Comparing Verification Condition Generation with Symbolic Execution: an Experience Report

Ioannis T. Kassios, Peter Müller, and Malte Schwerhoff

ETH Zurich, Switzerland
{ioannis.kassios,peter.mueller,malte.schwerhoff}@inf.ethz.ch

Abstract. There are two dominant approaches for the construction of automatic program verifiers, *Verification Condition Generation* (VCG) and *Symbolic Execution* (SE). Both techniques have been used to develop powerful program verifiers. However, to the best of our knowledge, no systematic experiment has been conducted to compare them.
This paper reports on such an experiment. We have used the specification and programming language Chalice and compared the performance of its standard VCG verifier with a newer SE engine called Syxc, using the Chalice test suite as a benchmark. We have focused on comparing the efficiency of the two approaches, choosing suitable metrics for that purpose. Our metrics also suggest conclusions about the predictability of the performance. Our results show that verification via SE is roughly twice as fast as via VCG. It requires only a small fraction of the quantifier instantiations that are performed in the VCG-based verification.

1 Introduction

During the last years, automated program verification has progressed significantly. This progress is due to advances in each of the three layers that comprise an automatic verification tool: the specification methodology, the program verifier, and the theorem prover. The *program verifier* layer extracts proof obligations from the specified program and passes them to the theorem prover. There are two prevalent approaches to design program verifiers:

- *Verification Condition Generation* (VCG) uses programming calculi such as weakest preconditions [9] to compute a formula whose validity entails the correctness of the program. VCG is used in many state-of-the-art verifiers, in particular, those built on top of the intermediate languages Boogie [18] and Why [10]. Examples of VCG-based verifiers are Chalice [21], Dafny [17], ESC/Java [7], Frama-C [1], Spec# [3], and VCC [8].
- *Symbolic Execution* (SE) [14] executes a program using symbolic instead of concrete values and accumulates constraints on those values, which are used to generate proof obligations. SE has gained momentum especially in the separation logic world (for instance, jStar [24], Smallfoot [5], and VeriFast [12]), but has also been used in combination with other specification methodologies, for instance in KeY [4] and Syxc [28].

To the best of our knowledge, there has been no systematic comparison of the two approaches. Such a comparison would provide insights into the workings of current verification tools and useful guidance for the design of future ones. This guidance is especially important since the choice of specification methodology no longer dictates a verification approach. For instance, dynamic frames are supported in the VCG-based Dafny and the SE-based KeY [27]. Permission-based methodologies such as separation logic are supported in the VCG-based Chalice and the SE-based Syxc; VeriCool [29] supports both VCG and SE in a single tool.

This paper reports on an experiment we conducted to compare verification condition generation and symbolic execution. We used the Chalice language [21] and compared its standard VCG-based verifier [20] to a new SE engine [28]. Both verifiers use Z3 [22] as theorem prover. Chalice is an interesting target for such a comparison for several reasons: (1) It is small enough to make it feasible to develop two verifiers for the same language. (2) It provides a variety of features that make verification challenging, in particular, dynamic memory allocation and multi-threading. (3) Chalice's specification methodology is closely related to separation logic [25] such that our observations can be expected to apply also to verifiers for that methodology.

We ran both verifiers on the Chalice test suite. Our experiment focuses mainly on the efficiency of the verification by measuring run times. However, we also measured the number of quantifier instantiations performed by Z3, which gives an impression of how predictable the performance is, as well as the number of conflicts encountered during the verification, which characterizes the size of the search space. The results of these measurements are reported in Sec. 3. They show that the SE-engine Syxc is about twice as fast as the VCG-based Chalice verifier on most benchmarks. It requires much less quantifier instantiations and generally leads to fewer conflicts. Further comparisons, for instance, about the ease of understanding verification failures, are left as future work.

The initial comparisons of the performance of the two verifiers identified several outliers. Their investigation lead to interesting observations about the design of Syxc, which we discuss in Sec. 4.

2 Background

In this section, we provide the background on verification condition generation and symbolic execution that is used in the rest of the paper. We assume for both approaches that loops are annotated with loop invariants.

