VerifyThis 2018

A Program Verification Competition

Marieke Huisman[1], **Rosemary Monahan**[2], **Peter Müller**[3], **Andrei Paskevich**[4], **Gidon Ernst**[5]

[1] University of Twente, The Netherlands, e-mail: m.huisman@utwente.nl
[2] Maynooth University, Ireland, e-mail: Rosemary.Monahan@nuim.ie
[3] ETH Zurich, Switzerland, e-mail: peter.mueller@inf.ethz.ch
[4] University Paris Saclay, France, e-mail: andrei.paskevich@lri.fr
[5] University of Melbourne, Australia, e-mail: gidon.ernst@unimelb.edu.au

Abstract. VerifyThis 2018 was a two-day program verification competition which took place on April 14 and 15, 2018 in Thessaloniki, Greece as part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2018). It was the sixth instalment in the VerifyThis competition series. This article provides an overview of the VerifyThis 2018 event, the challenges that were posed during the competition, and a high-level overview of the solutions to these challenges. It concludes with the results of the competition.

1 Introduction

VerifyThis 2018 took place on April 14 and 15, 2018 in Thessaloniki, Greece, as a two-day verification competition in the European Joint Conferences on Theory and Practice of Software (ETAPS 2018). It was the sixth edition in the VerifyThis series after the competitions held at FoVeOOS 2011, FM 2012, Dagstuhl Seminar 14171 (April 2014), ETAPS 2015–2017.

The aims of the competition were:

- to bring together those interested in formal verification, and to provide an engaging, hands-on, and fun opportunity for discussion;
- to evaluate the usability of logic-based program verification tools in a controlled experiment that could be easily repeated by others.

Typical challenges in the VerifyThis competitions are small but intricate algorithms given in pseudo-code with an informal specification in natural language. Participants have to formalise the requirements, implement a solution, and formally verify the implementation for adherence to the specification. There are no restrictions on the programming language and verification technology used. The time frame to solve each challenge is quite short (90 minutes) so that anyone can easily repeat the experiment. The verification challenges are available from the VerifyThis website http://www.pm.inf.ethz.ch/research/verifythis.html.

The correctness properties which the challenges present are typically expressive and focus on the input-output behaviour of programs. To tackle them to the full extent, some human guidance within a verification tool is usually required. At the same time, considering partial properties or simplified problems, if this suits the pragmatics of the tool, is encouraged. The competition welcomes participation of automatic tools as combining complementary strengths of different kinds of tools is a development that VerifyThis would like to advance.

Submissions are judged for correctness, completeness, and elegance. The focus includes the usability of the tools, their facilities for formalizing the properties and providing helpful output.

VerifyThis 2018 consisted of three increasingly difficult verification challenges, selected to showcase various aspects of software verification. Before the competition, an open call for challenge submissions was made. Overall, we received six proposals, of which we selected one for the competition, and some were excluded because published solutions are available. The challenges (presented later) provided reference implementations at different levels of detail.

Eleven teams participated (Table 1) in this edition of the competition. Teams of up to two people were allowed and physical presence on site was required. In addition, one non-competing solution was developed off site simultaneously. We particularly encouraged participation of:

- student teams (including PhD students);
- non-developer teams using a third-party tool;
- several teams using the same tool.

Teams using different tools for different challenges (or even for the same challenge) were also welcome. Student participants could apply for travel grants, sponsored by Amazon Web Services, to ease their participation.

The website of the 2018 instalment of VerifyThis can be found at http://www.pm.inf.ethz.ch/research/verifythis/Archive/2018.html. More background information on the competition format and the rationale behind it can be found in [9]. Reports from previous competitions of similar nature can be found in [15,3,7,11,12] and in the special issue of the International Journal on Software Tools for Technology Transfer (STTT) on the VerifyThis competition 2012 (see [10] for the introduction).

1.1 Invited Tutorial

We started the competition day with an interactive invited tutorial by Alexander Summers on the Viper tool [18]. Viper is a verification tool for permission-based reasoning about sequential and concurrent programs. Its primary purpose is to establish an infrastructure that can be relied on by front ends for higher-level programming languages to discharge their verification tasks (an example is VerCors [2], which was used in the competition by teams 7 and 8). A distinguishing feature of the tool is its support for the "magic wand" connective and quantified permissions.

Additionally, Viper provides a user-friendly surface language and integration into the Visual Studio Code editor. It can therefore be used stand-alone and was used in this mode in the competition (by team 10). During the tutorial, the participants had the opportunity to learn about Viper's methodology and to try out Viper on example verification problems.[1]

1.2 Post-mortem Sessions

Two concurrent post-mortem sessions were held the day after the competition, where participants explained their solutions and answered questions of the judges.

