Testing a Saturation-Based Theorem Prover: Experiences and Challenges*

Giles Reger[1], Martin Suda[2], and Andrei Voronkov[1,3,4]

[1] University of Manchester, Manchester, UK
[2] TU Wien, Vienna, Austria
[3] Chalmers University of Technology, Gothenburg, Sweden
[4] EasyChair

Abstract. This paper attempts to address the question of how best to assure the correctness of saturation-based automated theorem provers using our experience with developing the theorem prover Vampire. We describe the techniques we currently employ to ensure that Vampire is correct and use this to motivate future challenges that need to be addressed to make this process more straightforward and to achieve better correctness guarantees.

1 Introduction

This paper considers the problem of checking that a saturation-based automated theorem prover is *correct*. We consider this question within the context of the Vampire theorem prover [14], but many of our discussions generalise to similar theorem provers such as E [22], SPASS [26], and iProver [13]. We discuss what we mean precisely by correctness, describe how we detect bugs and, as our main contribution, outline the challenges that need to be addressed.

Automated theorem provers (ATPs) are often used as *black boxes* in other techniques (e.g. program verification) and those techniques rely on the results of the theorem prover for the correctness of their own results. Another area that makes use of ATPs is the application of so-called *hammers* [15, 12] in interactive theorem proving. These combinations usually provide functionality to reconstruct the proofs of the ATP using their own trusted kernels, although also offer users the option to skip such steps.

It is clear that correctness is important here, so how are we doing? Most theorem provers seem to be generally correct. However, cases of unsoundness are not uncommon. In SMT-COMP 2016 there were 603 conflicts (solvers returning different results) on 73 benchmarks caused by three solvers giving incorrect results for various reasons.[5] In the CASC competition [25], there is a period of testing where soundness is checked and resolved, and there have been a number of solvers later disqualified from the competition due to unsoundness. In our experience, adding a new feature to a theorem prover is a highly complex task and it is easy to introduce unsoundness, or general incorrectness, especially in areas of the code that are encountered during proof search infrequently.

* This work was supported by EPSRC Grant EP/K032674/1, ERC Starting Grant 2014 SYMCAR 639270, Austrian research projects FWF S11403-N23 and S11409-N23, and the Wallenberg Academy Fellowship 2014 – TheProSE.

[5] See `http://smtcomp.sourceforge.net/2016/`.

This paper begins by describing what we mean by correctness with respect to saturation-based theorem provers (Section 2) and the approach we take to finding and fixing bugs (Section 3). This provides sufficient context to present a set of challenges that need to be addressed to produce a better solution to this problem (Section 4). Addressing these challenges is part of our current ongoing research. An extended version of this paper containing examples of bugs found in Vampire is available online [20].

2 What Does Correctness Mean for Us?

Broadly there are two ways in which a theorem prover such as Vampire can be incorrect: either it *returns the wrong result*, or it *violates a contract of proper behaviour*.

2.1 Incorrect result

To understand what a correct and incorrect result mean to Vampire, we need to introduce some of the theoretical foundations of the underlying technique. We note that the approach used by Vampire is the same as that taken by other first-order theorem provers, so these discussions, and the challenges outlined later, generalise beyond Vampire.

Vampire accepts problems (formulas) in the form

$$(Premise_1 \wedge \ldots \wedge Premise_n) \rightarrow Conjecture \tag{1}$$

and can give one of three answers:

- *Theorem*, if (1) is true in all models,
- *Non-Theorem*, if there are models in which (1) is false, and
- *Unknown*, if Vampire cannot deduce one of the previous answers.

Providing one of the first two results when that result does not hold is clearly incorrect. Providing *Unknown* as the result is clearly incorrect in the sense that there is a known answer, but, due to the undecidability of first-order-logic and the general hardness of the problem, it is often unavoidable. However, as discussed below, we should understand the different ways in which *Unknown* as a result can be produced. Note that *Unknown* will be returned if Vampire exceeds either the time or memory allotted to it.

More specifically, Vampire is a refutational theorem prover; it establishes the *validity* of problems in the form (1) by detecting *unsatisfiability* of its negation:

$$Premise_1 \wedge \ldots \wedge Premise_n \wedge \neg Conjecture. \tag{2}$$

This works by translating (2) into a set of *clauses* $\mathcal{S}$ and adding consequences of $\mathcal{S}$ until the contradiction $false$ is derived or all possible consequences have been added. This process is called *saturation* and may not terminate in general for a satisfiable set $\mathcal{S}$.