2.1 Verification Condition Generation

Verification condition generators use a programming calculus such as a weakest precondition calculus to compute a *verification condition* (VC), a pure logical formula whose validity entails the correctness of the program w.r.t. its specification. The VC is then fed to the theorem prover. Many modern program verifiers

generate verification conditions by first translating the program and its specification into an intermediate language such as Boogie [18] or Why [10] and then computing the VC on the intermediate representation.

Since a verification condition is a pure logic formula, it must include all knowledge that is needed to prove the correctness of the program. This knowledge includes many properties of the programming language semantics, for instance, about values and types. For imperative languages, it also contains a heap model, typically expressed as a global map from locations to values. All aspects of the verification, including reasoning about heap properties such as aliasing, are then left to the theorem prover.

A characteristic of the VCG approach is that it computes only one VC per module and, thus, invokes the theorem prover only once per module. On the upside, dealing with one large VC allows the theorem prover to apply optimizations. On the downside, the VC tends to be large, even for small programs, which increases the complexity for the theorem prover and complicates quantifier instantiation because the prover will in general find more matching patterns. Another drawback of large VCs is that they are undecipherable to the human, so VCG-based debugging needs extra tool assistance [23].

2.2 Symbolic Execution

Symbolic execution [14] verifies a program by simulating an execution with variables that do not take concrete values, but whole expressions (known as *symbolic values*). Knowledge about the symbolic values is accumulated in a logical formula called the *path condition*. When a branch in the control flow is encountered, the execution takes both branches and conjoins the appropriate formula to the path condition of each branch. The prover is called each time an assertion needs to be proved. The prover is then given the path condition as an assumption and a relatively simple proof obligation.

A known limitation of this approach is its exponential execution time in the number of branches. Since SE verifies each path through a program independently, this may cause inefficiencies when there are redundancies between the proof obligations for different paths. With VCG, one would hope that theorem provers detect and exploit such redundancies in the single large verification condition, avoiding or moderating the exponential blowup of SE.

Berdine et al. [5] have noticed that the heap topology described by a separation logic specification can be treated independently of the rest of the information that the formula contains. This observation makes it possible for an SE-verifier to deal with heap properties inside the verifier, *without sending them to the prover*. This approach is implemented in Smallfoot and has been adopted by VeriCool, VeriFast, as well as the verifier Syxc that we used for our experiment. Smallfoot-style symbolic execution produces proof obligations that are simpler than VCs, because more reasoning is done within the program verifier and less in the theorem prover.

A potential drawback of this division of labor is that the prover has only partial access to the information that is available to the verifier, for instance,

because heap properties are not encoded in the proof obligation sent to the prover. This may mean that the prover can prove less than in the VCG approach.

Understanding the implications of Smallfoot-style SE in terms of efficiency, predictability, and completeness was one of the motivations for our experiment. While building Syxc, we encountered several instances of the incompleteness problem, and a striking instance of the exponential branching problem, which we report on in Sec. 4.

3 Experiment and Results

In this section, we describe the set-up of our experiment and discuss our measurements. The tools and the benchmarks that were used in the experiments can be found on-line under `http://www.pm.inf.ethz.ch/people/schwerhoffm/vstte_sevcg_verifiers.zip`

Benchmarks. For the experiment we have utilized 29 test cases from the current Chalice test suite. These test cases exercise the main Chalice features such as fractional permissions, objects, threads, locks, and message passing. The development of the Chalice test suite was unrelated to the present experiment. We separated the test cases into correct and incorrect programs. Both tools verify all the correct programs and reject all incorrect programs.

The Chalice test suite includes additional examples that we could not benchmark, because Syxc does not yet support all features the Chalice language offers. For five examples we also had to (1) manually desugar certain expressions because they are not supported by Syxc, and to (2) remove a few methods that used unsupported features.

Verification in Chalice is modular, that is, each method is verified without considering its callers or the implementations of methods it calls. Therefore, the total size of the benchmark programs is less important than the size of individual methods. The largest two programs are two implementations of AVL trees, one iterative and one recursive. The recursive version is by far the largest benchmark in terms of lines of code and number of methods, whereas the iterative version includes the longest method.

