[i+2]) \\
)
\end{aligned}
$$

The program to be verified is given as Algorithm 3.

4.2 Approach

The full solution is roughly 200 lines and too large for inclusion here. We instead give a high-level description of what went into the proof.

We verify the program by creating a valid tiling sequence $res[i]$ to correspond to each number $count[i]$. Correctness of the result then follows from these properties:

- $res[j].length = count[j]$ for j between 0 and 50 (inclusive).
- Every tiling in $res[j]$ is a valid sequence of length j.
- $res[i][y] = res[i][z]$ implies $y = z$. This states that every tiling in $res[i]$ is unique.
- Every valid tiling of length j is contained in $res[j]$.

During the VerifyThis competition we completed verification of the first two properties given above, again by using PVL as programming language, within the 90 minutes that were given for the challenge.

For this case study, we have verified the first three properties. This shows that the calculated value is an upper bound. We attempted to prove the last property by showing that an arbitrary valid tiling of length n would be contained in $res[j]$, but this turned out to be very involved and is out of scope of this paper. As each value $res[j]$ is a sequence of tilings, our encoding means that res is a sequence of sequences of sequences of booleans.

```
1  int $count[51]$;   ▹ $count[i]$ is the number of valid rows of size $i$
2  $count[0] := 1$;                      ▹ One tiling of length zero
3  $count[1] := 1$;                ▹ One length 1 tiling with only black
4  $count[2] := 1$;                ▹ One length 2 tiling with only black
5  $count[3] := 2$;           ▹ Tiling of length 3 is all black or all red
6  for ($n := 4$ to 50) {
7    $count[n] := count[n-1]$;          ▹ Row starts with a black tile
8    for ($k = 3$ to $n-1$)
9        ▹ Start with $k$ red, then 1 black, then a previous sequence
10   {
11     $count[n] := count[n] + count[n-k-1]$;
12   }
13   $count[n] := count[n] + 1$;                      ▹ Or is red entirely
14 }
```

Algorithm 3: The colored tiles program.

```
$last := \{\,\}$;                ▹ Initialise as empty sequence of tilings
loop_invariant $last.length = j$;
loop_invariant $0 \le j \le res[n-1].length$;
loop_invariant $\forall y\,.\,0 \le y < j \Rightarrow \mathrm{valid}(last[y], n)$;
loop_invariant $\forall y\,.\,0 \le y < j \Rightarrow last[y][0] = false$;
loop_invariant $\forall y\,.\,0 \le y < j \Rightarrow (last[y] =$
$\{false\} + res[n-1][y])$;          ▹ Allows proving has_false later
loop_invariant unique($last$);
for ($j := 0$ to $res[n-1].length$) {
  uniqueness_implies_unequal($res[n-1], j, \{false\}$);
  $last := last + \{\{false\} + res[n-1][j]\}$;
}
```

Algorithm 4: A ghost loop as part of the solution.

Updates to elements in sequences are not supported in PVL. Therefore, we created the variable *last* that contains the tilings per loop. At the end of the loop, *last* is added as final element to *res*. Throughout loop iteration n, $count[n] = last.length$. To maintain this property, a loop is added at each position where *count* increases.

An example of such a loop is given in Algorithm 4. This is taken from the solution, with details for dealing with permissions omitted, even though they are needed by VerCors for ensuring that the program is race-free. This loop is meant as *ghost code*: code that is needed for verification only, and does not change any of the other variables. However, PVL does not distinguish between what is ghost code and what is not[4]. We indicate ghost code by using a blue color. For line 7, the ghost-code loop adds, in each of $count[n-1]$ iterations, a tiling that starts with a non-red tile, followed by a previous tiling. The other assignments to $count[n]$ are treated similarly.

The loop also demonstrates the use of a shorthand lemma uniqueness_implies_unequal(res, y, p), meaning:

$$
\begin{aligned}
&\mathrm{unique}(res) \land 0 \le j < res.length \Rightarrow \\
&\qquad (\forall y\,.\,0 \le y < j \Rightarrow \neg(p + res[y] = p + res[j]))
\end{aligned}
$$

We are able to add this property by manually making a ‘ghost method call’: a method with an empty function body, that is only there so VerCors can automatically establish its postcondition. We will see an example of establishing such a property later.

At the end of Algorithm 4, we can again establish that $res[j].length = count[j]$. This is a loop invariant on all non-ghost loops. The outer loop maintains that this property holds for $0 \le i < n$. For the inner loop, we maintain $last = count[n]$. For the ghost-code loops, the invariant looks slightly different as we take into account that *last* is in the process of being incremented.