If Vampire derives a contradiction then it has shown that the problem (1) is *valid*, i.e. a theorem. Deriving a contradiction when the problem in (1) is not valid is *unsound* and an *incorrect result*.

If Vampire fails to derive a contradiction and *saturates* the set $\mathcal{S}$ in finitely many steps then there is a result [2] telling us that under certain conditions we can conclude that $false$ cannot be a consequence of $\mathcal{S}$ and therefore problem (1) is a non-theorem.

These conditions capture the *completeness* of the underlying inference system and generally require that all possible *non-redundant* inferences have been performed.

However, there are many things that Vampire does to heuristically improve proof search that break the completeness conditions. For example, (i) certain well-performing selection functions [10] might prevent inferences that need to be performed for completeness conditions to hold; and (ii) some preprocessing steps and proof search strategies explicitly remove clauses from the search space in an attempt to mitigate search space explosion [11, 21]. If the completeness conditions do not hold then upon saturation the result is *Unknown*. Sometimes it is easy to detect when these conditions hold, sometimes it is non-trivial, and sometimes they are erroneously broken. In this last case (when we think the conditions hold but they do not) this will lead to incorrectly reporting non-theorem i.e. this *completeness issue* is another kind of *incorrect result*.

To ensure the requirement that all possible non-redundant inferences will in the end be performed, we impose certain *fairness* criteria on the saturation process. More concretely, we require that no such inference is postponed indefinitely. Notice that this is by nature a tricky condition to deal with as it cannot be seen to have been violated after finitely many steps while the prover is running. And since, due to the semi-decidability of first-order logic, there is no upper bound on the length of the computation required to derive $false$, a non-fair implementation might in certain cases never be able to return *Theorem*, even if it is the correct answer and instead keep computing indefinitely. Thus, this *fairness issue* does not lead to an incorrect result per se, but rather just negatively influences performance. As such it may be extremely hard to detect and deal with.

2.2 Violating the contract of proper behaviour

There are two kinds of contracts of proper behaviour that Vampire can violate: those introduced implicitly by the underlying system, and those introduced explicitly by us in the form of assertions. We discuss both kinds of bug below:

- *Program crash.* A program crash is where Vampire terminates unexpectedly, usually due to an unhandled exception, floating point error (SIGFPE), or segmentation fault (SIGSEG). Unhandled exceptions are bugs as we should handle them. In general, Vampire handles all known classes of exceptions at the top level, but we have recently had issues with integrated tools (MiniSAT and Z3) producing exceptions that we did not handle. Floating point errors and segmentation faults are typical software bugs that should be detected and removed.
- *Assertion violation.* Vampire is developed defensively with frequent use of *assertions*. For example, these are inserted wherever a function makes some assumptions about its input or the results of a nested function call, and wherever we believe a certain line to be unreachable. Vampire consists of roughly 194,000 lines of C++ code with roughly 2,500 assertions, meaning that there is roughly one assertion per 77 lines. The majority of potential errors are detected early as assertion violations.

3 Finding Bugs

In this section we briefly describe how we detect and investigate bugs in Vampire where these two steps can be equally difficult. The search space for Vampire is vast, and finding the combination of inputs that triggers a bug is very difficult. Some bugs are incred-

ibly subtle, particularly soundness bugs or those involving memory errors, and tracking them down can involve hunting through thousands of lines of output.

3.1 The Input Search Space

The two inputs to Vampire are the input problem and a strategy capturing proof search parameters. The space of possible input problems is infinite. However, we do not currently explore this space systematically. Instead we sample from sets of representative benchmarks, e.g. TPTP [24] ($\sim$20k problems) and SMT-LIB [4] ($\sim$46k relevant problems). Vampire currently uses roughly 75 proof search parameters with more than half of these having more than two possible values and some taking arbitrary numeric values (although in testing we fix these to a predefined sensible set). Therefore, the search space is significantly larger than 2^{75}, i.e. too large to explore systematically.

3.2 The Debug Process

Bug reports come from two sources:

- Users of the Vampire system may report bugs to us. Currently this is an informal process carried out by personal email. Sometimes these bugs are actually feature requests, and other times they can be due to a misuse of Vampire.
- More commonly, they come from randomly sampling the parameter space and sets of available problems (ensuring reasonable diversity in terms of features and status, e.g. theorems and non-theorems). We use a cluster[6] that enables us to carry out around a million checks a day (using varying short time limits).