During one session, the judges asked the teams questions in order to better understand and appraise their solutions. These sessions provided an excellent opportunity to explore the strengths and weaknesses of the solutions presented by each team, while acquiring more detailed knowledge of the verification tools used. In parallel, all other participants presented their solutions, leading to lively discussion and exchange about tool developments. These presentations were also attended by some non-participants.

1.3 Judging Criteria

Limiting the duration of each challenge assists the judging and comparison of each solution. However, this task is still quite subjective and hence, difficult. Discussion of the solution with the judges typically results in a ranking of solutions for each challenge.

Criteria that were used for judging were:

- Correctness: is the formalisation of the properties adequate and fully supported by proofs? Were any bugs found in the code?
- Completeness: are all tasks solved, and are all required aspects covered? Are any assumptions made? Is termination verified?
- Readability: can the submission be understood easily, possibly even without a demo?
- Effort and time distribution: what is the relation between time expended on implementing the program vs. specifying properties vs. proving?
- Automation: how much manual interaction is required, and for what aspects? Does the solution make use of information from libraries?
- Novelty: does the submission apply novel techniques? What special features of the tool are used?

2 First Challenge: Gap Buffer

A gap buffer is a data structure for the implementation of text editors, which can efficiently move the cursor, as well as add and delete characters.[2] A gap buffer represents the editor's content as a character array a of size n, which has a gap of unused entries $a[l], \ldots, a[r-1]$, with respect to two indices $l \leq r$. The data it represents is composed as $a[0], \ldots, a[l-1], a[r], ..., a[n-1]$. The current cursor position is at the left index l, and if we type a character, it is written to $a[l]$ and l is increased. When the gap becomes empty, the array is enlarged and the data from r is shifted to the right.

The gap buffer provides four operations (shown in Fig. 1, given to the participants as part of the description). Procedures `left()` and `right()` move the cursor by one character; `insert()` places a character at the beginning of the gap $a[l]$; `delete()` removes the character at $a[l]$ from the range of text.

2.1 Verification Tasks

The intended behavior of the buffer should be specified in terms of a contiguous representation of the editor content. We suggested to use strings, functional arrays, sequences, or lists. The task was to verify that the gap buffer implementation satisfies this specification, and

[1] Available from http://www.pm.inf.ethz.ch/research/verifythis/Participation.html

[2] See e.g. http://scienceblogs.com/goodmath/2009/02/18/gap-buffers-or-why-bother-with-1

Table 1. Teams participating in VerifyThis 2018 (alphabetically by tool).

#	Team members	Tool		Team attributes
1	Stephen Siegel	CIVL	[19]	developer
2	Armaël Guéneau, Cyprien Mangin	Coq+CFML	[4]	student, developer
3	Lionel Blatter, Jean-Christophe Léchenet	Frama-C	[14]	student
4	Peter Lammich, Simon Wimmer	Isabelle Refinement Framework	[16]	developer
5	Stefan Bodenmüller, Jörg Pfähler	KIV	[5]	student, developer
6	Matthias Ulbrich	KeY, Dafny	[1,17]	developer (of KeY)
7	Wytse Oortwijn, Mohsen Safari	VerCors	[2]	student, developer
8	Sebastian Joosten, Marieke Huisman	VerCors	[2]	developer
9	Jafar Hamin	Verifast	[13]	student
10	Marco Eilers, Alexander Summers	Viper	[18]	developer
11	Raphael Rieu-Helft	Why3	[6]	student, developer
	Martin Clochard	Why3	[6]	student, developer, off site

```
procedure left()
    if l != 0 then
        l := l - 1
        r := r - 1
        a[r] := a[l]
    end-if
end-procedure

procedure right()
    // similar to left()
    // but pay attention to the
    // order of statements
end-procedure
```

```
procedure insert(x: char)
    if l == r then
        // see extended task
        grow()
    end-if
    a[l] := x
    l := l + 1

procedure delete()
    if l != 0 then
        l := l - 1
    end-if
end-procedure
```

```
procedure grow()
    var b := new char[a.length + K]
    // b[0..l] := a[0..l]
    for i = 0 to l - 1 do
        b[i] := a[i]
    end-for
    // b[r + K..] := a[r..]
    for i = r to a.length - 1 do
        b[i + K] := a[i]
    end-for
    r := r + K
    a := b
end-procedure
```

Fig. 1. Gap Buffer implementation in pseudo-code.

that every access to the array is within bounds. We encouraged to verify `insert()` with a precondition $l < r$ first and to ignore the call to `grow()`. An extended task was to lift this restriction and to specify and verify the procedure `grow()`. We explicitly asked the participants to do this modularly, not by referring to the implementation of `grow()` in the proof of `insert()`. We encouraged to make use of built-in functionality for copying array ranges (such as `System.arraycopy()` in Java) instead of the loops in `grow()`.