Verification Tools. The VCG-based Chalice verifier used in our experiments is built from revision 08870c66a385. This is a slightly outdated version that does not use the new Chalice permission model [11], which is not yet implemented in Syxc. It uses the Boogie build from revision ba07abf9500e and Z3 3.1 x64. Boogie has been limited to a single error per Boogie procedure, since this comes closest to the behavior of Syxc, which is limited to one error per Chalice method. Both tool chains use the SMTLib2 front-end of Z3 and log their interactions with Z3. In this configuration Z3 is used via std-io, not via an API.

The Syxc tool chain comprised the same Chalice build in order to parse the input, and the same Z3 installation. It uses Z3 in a configuration that is nearly

identical to that of Boogie, with two minor differences: (1) Z3 responds to every command it receives, and not only to satisfiability checks, and (2) declarations are global instead of scope-local to optimize the verification of multiple paths.

Both the standard VCG-based Chalice verifier and the SE engine Syxc are written in Scala 2.8.

Two possibly relevant differences between the Z3 encoding generated by Boogie and by Syxc are (1) Syxc currently uses a "weak" Z3 type system, in the sense that there are not types (sorts) for snapshots, references and lists, all of which are encoded as integer-typed symbols. (2) Syxc uses the same sequence axioms as Boogie, except that they only range over integers, whereas Boogie's sequence axioms are polymorphic. Sequences are used by six out of the 22 examples in our test suite.

The Metrics. In the experiment, we measured for each benchmark program:

- Verification time (wall time) in seconds
- Number of quantifier instantiations that Z3 performed during the verification
- Number of conflicts encountered by Z3 during the verification time

Verification time is the total run time, including parsing and type checking, symbolic execution or verification condition generation, and time spent in the prover. Since Chalice and Syxc use the same parser and type checker, differences in the measurements can be attributed to the actual verification.

The number of quantifier instantiations is interesting, because it serves as an indication of the predictability of the verifier. Quantifier instantiation is guided by heuristics, which are often overly sensitive to small changes in the program or specification. The fewer quantifier instantiations a technique needs, the less it depends on heuristics.

A conflict is a failed attempt by the prover to assign a value to a variable. More instantiation attempts indicate that the prover had to explore a larger search space before finding a satisfying assignment.

Experiment. The experiments have been run on an Intel Core2 Quad CPU Q9550 2.83GHz, 4GB RAM, with Windows 7 Enterprise x64. For each benchmark program, the statistics have been collected by verifying the program with the VCG-based Chalice verifier and with Syxc, and then running Z3 on the interaction log file in order to get statistics from Z3. This has been done ten times for each file and each verifier. The numbers of these ten runs have been averaged. The run times of both verifiers have been measured with the same tool.

Results. The results of the experiment are summarized in Fig. 1. For each program, we show: lines of code, number of methods, the results of SE, the results of VCG, and finally, for each metric, the percentage of the result of SE over the result of VCG. Times are measured in seconds. Incorrect programs are in folder "fail". Correct programs are in folder "hold".