Once an error is detected, we must diagnose and fix the fault. Below we describe some of our methods for doing this.

- *Tracing.* Vampire has its own library for tracing function calls. A macro is manually inserted at the start of each significant function. This macro enables the tracing library to maintain the current call stack, which is then printed on an assertion violation or during signal handling along with the number of such call points passed so far. This second piece of information can be used to explicitly log function calls for some range of call points, e.g. those just before the erroneous point. This feature is invaluable in quickly locating the cause of an assertion violation.
- *Memory Checking.* Vampire implements its own memory management library, allowing fine-grained control of memory allocation and deallocation and enforcement of soft memory limits. In debug mode, Vampire keeps track of each allocated piece of memory and checks that the corresponding deallocation is as expected. Vampire also reports memory leaks i.e. unallocated memory at the end of the proof search.
- *Segmentation Faults and Silent Memory Issues.* The most difficult bug to debug is a rogue pointer or piece of uninitialised memory. We find that a first step of applying Valgrind[7] will often detect the more straightforward issues. However, such bugs are often only noticed via incorrect results and fixed by much manual effort.
- *Proof Checking.* To detect unsoundness we employ proof checking, which we discuss further below. We do not currently have a corresponding method for checking that a saturated set complies with necessary completeness conditions.

[6] Consisting of 46 nodes with quad-core Intel Xeon CPUs and 12GB RAM.

[7] `http://valgrind.org`

3.3 Proof Checking

The easiest way to confirm a result indicating that the input formula is a theorem is to check that the associated proof only performs sound inference steps. This process is called proof checking and here we briefly describe the capabilities and limitations of the proof checking technique as currently realised in Vampire.

We introduce the idea of proof checking using an example (see [17] for more information about proofs in Vampire). Given the clauses

$$p(a) \qquad \neg p(x) \vee b = x \qquad \neg p(b)$$

Vampire will produce the following proof in TPTP format[8]

```
1. p(a) [input]
2. ~p(X0) | b = X0 [input]
3. ~p(b) [input]
4. a = b [resolution 2,1]
5. ~p(a) [backward demodulation 4,3]
7. $false [subsumption resolution 5,1]
```

A proof is a directed acyclic graph printed in a linear form where nodes that have no incoming edges are either input formulas or axioms introduced by Vampire, and the single node with no outgoing edges contains the contradiction. In the above proof each derived clause is labelled with the name of the inference and the lines of the premises.

To check a proof we just need to establish that for each inference its conclusion logically follows from its premises. By running `vampire -p proofcheck` we can produce a series of TPTP problems capturing each proof step. For example the following problem captures step 5 in the above proof.

```
fof(pr4,axiom, a = b ).
fof(pr3,axiom, ~p(b) ).
fof(r5,conjecture, ~p(a) ).
```

We can pass these directly to an independent theorem prover[9] and if a step cannot be independently verified then it should be investigated.

4 Challenges

We now present a discussion of what we have identified as the main challenges left to be solved, or at least addressed, given in order of importance, as we perceive it.

4.1 Full and Automated Proof Checking

As described in Section 3.3, there is already reasonable support for independently checking the correctness of proofs. However, this situation could still be improved.

Missing Features. There are parts of proofs that cannot currently be proof checked, the two main parts are:

[8] All TPTP-compliant provers must produce proofs in this format (see `http://www.cs.miami.edu/~tptp/TPTP/QuickGuide/Derivations.html`). We note that the TPTP project also provides separate proof checking tools [23].

[9] Currently we use E [22], iProver [13], and CVC4 [3] as independent provers but could use any accepting TPTP formatted problems.

– *Symbol Introducing Preprocessing.* Certain inference steps of the clausification phase, e.g. Skolemization and formula naming [19], introduce new symbols and as such do not preserve logical equivalence. This means the conclusion of the inference does not logically follow from its premises. What these steps preserve is global satisfiability of the clause set they modify. One necessary condition for correctness is that the introduced symbols be *fresh*, i.e. not appearing elsewhere in the input. This requires a non-trivial extension to the described approach.
– *SAT and SMT solving.* Vampire makes use of SAT and SMT solvers in various ways (see [18]). This means that we have some inferences in Vampire that are of the form $P_1 \wedge \ldots \wedge P_n \rightarrow C$ *by SAT/SMT*, or even the argument that some abstraction or grounding of the premises leads to C by SAT or SMT solving. To handle such proof steps we need to collect together the premises (potentially apply the necessary abstraction or grounding) and run a SAT or SMT solver as appropriate.