The participants had 60 minutes to implement, specify, and verify the challenge. It was intended to be straightforward to solve, not requiring advanced features of verification tools or familiarity with specific theories. The challenge exercises tool support for arrays, abstraction capabilities in the contracts/specifications of procedures, and modular verification.

2.2 Comments on Solutions

We received 10 complete solutions for part 1, compared to 1 partial and 4 complete solutions for part 2.

We observed that not all teams specified a *contiguous* representation, i.e., the fact that there is a gap was reflected in the specification which is arguably not an elegant approach when it would come to reasoning about a client (i.e., an editor) of this interface. Users of Dafny, Isabelle, KIV, KeY, and VerCors could benefit from their tool support for abstract sequences. Among those, we find Dafny's syntax for array slices and implicit promotion of arrays to mathematical sequences the most elegant (submitted by Matthias Ulbrich after the competition),

```
0 <= l <= r <= a.Length && s == a[..l] + a[r..]
```

where `s: seq<char>` is the abstract representation.

The solutions by the KIV and Isabelle teams expressed the specification independently from the code and used refinement to prove the correspondence.

The Frama-C team expressed the specification of `delete()` as two disjoint behaviors, nicely corresponding to the case distinction made in the implementation. Another remarkable aspect of their solution is the use of the `realloc()` library function in `grow()`, which is not only idiomatic in C but also entirely avoids reasoning about the first loop.

Among the few errors made in the preconditions and representation invariant, a common issue was to mistakenly specify $l < r$ in the invariant instead of $l \leq r$, which precludes moving the cursor to the rightmost position.

Separation Logic, as used by the Coq/CFML team, is very suitable to reason modularly about the array modifications. Their model is a representation based on Huet's "zipper" [8], which is well-suited to specify the contract of `grow()` but exposes the gap in the buffer. The authors of that solution considered another abstraction to lists but did not attempt it due to the limited time.

With the exception of CIVL, all tools support modular reasoning about procedures (pre-/postconditions). In the latter case, a manual encoding was done.

3 Second Challenge: Colored Tiles

This challenge is based on Project Euler problem #114.

Alice and Bob are decorating their kitchen, and they want to add a single row of fifty tiles on the edge of the kitchen counter. Tiles can be either red or black, and for aesthetic reasons, Alice and Bob insist that red tiles come by blocks of at least three consecutive tiles. Before starting, they wish to know how many ways there are of doing this. The algorithm is as follows:

```
var count[51]       // count[i] is the number of valid rows of size i
count[0] := 1       // []
count[1] := 1       // [B] - cannot have a single red tile
count[2] := 1       // [BB] - cannot have one or two red tiles
count[3] := 2       // [BBB] or [RRR]
for n = 4 to 50 do
  count[n] := count[n-1]  // either the row starts with a black tile
  for k = 3 to n-1 do     // or it starts with a block of k red tiles
      count[n] := count[n] + count[n-k-1]  // followed by a black one
  end-for
  count[n] := count[n]+1  // or the entire row is red
end-for
```

The participants had to verify that at the end, `count[50]` contains the correct number. Since the algorithm works by enumerating the valid colorings, a good solution is expected to provide a nice specification of a valid coloring and to prove that each coloring counted by the algorithm is valid, no coloring is counted twice, and no valid coloring is missed.

3.1 Comments on Solutions

While no on-site participant completed the challenge, a number of submissions provided a full specification and partially finished proof. The approach most used for specification consisted in describing explicitly the set of all valid colorings for a given number of tiles and linking the values computed by the algorithm to the cardinality of this set. In the Isabelle solution, quite elegantly, the set was defined directly by set comprehension. In other solutions, the set of colorings was computed in a recursive or iterative fashion, following the structure of the algorithm. In the Dafny submission, the sought value was computed by counting valid entries in the set of all possible colorings.

The suggested definitions of a valid coloring turned out to be quite diverse. Some submissions defined it in a pointwise fashion, by stating that each read tile must have two red adjacent tiles or two red tiles to its left or to its right. The others defined it recursively, by describing valid extensions of shorter valid colorings. Most submissions described a coloring as a sequence of colors (Booleans), while the Viper solution used the run-length representation.

Having access to rich mathematical library, as was the case for Isabelle, considerably simplified specification and proof. In particular, the second-order sum operator allowed for succinct and convincing specification.