To establish that elements in $res[j]$ are unique, we assert that the element we add to *last* is not contained in *last* already. However, this fact was not discovered automatically

[4] VerCors does, but only for Java and C.

by VerCors: we needed to use two main arguments. First, we reason about the prefix of all elements in $last$: In line 7, only sequences that start with a black node are added. In the next loop, sequences have a black node within the first k elements. At each of these points, including the last, the first occurrence of a black tile is later than that of the sequences present in $last$. We reason about this by defining has_false:

$$\text{has\_false}(s,k) = \exists y\,.\,0 \le y \le k \land \neg s[y]$$

Our second argument is used within the ghost-loops: By adding any prefix of red and black tiles to two tilings, we do not change whether or not those tilings are different.

To add an argument in a program, we again use ghost code. This time, the ghost code has the form of a method call. For instance, we needed to make the argument that if the l'th element is black in some tiling, then the first k elements of that tiling contain a black tile somewhere, provided that $l \le k$. VerCors can prove this automatically, but needs help to show the same statement holds when quantifying over a sequence. We turn this into a lemma by specifying the contract for our ghost method as given in Algorithm 5.

```
requires $0 \le l \le k$;
requires $\forall z\,.\,0 \le z < last.length \Rightarrow \neg last[z][l]$;
ensures $\forall z\,.\,0 \le z < last.length \Rightarrow$ has_false($last[z], k$);
void lemma_has_false($last, k, l$) {
  parallel_block ($z := 0$ to $last.length$)
  requires $\neg last[z][l]$;
  ensures has_false($last[z], k$); {
                    ▷ Proof determined automatically by VerCors
  }
}
```

Algorithm 5: Defining a lemma by writing a method.

As we can see in Algorithm 5 we conveniently use the syntax of a parallel block to work inside the quantifier. This is an interesting point where we can use a feature of the tool that is intended to reason about concurrent programs and apply it to prove quantified statements in sequential programs. We actually used this technique to prove other lemmas such as uniqueness_implies_unequal(res, y, p) as well.

5 Conclusion

This paper demonstrates that VerCors is well capable of reasoning about sequential programs, even though VerCors specialises in reasoning about parallelism and concurrency. The program logic of VerCors is based on concurrent separation logic and enforces ownership to be handled explicitly, giving some overhead. We plan to reduce this overhead in future work, by investigating ways to automatically infer the ownership annotations.

The slicing notations, $xs[..l]$ and $xs[l..]$, were needed in the first case study. It would have saved a lot of time if they were built-in. We currently have a bachelor student working on building in support for this and related constructions.

On the other hand, there were also examples on how reasoning with VerCors was especially convenient: the parallel-block construct meant that we could reason inside quantified statements easily. We believe that other tools might become easier to work with if they would support a similar operation in their (ghost) language. The convenience of parallel blocks inspired us to want to treat loops similarly in VerCors.

The case study has helped us identify which parts of VerCors are convenient strong points. This includes the use of parallel blocks, and of resources. It also identified concrete things to improve in VerCors: automatic ownership annotations for sequential code, supporting a more convenient way of dealing with loops, and adding slicing notations.

Acknowledgements

This work is supported by NWO grant 639.023.710 for the Mercedes project and by the NWO TOP grant 612.001.403 for the VerDi project.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. 2016. *Deductive Software Verification – The KeY Book*. Lecture Notes in Computer Science, Vol. 10001. Springer International Publishing.

[2] A. Amighi, S. Blom, and M. Huisman. 2014. Resource Protection Using Atomics - Patterns and Verification. In *APLAS*. 255–274.

[3] A. Amighi, S. Blom, and M. Huisman. 2016. VerCors: A Layered Approach to Practical Verification of Concurrent Software. In *PDP*. 495–503.

[4] A. Amighi, C. Haack, M. Huisman, and C. Hurlin. 2015. Permission-based separation logic for multithreaded Java programs. *LMCS* 11, 1 (2015).

[5] Afshin Amighi, Marieke Huisman, Stefan Blom, Saddek Bensalem, and Simon Bliudze. 2018. Verification of Shared-Reading Synchronisers. In *1st International Workshop on Methods and Tools for Rigorous System Design 2018*.

[6] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *iFM (LNCS)*, Vol. 10510. Springer, 102 – 110.

[7] Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. 2015. KIV: overview and VerifyThis competition. *STTT* 17, 6 (01 Nov 2015), 677–694. https://doi.org/10.1007/s10009-014-0308-3

[8] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *ESOP (LNCS)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 125–128.

[9] Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan (Eds.). 2015. *VerifyThis 2012 - A Program Verification Competition (STTT)*. Vol. 17. Issue 6.

[10] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. 2011. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM*.

[11] P. Müller, M. Schwerhoff, and A.J. Summers. 2016. Viper - A Verification Infrastructure for Permission-Based Reasoning. In *VMCAI*.

[12] W. Oortwijn, S. Blom, D. Gurov, M. Huisman, and M. Zaharieva-Stojanovski. 2017. An Abstraction Technique for Describing Concurrent Program Behaviour. In *VSTTE (LNCS)*, Vol. 10712. 191 – 209.

[13] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. 2015. AutoProof: Auto-active Functional Verification of Object-oriented Programs. In *TACAS (LNCS)*. Springer.