File			Syxc			Chalice			%		
Name	LOC	Methods	Time	QI	Conflicts	Time	QI	Conflicts	Time	QI	Conflicts
fail\cell	30	3	1.12	13	1	2.39	410	21	46.86	3.17	4.76
fail\LoopLockChange	44	3	1.12	134	9	2.47	1519	90	45.34	8.82	10.00
fail\ProdConsChannel	60	6	1.24	47	10	2.51	1009	72	49.40	4.66	13.89
fail\prog1	53	7	1.12	28	6	2.41	1045	97	46.47	2.68	6.19
fail\prog2	26	4	1.04	7	3	2.25	124	12	46.22	5.65	25.00
fail\prog3	20	3	0.98	8	1	2.25	128	7	43.56	6.25	14.29
fail\prog4	36	3	1.09	25	4	2.35	375	47	46.38	6.67	8.51
hold\AVLTree.iterative	212	3	2.24	231	177	9.45	146238	1288	23.70	0.16	13.72
hold\AVLTree.nokeys	572	19	34.21	2357	4304	154.03	2541875	19593	22.21	0.09	21.97
hold\cell	131	11	1.78	233	40	3.23	9599	526	55.11	2.43	7.60
hold\CopyLessMessagePassing	54	3	1.46	218	35	2.72	4769	210	53.68	4.57	16.67
hold\CopyLessMessagePassing-with-ack	62	3	1.71	751	75	2.76	4014	181	61.96	18.71	41.44
hold\CopyLessMessagePassing-with-ack2	66	3	1.66	792	78	2.92	10542	325	56.85	7.51	24.00
hold\dining-philosophers	74	3	1.62	1181	128	3.09	17144	416	52.43	6.89	30.82
hold\iterator	123	6	3.65	436	210	3.27	7577	316	111.62	5.75	66.39
hold\iterator2	111	6	1.69	174	41	3.17	12867	304	53.31	1.35	13.49
hold\LoopLockChange	69	5	1.26	220	42	2.60	2624	207	48.46	8.38	20.29
hold\OwickiGries	31	2	1.21	121	41	2.61	5696	206	46.36	2.12	19.90
hold\PetersonsAlgorithm	70	3	2.12	1196	278	7.36	160574	1500	28.80	0.74	18.53
hold\ProdConsChannel	78	6	1.43	282	117	2.78	4640	316	51.44	6.08	37.03
hold\producer-consumer	171	11	1.91	466	147	3.84	17667	598	49.74	2.64	24.58
hold\quantifiers	28	1	1.02	16	4	2.23	196	12	45.74	8.16	33.33
hold\RockBand	109	13	1.30	79	14	2.92	9129	341	44.52	0.87	4.11
hold\Sieve	56	4	1.42	308	93	2.74	7596	246	51.82	4.05	37.80
hold\swap	19	2	0.98	10	2	2.20	261	20	44.55	3.83	10.00
hold\prog1	33	4	1.07	40	27	2.32	1101	106	46.12	3.63	25.47
hold\prog2	52	7	1.11	22	17	2.31	258	28	48.05	8.53	60.71
hold\prog3	163	16	1.58	627	144	3.53	21632	640	44.76	2.90	22.50
hold\prog4	19	1	1.07	76	22	2.32	877	61	46.12	8.67	36.07

Fig. 1. Results of the Experiment

The standard deviation of the averages (not shown in the table) is negligible in all measurements. For run time, the worst deviation is 3% of the mean time, observed for the fail\cell example in the VCG tool. For the other metrics, VCG has no deviation (deterministic behavior), while SE has very minor deviations only in the large examples (these are attributed to the non-determinism of some standard Scala collection libraries). The insignificant deviation indicates that the results are stable and repeatable.

A consistent observation is that the execution time of SE is in the range of 40% to 50% of that of VCG. Thus, SE outperforms VCG, by what seems to be a constant factor.

An interesting observation is that the performance benefit of SE in the two largest programs (the AVL tree implementations) is bigger: there, the execution time of SE drops to around 20% of the time of VCG. This is promising, but far from conclusive for the scalability of SE. To draw reliable conclusions about scalability, we would need more and larger benchmarks. However, we chose not to develop extra benchmarks just for this experiment.

The number of quantifier instantiations in SE is consistently under 10% of that in VCG. Although it was expected that less quantifier instantiations would occur in SE because heap properties are handled outside the prover, such a difference was beyond our expectations. This is an indication that the performance of SE might be more predictable than the performance of VCG.

The percentage of conflicts had a greater variation than the percentage in the other metrics. Still conflicts in SE are consistently much fewer than in VCG, indicating that SE verification is "more focused" (explores a smaller proof space) than VCG.

We did not find significant differences in the results between correct and incorrect benchmark programs, although we intuitively believed that SE would deliver verification errors earlier. That said, given the small number of failing test cases that we have, this observation should be taken with a grain of salt.

Among the test cases, there is one outlier, the "iterator" program, in which VCG marginally outperforms SE. We are currently looking into this outstanding behavior, with the hope of discovering interesting design issues, as happened with outliers in previous experiments (see for example Sec. 4.2).

