A Multithreaded Vampire with Shared Persistent Grounding

Michael Rawson and Giles Reger
University of Manchester

***Abstract*—Automated theorem provers (ATPs) typically run in a single thread. Hardware parallelism is then exploited through *portfolios*, in which distinct and disjoint strategies are launched as fully-independent processes and do not cooperate. Whilst there has been some historic exploration of cooperation, the technical challenge has prevented this from being fully explored in modern ATPs. The following describes the non-trivial engineering effort required to make the Vampire theorem prover multithreaded, such that multiple proof attempts coexist in the same memory space. This lays the foundations for a new generation of proof search techniques able to cooperate with other proof attempts running in parallel. As an initial demonstration, we implement a shared *persistent grounding* daemon that receives all clauses generated by all proof attempts and checks whether a heuristically-grounded version is unsatisfiable. The resulting multi-threaded system achieves limited contention compared to the previous process-based implementation, and persistent grounding improves performance in certain cases.**

I. Introduction

Whilst parallel computational resources have become abundant and used with effect in many areas of computer science, they are yet to make a significant impact on automated theorem proving. We have seen substantial developments in SAT solving [1], [2], [3] and progress within SMT [4], [5], [6] but, to date, parallel automated theorem proving is typically historic with no modern implementation [7], [8], [9], or parallel at the level of portfolios without shared memory. The popularity of parallel portfolios is likely due to their ease of implementation and practical impact: it is common folklore that a good way to combat explosive proof search is a set of complementary search strategies. This success goes some way to explaining why research in other directions has been slow.

In this paper we discuss our initial work on a new shared-memory architecture for the Vampire automated first-order theorem prover [10]. Vampire is a saturation-based theorem prover that implements the superposition calculus [11] as its main mode, but also contains routines for instance-based reasoning [12] and finite model building [13]. It has won first place in the main track of the CASC competition for over 20 years [14] and implements advanced reasoning techniques for theory reasoning [15], [16], [17], inductive reasoning [18] and higher-order reasoning [19]. It consists of over 200k lines of C++ with contributions from over 15 developers and a permissive licence [20]. As such, it is a mature and highly-complex piece of software.

Since 2010, Vampire has supported some form of multi-process parallelism where a portfolio of predetermined (and automatically generated) *strategies* (sets of proof search heuristics) could be implemented by forked processes. This achieves good results, but limits options for cooperation between proof attempts due to reliance on inter-process communication. In 2015, we proposed a concurrent architecture [21] that interleaved proof attempts within a single process whilst sharing (some) memory to explore a novel method for cooperation. Our conclusion at the time was that we needed true shared-memory parallelism to make progress.

We experienced two main difficulties with such an approach in Vampire. The first is that it is difficult to implement correctly: this is a well-known feature of parallel programming, and we discuss our approach and experience below. The second is *contention*, which for our purposes is negative performance impact caused by multiple threads using the same resource simultaneously, typically by having to wait for a lock held by another thread. Avoiding contention requires careful design of shared-memory schemes within an ATP.

A reasonable line of questioning raised in review asks whether it would be easier to start from scratch. It would probably be technically easier to do so: however, ATP systems at Vampire's level of maturity take significant time to develop, even with the benefit of hindsight, so instead we offer pragmatic suggestions to convert existing systems.

The two main contributions of this paper are (1) A detailed discussion of the technical challenges and experience involved in transitioning a complex, mature theorem prover from a process-based model to a thread-based, shared-memory architecture (Section II), and (2) A new *persistent grounding* technique designed to take advantage of the shared memory concurrency provided by the architecture (Section III).

II. Challenges and Experience

This section reflects on the engineering challenges we faced when converting Vampire into a multi-threaded solver, and the approach we took to overcome them. We include this discussion to provide guidance for others attempting to complete a similarly-challenging task. Currently, the implementation is available in a branch of the Vampire repository[1].