Worthy of note, the CIVL submission produced an automated proof of unicity: for the three categories of valid colorings (starts with a black tile, starts with $n \geq 3$ red tiles followed by a black one, contains only red tiles), no coloring belong to two categories at once, and in the second category, no coloring has two different positions of the first black tile.

4 Third Challenge: Array-Based Queue Lock

Array-Based Queuing Lock (ABQL) is a variation of the Ticket Lock algorithm with a bounded number of concurrent threads and improved scalability due to better cache behaviour.

We assume that there are `N` threads and we allocate a shared Boolean array `pass[]` of length `N`. We also allocate a shared integer value `next`. In practice, `next` is an unsigned bounded integer that wraps to 0 on overflow, and we assume that the maximal value of `next` is of the form $k\texttt{N} - 1$. Finally, we assume at our disposal an atomic `fetch_and_add` instruction, such that `fetch_and_add(next,1)` increments the value of `next` by 1 and returns the original value of `next`.

The elements of `pass[]` are spinlocks, assigned individually to each thread in the waiting queue. Initially, each element of `pass[]` is set to `false`, except `pass[0]` which is set to `true`, allowing the first coming thread to acquire the lock. Variable `next` contains the number of the first available place in the waiting queue and is initialized to 0.

Here is an implementation of ABQL in pseudocode:

```
procedure abql_init()
  for i = 1 to N - 1 do
    pass[i] := false
  end-for
  pass[0] := true
  next := 0
end-procedure

function abql_acquire()
  var my_ticket := fetch_and_add(next,1) mod N
  while not pass[my_ticket] do
  end-while
  return my_ticket
end-function

procedure abql_release(my_ticket)
  pass[my_ticket] := false
  pass[(my_ticket + 1) mod N] := true
end-procedure
```

Each thread that acquires the lock must eventually release it by calling `abql_release(my_ticket)`, where `my_ticket` is the return value of the earlier call of `abql_acquire()`. We assume that no thread tries to reacquire the lock while already holding it, neither it attempts to release the lock which it does not possess.

Notice that the first assignment in `abql_release()` can be moved at the end of `abql_acquire()`.

The verification tasks were as follows:

1. Safety of ABQL under the given assumptions. Specifically, participants had to prove that no two threads can hold the lock at any given time.
2. Fairness, that is, the threads acquire the lock in order of request.
3. Liveness under a fair scheduler: each thread requesting the lock will eventually acquire it.

The participants had the liberty to adapt the implementation and specification of the concurrent setting as best suited for the verification tool of their choice. In particular, solutions with a fixed value of `N` were acceptable. It was expected, however, that the general idea of the algorithm and the non-deterministic behaviour of the scheduler were to be preserved.

4.1 Comments on Solutions

We received two solutions specifying and proving safety of ABQL in the general case (VerCors and Why3) and one solution proving safety and fairness for a bounded number of threads and a fixed maximal value of `next` (CIVL). Why3 submission also provided an unproved specification for fairness using a ghost queue.

Two main approaches were used for this challenge: direct implementation in tools with support for concurrency (Vercors, CIVL, Frama-C) or encoding the whole `N`-thread environment as a transition system. In the first case, the lock-free nature of the algorithm created an additional difficulty; both Vercors submissions added extra mutual exclusion primitives to the provided implementation. In the second case, the participants had to identify the relevant sequence points in the code; the number of states per thread ranged from 3 (KeY, Isabelle) to 4 (Viper) to 7 (Why3).

We would like to single out the specification framework provided by the VerCors tool, which allowed for natural and unencumbered thread-modular annotations.

5 Results and Closing Remarks

5.1 Awarded Prizes

We awarded five prizes. The announcements were made during the conference banquet and each winning team received a certificate and a prize in form of an Amazon voucher.

- Overall Best Team: Peter Lammich and Simon Wimmer (Isabelle, TU Munich)
- Student Gold Medal: Raphaël Rieu-Helft (Why3, Inria)
- Student Team Silver Medal: Wytse Oortwijn and Mohsen Safari (VerCors, Twente)
- Distinguished Tool Feature: Jörg Pfähler and Stefan Bodenmüller (Augsburg) for their program refinement method in KIV
- Best Challenge Submitted: Jean-Christophe Filliâtre, Colored Tiles

5.2 Lessons Learned

The two major challenges in organizing a competition like this are coming up with good riddles and judging the submissions.

For the first task, we would recommend to start searching as early as possible. Not many people will answer the call for problems and not all suggestions will be suitable for a 90 minute challenge. It may well happen that the organizers will have to invent a problem or two by themselves. Recruiting a knowledgeable and altruistic colleague or community member (who would have to forgo participation in the competition that year) for brainstorming sessions may also be of great help.