4 Additional Observations

In Sec. 2.2, we mentioned two challenges for symbolic execution: (1) the potential incompleteness caused by separating heap properties from path conditions and reasoning about the former in the verifier and about the latter in the prover (in Smallfoot-style SE), and (2) the exponential branching problem. The comparison to the VCG-based Chalice verifier exposed problems related to both challenges in an earlier version of Syxc: Syxc was not able to prove some examples that Chalice proved soundly, and for one example, the relative performance of Syxc was significantly worse than for the other examples. In this section, we discuss

the problems and describe the solution that is implemented in the current version of Syxc.

4.1 Heap Compression

Smallfoot-style SE is more susceptible to incompleteness than VCG, since the SE verifier does not give all the available heap-related information to the theorem prover. Dealing with such incompletenesses is not straightforward and involves several design trade-offs. In this subsection, we show one such incompleteness and its solution.

In Smallfoot-style SE, information about a heap location is expressed by a *heap chunk* $t.f \mapsto t'$, where t, t' are symbolic values and f is a field name. The heap chunk $t.f \mapsto t'$ means that access to the field f of the object t is granted and that t' is the value of that field. Syxc adds fractional permissions, so a Syxc heap chunk has the following form: $t.f \mapsto t'\#p$. The extra information p is the percentage of permission granted. Heap chunks are stored in the *symbolic heap*, which is used by the verifier to decide access permissions, ideally without consulting the prover. The relationship between symbolic values is encoded in the path condition, which is available to the prover when a proof obligation is checked.

Suppose that a program acquires the heap chunks $t_1.f \mapsto t_2\#30\%$ and $t_3.f \mapsto t_4\#30\%$. Suppose also that the path condition implies $t_1 = t_3$. This situation illustrates two sources of incompleteness caused by the division of labor between the verifier and the prover. First, the verifier fails to show an assertion that we have 60% permission to $t_1.f$, because without consulting the prover, it cannot derive the needed information $t_1 = t_3$ from the path condition. Second, the prover fails to show an assertion that $t_2 = t_4$, because this equality is a consequence of the contents of the symbolic heap, which is not available to the prover.

We fix this problem by using the path condition and the theorem prover to *compress* the symbolic heap, that is, to reflect aliasing information as follows: For each pair of heap chunks $t_1.f \mapsto t_2\#p_1$ and $t_3.f \mapsto t_4\#p_2$:

- Ask the prover if $t_1 = t_3$ follows from the path condition.
- If yes, then:
  - Replace the two heap chunks by one: $t_1.f \mapsto t_2\#p_3$
  - Add to the path condition the following conjuncts: $t_1 = t_3$, $t_2 = t_4$, $p_3 = p_1 + p_2$

Notice that the compression process introduces more equalities into the path condition. The stronger path condition may justify more compression. Therefore, we compress the symbolic heap iteratively, an operation that costs $O(n^3)$ queries to the prover (where n is the number of heap chunks).

A related problem appears when we have the following two heap chunks: $t_1.f \mapsto t_2\#60\%$ and $t_3.f \mapsto t_4\#60\%$. Since the accumulative permission to a field cannot exceed 100%, we can conclude from this symbolic heap that $t_1 \neq t_3$.

To allow the prover to exploit this knowledge, we add it to the path condition during the heap compression.

The need for heap compression is due to the fact that Smallfoot-style SE does not include heap (and permission) properties in the proof obligations sent to the prover, while they are included in the verification condition produced by a VCG-based verifier. Our experiments show that SE outperforms VCG even though we have to perform the explicit heap compression. Nevertheless, we believe that it has a lot of potential for optimization, for instance, it could be performed on demand only when a proof fails.

Our experience with VCG and SE suggests that the higher efficiency of SE does not necessarily come at the price of less precision. However, tool developers have to be careful to enable the necessary information flow between the symbolic heap and the path condition.

4.2 Branching

Symbolic execution verifies each path through a module separately. In addition to the branches introduced by control flow, Syxc also introduces branches when it needs to represent properties expressed as implications. For example, the Chalice specification $b \Rightarrow acc(e.f)$ means that if b is true then the current thread has write access to field $e.f$, and no access permission otherwise. Since such a conditional permission cannot be represented with the heap chunks described above, Syxc branches. One branch adds b to the path condition and an appropriate heap chunk for $e.f$ to the symbolic heap, whereas the other branch adds $\neg p$ to the path condition and leaves the symbolic heap unchanged.