A. Design

The architecture is based on the previous process-based architecture, which has not previously been described elsewhere. As illustrated in Fig. 1, the input problem is first parsed into a set of initial formulas over a signature (that is, the symbols

[1]https://github.com/vprover/Vampire/tree/caps

https://doi.org/

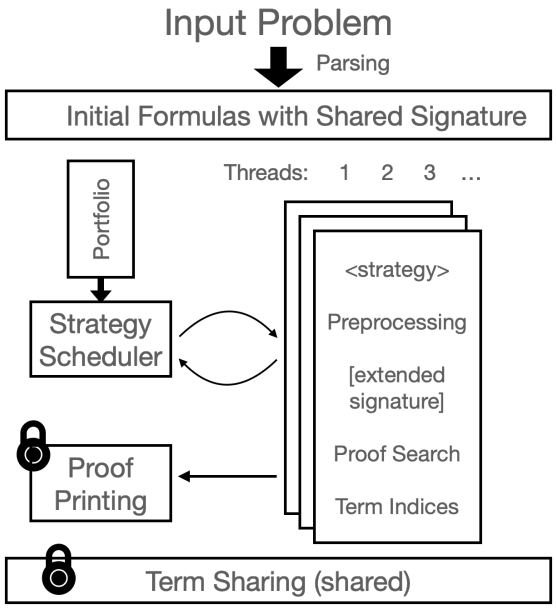

Fig. 1. Schematic of Architecture.

appearing in the problem) shared between all proof attempts. A strategy scheduler uses a portfolio of strategies to generate a set of k threads. The parent scheduler supervises the child threads, reporting success if any child succeeds and spawning new threads to keep available CPU cores busy. Each thread preprocesses the problem, potentially extending the signature by e.g. introducing names for subformulas, and then performs proof search. This typically involves the use of complex data structures (*term indices*) for storing and searching for relevant clauses. VAMPIRE's complex custom memory allocator is disabled for this work, incurring a small performance hit.

Two complex parts of the architecture are currently protected by a coarse-grained lock. Only one proof attempt should print a proof, so this process is gated such that subsequent successful attempts block forever. A more difficult issue is *term sharing*. Part of the standard VAMPIRE is a hash-consing structure used to implement perfect term sharing, i.e. avoid duplication of terms. This is very convenient as it allows rapid identification of terms by pointer comparison, a property which is assumed throughout VAMPIRE. In our multithreaded architecture we share this structure and protect it by a lock. Term sharing must be able to distinguish between terms built solely from the shared signature and terms involving thread-specific symbols: that is, terms that could appear in any attempt versus terms that only have meaning in a single attempt.

B. Approach

Converting a large, complex and performance-sensitive system such as VAMPIRE to work in thread-parallel is not especially easy. The approach outlined previously [21] in which proof attempts *interleave* in a single thread of execution, rather than exist concurrently, at first seemed like a good intermediate step before starting work on a fully thread-parallel, shared-memory system. However, we found that bugs introduced by interleaved proof attempts were very difficult to track down, not least because very often they had no observable effect.

Instead we take a more chaotic approach, leaning heavily on tooling for developing multi-threaded applications, particularly tools for detecting *data races*. Data races, for our purposes, are execution scenarios in which two threads access shared memory without synchronisation, and at least one access is a write. Detection of races is extremely useful in our case as it provides a good proxy for identifying when one proof attempt influences the execution of another. Nearly all thread-related bugs — of which there were many — could then be squashed by examining the context in which races occur and introducing synchronisation or data reorganisation where appropriate.

Tools for detecting dubious constructs and execution states in low-level programming have improved significantly. We were particularly impressed by the LLVM-based [22] linter *clang-tidy* [23], which helped to identify and remove existing discouraged constructs in VAMPIRE's codebase, and the *ThreadSanitiser* [24] compiler instrumentation for the detection of data races. Armed with these tools, we simply introduced threads into VAMPIRE and waited for the tool reports. Races happened frequently in VAMPIRE at first, where code written under the implicit assumption of single-threaded execution breaks down, triggering a ThreadSanitiser report.

In general, data races tend to lead to crashes rather than unsound behaviour but to avoid the latter we rely on (i) existing mechanisms for automated testing utilising large sets of labelled benchmarks [25], and (ii) VAMPIRE's support for proof checking which allows us to independently verify the correctness of proof search [26].

C. Thread-Local Storage, Atomics and Locking

The most common source of the races was the re-use of heap-allocated temporaries such as stacks or maps, often used in iterative translations of recursive algorithms present throughout the system. Reusing these values once allocated can improve performance in the single-threaded case by avoiding repeated (de)allocations. The majority of such cases can be resolved by the use of thread-local storage as a compromise, incurring one allocation per thread. The 2011 C++ standard [27] provides a `thread_local` keyword and associated machinery.