Extra information may need to be added to proofs to support these checks.

Automating Proof Checking. Having tools able to check the correctness of proofs is irrelevant if those tools are not used. Ideally, theorem provers should provide the functionality to check the proofs that they produce automatically. As the problems produced during proof checking are often easy to solve, one could imagine a situation where, in a certain mode, a theorem prover applied proof checking to its proof output.

Independence. It might not be possible to find an independent solver able to handle the problems produced by proof checking. A solver might not be able to check an individual step, because it is too hard, or not be able to handle the language features the problem contains. A weaker independence could be achieved by making use of a previous version of the original theorem prover that we are more confident in.

4.2 Analysability of Unsound Proofs

Checking whether a proof is correct or not is essential. However, knowing that a proof is incorrect is not, in itself, very useful. Another missing piece to this puzzle are tools that can analyse proofs and extract, summarise or explain the *reason* the proof is incorrect. The proof checking process will reveal the proof step that fails to hold, but the problem of detecting the underlying reason for that proof step to have occurred is non-trivial.

One step in this direction is the application of *delta-debugging* [27] to reduce the input to a simpler form to aid debugging efforts. This approach has been explored for SAT/QBF solvers [5, 1] applied to both the input problem and the parameter space.

4.3 Handling Non-theorem Results

So far we have ignored the incorrect result of reporting a problem to be satisfiable when it is not. It is not clear how to practically check whether a saturated set is indeed saturated as the notion of saturation is dependent on the used calculus and its instantiation with parameters such as the term ordering and literal selection methods.

Non-redundant inferences. A necessary condition for completeness it that proof search never deletes anything that is not redundant. Checking this is significantly more complex than proof checking. In proof checking we must check that each inference of the proof is sound i.e. that we were allowed to perform those inferences to derive a

contradiction. If we have a saturated set then we should check that every inference that we chose not to perform was redundant; this is what we often have to do manually, with some intuition about what such inferences might be. The number of such inferences is typically a few orders of magnitude larger than the length of a typical proof.

Monitoring fairness. To avoid missing a saturated set we need to satisfy the fairness criteria discussed in Section 2.1. However, this is not *monitorable* in a formal sense [8, 9] as it cannot be satisfied or violated based on a finite number of observations. However, if we were to introduce a *stronger* property of *bounded fairness* [7], e.g. a clause of age A will be processed within kA iterations for some constant k, then this property becomes monitorable (this is now a *response* property).

4.4 Achieving Better Coverage with Random Testing

As previously discussed, due to the enormous variability in proof search parameters and possible problem inputs, the best approach to detecting errors and incorrect results is through random search. However, the current approaches to random search are not optimal. Here we briefly outline areas of improvement.

Code Coverage. Our current approach makes no attempts to ensure that testing covers all lines in the code. Even though this is a very weak notion of coverage, it could be used to detect areas of code that should be tested, or removed if never used.

Coverage of the Parameter Space. Whilst random sampling of the parameter space can be effective at discovering bugs, it is not clear that all areas of the parameter space are of equal interest. Clearly, combinations of features that have not been tested together should have priority, and features added more recently should be tested more thoroughly. In this vein we could borrow from T-wise test case generation strategies for Software Product Lines [16] which aims to test all T-combinations of features.

Coverage of the Problem Space. This is an area where relatively little has been done (in the first-order setting). We currently use libraries of existing problems as possible inputs to the testing process. However, if we do not have a problem that exercises a certain feature sufficiently, we are unlikely to detect bugs related to that feature. For example, the TPTP language contains features that are very rarely used within the TPTP library. This issue is not confined to language features. Proof search is dependent on particular dimensions of the input problem (e.g. size, signature) that are difficult to quantify. If the input problems do not cover these dimensions sufficiently then certain parts of Vampire will not be tested effectively. A useful area of research would be the automatic generation of problems, or *fuzzing* of existing problems, to cover such dimensions. In this direction we could borrow from successful results in SAT/QBF solving [5, 6].