The organizers should remember that with more than a few actively developed tools for program verification on the scene, quite a lot of nice specification and verification problems (especially ones that can be handled in a limited time) have already been solved and published somewhere: in scientific papers, in online tutorials or simply somewhere on the tool's web page. Having rejected two promising ideas this year, we would advise our successors to spend some time looking for any previous work on the proposed challenges.

As for the second task, it is a good idea to prepare a long list of checkboxes for each problem beforehand and revise it constantly during the post-mortem interviews. For each aspect of each verification task, there will be some typical and some unique mistakes or omissions. Each of them may be only partially proved or not proved at all. Having a common evaluation grid, being ready to

extend it, and to update it retroactively for the previously discussed submissions will greatly simplify the final systematization of the results. This evaluation grid must also provide space for qualitative observations: highlights or curious aspects of a given solution or tool.

Acknowledgments

We thank Amazon Web Service for generous funding of travel grants and prizes. We thank Tej Chajed, Stefan Ciobaca, Ernie Cohen, Tim Denvir, Jean-Christophe Filliâtre, and Dennis Furey who submitted challenge proposals for the competition. The friendly easyconferences staff helped us with the organization of the competition dinner. Finally, we thank all participants for sitting through the—metaphorically and literally—hot sessions and for sharing and discussing their solutions.

References

1. W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. The KeY platform for verification and analysis of Java programs. In D. Giannakopoulou and D. Kroening, editors, *6th International Conference on Verified Software: Theories, Tools and Experiments (VSTTE 2014)*, volume 8471 of *LNCS*, pages 55–71. Springer, 2014.
2. S. Blom and M. Huisman. The VerCors tool for verification of concurrent programs. In C. B. Jones, P. Pihlajasaari, and J. Sun, editors, *19th International Symposium on Formal Methods (FM 2014)*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
3. T. Bormer, M. Brockschmidt, D. Distefano, G. Ernst, J.-C. Filliâtre, R. Grigore, M. Huisman, V. Klebanov, C. Marché, R. Monahan, W. Mostowski, N. Polikarpova, C. Scheben, G. Schellhorn, B. Tofan, J. Tschannen, and M. Ulbrich. The COST IC0701 verification competition 2011. In B. Beckert, F. Damiani, and D. Gurov, editors, *International Conference on Formal Verification of Object-Oriented Systems (FoVeOOS 2011)*, volume 7421 of *LNCS*, pages 3–21. Springer, 2011.
4. A. Charguéraud. Characteristic formulae for the verification of imperative programs. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.
5. G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV: overview and VerifyThis competition. *International Journal on Software Tools for Technology Transfer*, pages 1–18, 2014.
6. J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *22nd European Symposium on Programming (ESOP 2013)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
7. J.-C. Filliâtre, A. Paskevich, and A. Stump. The 2nd Verified Software Competition: Experience report. In V. Klebanov, A. Biere, B. Beckert, and G. Sutcliffe, editors, *1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE 2012)*, volume 873 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.
8. G. Huet. The zipper. *Journal of Functional Programming*, 7:549–554, 1997.
9. M. Huisman, V. Klebanov, and R. Monahan. On the organisation of program verification competitions. In V. Klebanov, B. Beckert, A. Biere, and G. Sutcliffe, editors, *1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE 2012)*, volume 873 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.
10. M. Huisman, V. Klebanov, and R. Monahan. VerifyThis 2012. *Int. J. Softw. Tools Technol. Transf.*, 17(6):647–657, Nov. 2015.
11. M. Huisman, V. Klebanov, R. Monahan, and M. Tautschnig. VerifyThis 2015. A program verification competition. *Int. J. Softw. Tools Technol. Transf.*, 2016.
12. M. Huisman, R. Monahan, P. Müller, and E. Poll. VerifyThis 2016. A program verification competition. *CTIT Technical Report Series*, 2016.
13. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - Third International Symposium, (NFM 2011)*.
14. F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.
15. V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. The 1st Verified Software Competition: Experience report. In M. Butler and W. Schulte, editors, *17th International Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 154–168. Springer, 2011.
16. P. Lammich. Automatic data refinement. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2013.
17. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2010)*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
18. P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

19. S. F. Siegel, M. B. Dwyer, G. Gopalakrishnan, Z. Luo, Z. Rakamaric, R. Thakur, M. Zheng, and T. K. Zirkel. CIVL: The concurrency intermediate verification language. Technical Report UD-CIS-2014/001, Department of Computer and Information Sciences, University of Delaware, 2014.