Another problem area is integer counters, often used for computing statistics and satisfying freshness constraints such as "select a fresh symbol for the Skolem function". Usually the only operation required is "read-and-increment", but this must sometimes be reflected across threads to maintain soundness of e.g. Section III. This operation can be safely achieved atomically: C++'s `<atomic>` proved useful here.

Only surprisingly rarely was a full lock required to synchronise compound operations. This relatively-coarse technique was only required for widely-used modules with non-trivial internal invariants such as the implementation of term sharing. Due to the small number of locks, deadlock was mostly avoided.

D. Data Organisation and Partitioning

Significant headaches can be avoided by carefully choosing which data are shared between proof attempts. A clever implementation could aggressively share all common data using very fine-grained synchronisation. For example, VAMPIRE maintains various term indices to quickly retrieve various syntactic data that satisfy some condition, like "retrieve all the literals that unify with L". In principle it would be possible to share at least some of these and save some memory, but in practice this is enormously difficult to implement correctly and efficiently. However, we remain interested in parallel term indices and may investigate these independently in future.

Currently, each proof attempt maintains its own clause space, computed properties and statistics, indices, introduced definitions, and ground reasoning systems such as those used in global subsumption [28] or AVATAR [29]. They do however share synchronised access to creating fresh symbols (although not all symbols are used in all proof attempts), term sharing, and persistent grounding (Section III). We feel this is a good initial trade-off.

E. Timing and Internal Control

One crucial difference between the multi-processing and multi-threading approaches to portfolio modes is that processes can be signalled to stop execution in a timely manner, whereas most threading abstractions do not have this ability. Threaded proof attempts must therefore frequently check for exit conditions, e.g. another proof attempt succeeded/time is up. Making these checks can be tricky: too frequently and there will be some performance impact; too infrequently and user experience or portfolio performance will begin to degrade. VAMPIRE executes a series of loops in its internal search routines: each iteration of these loops can take drastically different lengths of time depending upon the input problem.

F. Synchronisation and Performance

All the synchronisation measures introduced do incur some performance impact. Atomic operations are not quite *free*, but are very close in practice. Thread-local storage requires some checks for lazy initialisation, which can occur frequently if the compiler is unable to elide them, and is therefore not as cheap as we would like. VAMPIRE uses a global "environment" structure which was made thread-local: C++ semantics mean that this is considerably more efficient if an extra level of indirection is added such that the environment is accessed via thread-local *pointer*. Locks are currently a major bottleneck: while contention was expected to be high, another problem is that the locked sections are typically relatively short and inexpensive compared to the locking overhead. We will investigate finer-grained locking and alternative locking strategies in future.

G. Experimental Evaluation

To validate the resulting system we carry out two experiments using the 500 first-order problems from the 2020 first-order theorem division of CASC. All experiments in this paper are run for 60 seconds per problem on a Ubuntu desktop machine with an 8-core CPU[2] and 16GB RAM.

TABLE I
EVALUATING SCALABILITY OF THREADED ARCHITECTURE.

Threads	# solved	Avg time (s)	Total/Avg (s) on ∩	Speedup
1	399	7.05	2187 / 6.21	-
2	413	4.80	987 / 2.80	2.22
4	412	3.49	520 / 1.48	4.21
6	413	2.79	539 / 1.53	4.06
8	402	3.27	533 / 1.51	4.10
10	404	3.26	534 / 1.52	4.10

Firstly, we compare the new thread-based architecture with the previous process-based implementation. The thread-based architecture solves 413 problems (10 uniquely) and the process-based architecture solves 424 problems (21 uniquely). The slight degradation in performance is unsurprising given the additional contention in the thread-based approach. The symmetric difference reflects the sensitivity of VAMPIRE to variations in timing and memory usage. On average, the new thread-based architecture took 1.25x longer to solve problems. However, this is heavily influenced by short-running problems. Excluding problems solved in under 1s, the slowdown is 1.02x.

Secondly, we examine the scalability of the thread-based solution using the same set of problems whilst varying the number of threads. The results are in Table I. The number of problems solved peaks between 2 and 6 threads. We achieve approximately-linear speedup with 2 and then 4 threads, but then plateau (based on the total time taken to solve the 352 problems solved by all attempts). The average solution time overall was the lowest for 6 threads — the lower average solution times for the intersection of solved problems suggests that these were the easier problems.