Experiments with an earlier version of Syxc identified "Peterson's Algorithm" from the Chalice test suite as a significant outlier: while SE performed better than VCG in the other examples, it performed significantly worse here. Our analysis revealed that the bad performance was caused by excessive branching on implications. However, all the implications in this example were *pure*, that is, did not contain any access permission predicates. Therefore, the whole implication could be added to the path condition, and no branching is necessary. Implementing this change made SE outperform VCG again.

Our experience suggests that the theorem prover has better ways to handle implications than just case splits; the larger proof obligation with the implications allows the prover to avoid redundant proofs. This seems to confirm the intuition that VCG outperforms SE in case of heavy branching.

Note that our solution is possible only for pure implications, because a pure implication does not influence the symbolic heap. To alleviate the problem of branching on impure implications, we consider introducing "conditional" heap chunks. Initial experiments in this direction show some promise.

5 Related Work

As mentioned earlier, there is a multitude of tools that support SE in the style of Smallfoot [5]. These include jStar [24], VeriCool [29], VeriFast [12], and Syxc [28].

Of those, Smallfoot, jStar, and VeriFast use a fragment of separation logic [26] as their specification language, while the target programming language differs. VeriCool and Syxc use implicit dynamic frames [29]. VeriFast and Syxc also support fractional permissions [6]. Implicit dynamic frames have similar expressive power to the fragment of separation logic used by SE tools [25].

The KeY system [4] uses a different form of symbolic execution that does not separate heap properties from "pure" properties, in the style of Smallfoot. The specification language of KeY is JML [16] with dynamic frames [13]. The KeY approach admits some degree of interaction with the prover. Our comparison focuses on Smallfoot-style SE. It is not clear to what degree our observations apply to KeY-style SE.

Tools that are based on VCG include Chalice [20], Dafny [17], ESC/Java [7], Frama-C [1], Regional Logic [2], Spec# [3], VCC [8], and VeriCool [29]. Most of them employ the intermediate languages Boogie [18] or Why [10]. Chalice and VeriCool support implicit dynamic frames (in Chalice with fractional permissions), Dafny supports dynamic frames, and Spec# and VCC support ownership [19].

VeriCool [29] supports both SE and VCG. The authors report on experiments using the visitor pattern and an artificial example that show an overwhelming advantage for SE [30]. Our experiments do not confirm this result, even though our tools are similar to VeriCool (based on implicit dynamic frames, Smallfoot-style SE, and Z3). We decided to perform our comparison using Chalice and Syxc rather than VeriCool because the specification language supported by VeriCool's VCG engine is significantly different from the one supported by its SE engine, which means that we could not have tried the exact same code in both engines. For example, the VeriCool specification language for SE does not support quantification.

VSTTE 2010 hosted a software verification competition [15]. The evaluation focused on the quality of specifications and the strength of the verification, but does not permit a comparison of the efficiency of verification approaches like the one we present here.

6 Conclusion

In this paper, we reported on an experiment that compared two dominant approaches for the construction of automated program verifiers, Verification Condition Generation and Symbolic Execution. Our experiment shows that SE is generally more efficient, leads to a smaller search space for the prover, and requires fewer quantifier instantiations. Even though the differences in the run times are noticeable, they are just a constant factor. Future experiments will have to show whether more substantial differences exist for larger modules. Another topic for future work is to compare other important criteria, such as completeness, the ease of understanding and fixing verification errors, as well as the performance on more substantial failing cases. The latter is also a strong indicator of the

responsiveness of the tools in an interactive setting, in which the program and its specifications are developed over several iterations of running the tool.

We believe that experimental evaluation is an important aspect of advancing the field of software verification. The VSTTE verification competition provides interesting insights in to the strength of various verifiers (tools and their users). Our experiment complements the competition by providing a comparison of two verification approaches in terms of their efficiency. We hope that it will encourage others to perform additional studies that will help the community to better understand the impact of design decisions on the performance of verification tools.