5 Conclusion

This paper describes our experience testing the Vampire theorem prover and what we see as the challenges to overcome to help us improve this effort. The ideas we discuss generalise to other theorem provers and some efforts, such as proof checking techniques and better problem coverage, would be widely beneficial. Addressing the challenges set out in this paper is part of our current research and we plan to provide a proof checking tool that can fully and automatically check proofs produced by Vampire.

References

1. C. Artho, A. Biere, and M. Seidl. Model-based testing for verification backends. In *Proc. 7th Intl. Conf. on Tests & Proofs (TAP'13)*, LNCS, p. 17 pages. Springer, 2013.
2. L. Bachmair and H. Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*, vol. I, chapter 2, pp. 19–99. Elsevier Science, 2001.
3. C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, number 6806 in Lecture Notes in Computer Science, pp. 171–177. Springer-Verlag, 2011.
4. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
5. R. Brummayer, F. Lonsing, and A. Biere. *Automated Testing and Debugging of SAT and QBF Solvers*, pp. 44–57. Springer Berlin Heidelberg, 2010.
6. N. Creignou, U. Egly, and M. Seidl. *A Framework for the Specification of Random SAT and QSAT Formulas*, pp. 163–168. Springer Berlin Heidelberg, 2012.
7. N. Dershowitz, D. N. Jayasimha, and S. Park. *Bounded Fairness*, pp. 304–317. Springer Berlin Heidelberg, 2003.
8. V. Diekert and M. Leucker. Topology, monitorable properties and runtime verification. *Theoretical Computer Science*, 537:29 – 41, 2014. Theoretical Aspects of Computing (ICTAC 2011).
9. Y. Falcone, J.-C. Fernandez, and L. Mounier. *Runtime Verification of Safety-Progress Properties*, pp. 40–59. Springer Berlin Heidelberg, 2009.
10. K. Hoder, G. Reger, M. Suda, and A. Voronkov. Selecting the selection. In *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 – July 2, 2016, Proceedings*, pp. 313–329. Springer International Publishing, 2016.
11. K. Hoder and A. Voronkov. Sine qua non for large theory reasoning. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, vol. 6803 of *Lecture Notes in Computer Science*, pp. 299–314. Springer, 2011.
12. C. Kaliszyk and J. Urban. Hol(y)hammer: Online ATP service for HOL light. *Mathematics in Computer Science*, 9(1):5–22, 2015.
13. K. Korovin. iprover - an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, vol. 5195 of *Lecture Notes in Computer Science*, pp. 292–298. Springer, 2008.
14. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, vol. 8044 of *Lecture Notes in Computer Science*, pp. 1–35, 2013.
15. L. C. Paulson and J. C. Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In *IWIL 2010. The 8th International Workshop on the Implementation of Logics*, vol. 2 of *EPiC Series in Computing*, pp. 1–11. EasyChair, 2012.
16. G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. l. Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pp. 459–468. IEEE Computer Society, 2010.
17. G. Reger. Better proof output for Vampire. In *Vampire 2016. Proceedings of the 3rd Vampire Workshop*, vol. 44 of *EPiC Series in Computing*, pp. 46–60. EasyChair, 2017.
18. G. Reger and M. Suda. The uses of sat solvers in vampire. In *Proceedings of the 1st and 2nd Vampire Workshops*, vol. 38 of *EPiC Series in Computing*, pp. 63–69. EasyChair, 2016.

19. G. Reger, M. Suda, and A. Voronkov. New techniques in clausal form generation. In *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, vol. 41 of *EPiC Series in Computing*, pp. 11–23. EasyChair, 2016.
20. G. Reger, M. Suda, and A. Voronkov. Testing a Saturation-Based Theorem Prover: Experiences and Challenges (Extended Version). *ArXiv e-prints*, 2017.
21. A. Riazanov and A. Voronkov. Limited resource strategy in resolution theorem proving. *J. Symb. Comput.*, 36(1-2):101–115, 2003.
22. S. Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.
23. G. Sutcliffe. Semantic Derivation Verification: Techniques and Implementation. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.
24. G. Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009.
25. G. Sutcliffe. The CADE ATP system competition - CASC. *AI Magazine*, 37(2):99–101, 2016.
26. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, vol. 5663 of *Lecture Notes in Computer Science*, pp. 140–145. Springer, 2009.
27. A. Zeller. *Yesterday, my Program Worked. Today, it Does Not. Why?*, pp. 253–267. Springer Berlin Heidelberg, 1999.