In summary, performance degrades slightly when replacing processes by threads (most likely due to contention) but the overhead is acceptable ($\sim 2\%$ on longer running problems).

III. PERSISTENT GROUNDING

As a first step to explore the benefits of the new architecture, we introduce a lightweight form of clause sharing. All clauses produced by all proof attempts are grounded, shared, and passed to a SAT solver to detect a form of *global* inconsistency, i.e. an inconsistency in the ground abstraction of the full search space explored by all proof attempts, past and present.

The idea of grounding the search space of a first-order prover in an attempt to detect inconsistency is not novel [30], [31] and some methods, such as instance generation [12] perform grounding as part of proof search already. What is new in our approach is the *persistence* of the grounding: grounded clauses escape from and outlive their thread, allowing clauses from different proof attempts to interact.

A. Extension to Architecture

We introduce a queue (synchronised by single lock) that proof attempts add produced (and grounded) clauses to and a

[2]Intel® Core™ i7-6700 CPU @ 3.40GHz

thread that loops, adding the grounded clauses to the MiniSAT solver [32] — yielding if the queue is empty — and checking for unsatisfiability. If the grounding is inconsistent the thread will report this immediately, interrupting other threads. Currently, full proof printing is not implemented and only the unsatisfiable core of grounded first-order clauses is identified. It is work-in-progress to rebuild the derivations that produced these clauses as a separate post-processing step.

We maintain a mapping from (grounded) first-order literals to SAT literals such that a fresh first-order literal leads to a fresh SAT literal, with the mapping stored for later. This mapping relies on the shared term indexing structure to efficiently identify atoms that are shared between proof attempts, ensuring they are represented using the same SAT variables.

B. Grounding Choices

There are numerous ways in which we could choose to ground first-order clauses. We implement three alternatives:

- **fresh**: all variables are replaced by a single fresh constant.
- **common**: all variables are replaced by the most common constant from the input problem.
- **input**: the clause is grounded repeatedly for every constant in the input problem.

Where the input problem is multi-sorted the above constants are selected per-sort. We compute constant frequency on the problem before preprocessing i.e. before subformulas are copied or reduced.

C. Experimental Analysis

We use the same 500 problems and experimental setup as above to analyse the impact of this new addition. Our first experiment is to isolate the impact of persistent grounding from threading by running with a single thread. In this setting, we solve 399 problems without persistent grounding and 398 with (using the **fresh** grounding) but with a symmetric difference of 11 problems — persistent grounding allows us to solve 5 problems we did not solve without it. Some problems were also solved significantly faster: for 8 problems the speedup was $> 2\times$, with one problem (`SWB105+1`) solved $15\times$ faster (from 25s to 1.6s).

Next, we compare the different grounding mechanisms (using 6 threads). The results are given in Table II (top 4 rows). The first observation is that we solve 8 problems that we did not solve without persistent grounding, and each grounding mechanism solves some problems uniquely.

However, the average time to solve each problem increases. The **fresh** grounding mechanism fares the worst with the **common** grounding mechanism producing proofs more than a second before other mechanisms 5 times. Within this there are some notable interesting cases. For example, `GRP667+1` was solved using **input** in 15s whilst others failed to solve it using persistent grounding and it was eventually solved in the normal way after 50s. Similarly, `ITP006+4` was solved using **common** in 9s rather than the 25s elsewhere.

TABLE II
PERSISTENT GROUNDING EVALUATION.

	# solved (uniq)	Best by >1s	Avg. time (s)
none	413 (6)	-	2.79
fresh	410 (1)	0	3.09
common	411 (2)	5	2.95
input	411 (2)	3	3.11
fresh	410 (2)	4	2.94
active-only	412 (3)	0	3.01
no-splitting	393 (5)	16	3.19
combination of PG	421 (12)	-	2.84 (best)

We explore two further variants (rows 5–7 of Table II): in *active-only* we restrict persistent grounding only to so-called *active* clauses [10] and in *no-splitting* we turned clause splitting off for all strategies. Clause splitting introduces additional (per proof attempt) propositional literals into split clauses, potentially reducing the amount of sharing between proof attempts. Active-only solves more problems and (not shown in the table) enjoys a slight reduction in solving times in cases where persistent grounding is not used to solve the problem. Turning clause splitting off solves fewer problems but is nicely complementary (solving 5 problems uniquely).