Acknowledgments. We are grateful to Jan Smans for his patient explanations of VeriCool and to Leonardo de Moura and Christoph Wintersteiger for helpful explanations of Z3's statistics and heuristics. We also like to thank Michał Moskal for help on Boogie and Z3, Uri Juhasz for providing us with the AVL-tree example, and Alex Summers for many fruitful discussions. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. J. B. Almeida, M. J. Frade, J. S. Pinto, and S. Melo de Sousa. Verifying C programs. In *Rigorous Software Development*, Undergraduate Topics in Computer Science, pages 241–256. Springer-Verlag, 2011.
2. A. Banerjee, D. Naumann, and S. Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP'08*, volume 5142 of *Lecture Notes In Computer Science*, pages 387–411. Springer-Verlag, 2008.
3. M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.
4. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes In Computer Science*. Springer-Verlag, 2007.
5. J. Berdine, Cristiano Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO'05*, volume 4111 of *Lecture Notes In Computer Science*, pages 115–137. Springer-Verlag, 2006.
6. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *SA'03*, volume 2694 of *Lecture Notes In Computer Science*, pages 55–72. Springer-Verlag, 2003.
7. P. Chalin, J. R. Kiniry, G. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO'05*, volume 4111 of *Lecture Notes In Computer Science*, pages 342–363. Springer-Verlag, 2005.
8. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskał, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes In Computer Science*, 2009.
9. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall, 1976.

10. J. C. Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, 2003.
11. S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional permissions without the fractions. In *Formal Techniques for Java-like Programs (FTfJP)*, 2011.
12. B. Jacobs, J. Smans, and F. Piessens. VeriFast: Imperative programs as proofs. In *VS-Tools workshop at VSTTE'10*, 2010.
13. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM'06*, volume 4085 of *Lecture Notes In Computer Science*, pages 268–283. Springer-Verlag, 2006.
14. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
15. V. Klebanov, P. Müller, N. Shankar, G. 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, *FM'11*, volume 6664 of *Lecture Notes In Computer Science*. Springer-Verlag, 2011.
16. G. Leavens, A. L. Baker, and C. Ruby. JML: a notation for detailed design. In I. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
17. K. R. M. Leino. Specification and verification of object-oriented software. In *Marktoberdorf International Summer School 2008, Lecture Notes*, 2008.
18. K. R. M. Leino. This is Boogie 2. Working Draft - available at `http://-research.microsoft.com/en-us/um/people/leino/papers.html`, 2008.
19. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *ECOOP'04*, volume 3086 of *Lecture Notes In Computer Science*, pages 491–516. Springer-Verlag, 2004.
20. K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In G. Castagna, editor, *ESOP'09*, volume 5502 of *Lecture Notes In Computer Science*, pages 378–393. Springer-Verlag, 2009.
21. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes In Computer Science*, pages 195–222. Springer-Verlag, 2009.
22. L. Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS'08*, volume 4963 of *Lecture Notes In Computer Science*, pages 337–340. Springer-Verlag, 2008.
23. P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In M. Butler and W. Schulte, editors, *FM'11*, volume 6664 of *Lecture Notes In Computer Science*. Springer-Verlag, 2011.
24. M. Parkinson and D. Distefano. jStar: Towards practical verification for Java. In G. E. Harris, editor, *OOPSLA'08*, pages 213–226. ACM, 2008.
25. M. Parkinson and A. Summers. The relationship between separation logic and implicit dynamic frames. In Gilles Barthe, editor, *ESOP'11*, volume 6602 of *Lecture Notes In Computer Science*. Springer-Verlag, 2011.
26. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE Computer Society, 2002.
27. P. Schmitt, M. Ulbrich, and B. Weiß. Dynamic frames in Java dynamic logic. In *FoVeOOS'10*, volume 6528 of *Lecture Notes In Computer Science*, pages 138–152. Springer-Verlag, 2011.

28. M. Schwerhoff. Symbolic execution for Chalice. Master's thesis, ETH Zurich, 2011.
29. J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP'09*, Genoa, pages 148–172. Springer-Verlag, 2009.
30. J. Smans, B. Jacobs, and F. Piessens. Symbolic execution for implicit dynamic frames. Available from `http://people.cs.kuleuven.be/~jan.smans/vericool3/`, 2009.