In summary, the persistent grounding method can drastically speed up proof search when it finds a proof but it generally adds a noticeable overhead. Overall, we solve 12 problems with variants of persistent grounding that we were unable to solve without it. The main observation is that it is possible to prove more by sharing information between proof attempts than simply running the union of proof attempts separately but more work is required to make this approach efficient.

IV. REFLECTION AND FUTURE WORK

We describe our initial efforts transforming VAMPIRE to a multi-threaded architecture and show how this new shared memory architecture can easily support methods for clause sharing. Whilst the concepts involved are straightforward, the engineering effort required to transform a mature codebase from a process-based single memory architecture to a thread-based shared-memory one is large. We have described our experience for others. Our general findings are:

1) It is more important to find a clean way to separate data and isolate points of sharing than it is to introduce "clever" fine-grained synchronisation. This ensures that debugging is manageable. We achieved a lot with `thread_local` and `atomic`.
2) In a large codebase like VAMPIRE there are tens or hundreds of little bottlenecks rather than few big ones and they interact in complex ways. Simply optimising one bottleneck rarely gives overall gains, improvements must be more architecturally-focussed.
3) Portfolio strategies are typically very short (often <1s) so "small" performance hits can have a large impact. Work is required to make portfolios robust to this setting.

The new shared persistent grounding method gave lacklustre results but only represents a first step in a number of oppor-

tunities presented by the new architecture. Directions we plan to pursue in the future include:

- Extending the shared signature. Currently, if two proof attempts introduce a definition for the same subformula this will be added to each local extended signature and the overlap will not be shared. A shared definition manager could increase the size of the shared signature, increasing the opportunity for cooperation.
- As originally proposed in [21], sharing the SAT solver used for clause splitting in AVATAR. Within a single proof attempt, this SAT solver is used to enumerate sub-problems. When shared, it can share information about previously proved sub-problems between proof attempts (similar to sharing learned clauses in parallel SAT [2]).
- Sharing simplification mechanisms (and associated data structures e.g. term indices). VAMPIRE contains a number of mechanisms for removing redundant parts of the search space. By sharing these mechanisms we can import information from other proof attempts that makes the current problem easier.
- Other clause sharing mechanisms. Whilst sharing many clauses risks proof attempts converging (undoing the complementary power), we can explore methods that aim to identify useful clauses to share. A fashionable approach would be to employ machine learning techniques to learn which clauses are good to share. Alternatively, we could take inspiration from SAT's *lazy clause exchange* [33] where clauses are only shared if useful locally. Finally, it is likely that not all clauses will be equally useful to all other proof attempts, which suggests a setting where clauses are *pulled* rather than *pushed* based on a local assessment of usefulness.

ACKNOWLEDGEMENT

This work was funded by EPSRC project EP/V000209/1: *CAPS: Collaborative Architectures for Proof Search*.

REFERENCES

[1] Y. Hamadi, S. Jabbour, and L. Sais, "ManySAT: a parallel SAT solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, no. 4, pp. 245–262, 2010.

[2] T. Balyo and C. Sinz, "Parallel satisfiability," in *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 3–29.

[3] M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding CDCL SAT solvers by lookaheads," in *Haifa Verification Conference*. Springer, 2011, pp. 50–65.

[4] C. M. Wintersteiger, Y. Hamadi, and L. Moura, "A concurrent portfolio approach to SMT solving," in *CAV '09*, 2009, pp. 715–720. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02658-4_60

[5] A. E. Hyvärinen, M. Marescotti, L. Alt, and N. Sharygina, "OpenSMT2: An SMT solver for multi-core and cloud computing," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2016, pp. 547–553.

[6] A. E. Hyvärinen and C. M. Wintersteiger, "Parallel satisfiability modulo theories," in *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 141–178.

[7] J. Denzinger and I. Dahn, "Cooperating theorem provers," in *Automated Deduction—A Basis for Applications*. Springer, 1998, pp. 383–416.

[8] M. P. Bonacina, "Parallel theorem proving," *Handbook of Parallel Constraint Reasoning*, pp. 179–235, 2018.

[9] J. Schumann and R. Letz, "PARTHEO: a high performance parallel theorem prover," in *CADE*, ser. LNAI, vol. 449, Kaiserslautern, 1990, pp. 40–56.

[10] L. Kovács and A. Voronkov, "First-order theorem proving and Vampire," in *CAV 2013*, ser. LNCS, vol. 8044, 2013, pp. 1–35.

[11] R. Nieuwenhuis and A. Rubio, "Paramodulation-based theorem proving," in *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds., 2001, vol. I, ch. 7, pp. 371–443.

[12] K. Korovin, "Inst-Gen – a modular approach to instantiation-based automated reasoning," in *Programming Logics*, 2013, pp. 239–270. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37651-1_10

[13] G. Reger and M. Suda, "The uses of SAT solvers in Vampire," in *Proceedings of the 1st and 2nd Vampire Workshops*, ser. EPiC Series in Computing, L. Kovács and A. Voronkov, Eds., vol. 38. EasyChair, 2015, pp. 63–69.

[14] [Online]. Available: http://www.tptp.org/CASC/

[15] G. Reger, N. Bjørner, M. Suda, and A. Voronkov, "AVATAR modulo theories," in *GCAI 2016*, ser. EPiC, vol. 41. EasyChair, 2016, pp. 39–52.

[16] G. Reger, M. Suda, and A. Voronkov, "Unification with abstraction and theory instantiation in saturation-based reasoning," in *TACAS 2018*, ser. LNCS, 2018.

[17] G. Reger, J. Schoisswohl, and A. Voronkov, "Making theory reasoning simpler," in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021*, ser. Lecture Notes in Computer Science, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 164–180. [Online]. Available: https://doi.org/10.1007/978-3-030-72013-1_9

[18] G. Reger and A. Voronkov, "Induction in saturation-based proof search," in *International Conference on Automated Deduction*. Springer, 2019, pp. 477–494.

[19] A. Bhayat and G. Reger, "A combinator-based superposition calculus for higher-order logic," in *Automated Reasoning - 10th International Joint Conference, IJCAR*, ser. Lecture Notes in Computer Science, N. Peltier and V. Sofronie-Stokkermans, Eds., vol. 12166. Springer, 2020, pp. 278–296. [Online]. Available: https://doi.org/10.1007/978-3-030-51074-9_16

[20] [Online]. Available: https://vprover.github.io/

[21] G. Reger, D. Tishkovsky, and A. Voronkov, "Cooperating proof attempts," in *International Conference on Automated Deduction*. Springer, 2015, pp. 339–355.

[22] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[23] B. Babati, G. Horváth, V. Májer, and N. Pataki, "Static analysis toolset with Clang," in *Proceedings of the 10th International Conference on Applied Informatics (30 January–1 February, 2017, Eger, Hungary)*, 2017, pp. 23–29.

[24] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic race detection with LLVM compiler," in *International Conference on Runtime Verification*. Springer, 2011, pp. 110–114.

[25] G. Reger, M. Suda, and A. Voronkov, "Testing a saturation-based theorem prover," in *TAP 2017*. Springer, 2017, pp. 152–161.

[26] G. Reger, "Better proof output for vampire," in *Vampire 2016*, ser. EPiC, vol. 44. EasyChair, 2017, pp. 46–60.

[27] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*, 3rd ed. ISO, Sep. 2011.

[28] G. Reger and M. Suda, "Global subsumption revisited (briefly)," in *Vampire 2016.*, ser. EPiC, 2017.

[29] A. Voronkov, "AVATAR: The architecture for first-order theorem provers," in *CAV*. Springer, 2014.

[30] S. Schulz, "A comparison of different techniques for grounding near-propositional CNF formulae." in *FLAIRS Conference*, 2002, pp. 72–76.

[31] ——, "Light-weight integration of SAT solving into first-order reasoners — first experiments," in *Vampire Workshop*, 2017, pp. 9–19.

[32] N. Eén and N. Sörensson, "An extensible SAT solver," in *International conference on theory and applications of satisfiability testing*. Springer, 2003, pp. 502–518.

[33] G. Audemard and L. Simon, "Lazy clause exchange policy for parallel SAT solvers," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2014, pp. 197–205.