`versat`: A Verified Modern SAT Solver

Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy

Computer Science
The University of Iowa

Abstract. This paper presents `versat`, a formally verified SAT solver incorporating the essential features of modern SAT solvers, including clause learning, watched literals, optimized conflict analysis, non-chronological backtracking, and decision heuristics. Unlike previous related work on SAT-solver verification, our implementation uses efficient low-level data structures like mutable C arrays for clauses and other solver state, and machine integers for literals. The implementation and proofs are written in GURU, a verified-programming language. We compare `versat` to a state-of-the-art SAT solver that produces certified "unsat" answers. We also show through an empirical evaluation that `versat` can solve SAT problems on the modern scale.

1 Introduction

Several important recent works have applied powerful verification methods based on full-fledged inductive theorem proving to verify important systems artifacts. CompCert is an optimizing compiler for a subset of the C programming language, for which semantics preservation has been proved in the COQ proof assistant (see [12], and many other papers at the project web page, `compcert.inria.fr`). The seL4 microkernel verification effort uses the Isabelle theorem prover to prove that the microkernel implementation in C and assembly refines a high-level non-deterministic model expressing the desired system properties [10]. These impressive verification efforts show that the trustworthiness of practical systems artifacts can be raised to the highest levels currently known, using interactive theorem proving.

In a similar spirit, this paper presents `versat`, an efficient SAT solver for which we have verified correctness of "unsat" answers. SAT and SMT solvers are critical components of automatic verification tools like bounded model-checkers and k-induction provers [9, 5], and are used for many other static analysis applications, such as symbolic execution [11]. However, just as any complex piece of software, SAT solvers do have bugs. Brummayer et al. reports crashes and incorrect answers from top-ranked solvers at the SAT competition 2007 and 2009 [4]. This paper represents a first step towards the development of verified high-performance analysis tools, by verifying correctness of "unsat" answers from a modern performant SAT solver. Just as operating systems and compilers are the foundations for computing systems generally, SAT and SMT solvers are increasingly the foundation for analysis and automatic verification tools. So we see SAT solvers as artifacts of fundamental interest, and hence natural targets for verification.

Specification. We have proved statically that whenever `versat` reports a set of input clauses unsatisfiable, then there exists a resolution proof of the empty clause from those input clauses. This proof (as a data structure) is not constructed at run-time. Rather, our verification confirms statically that it exists, for all formulas `versat` reports unsatisfiable. As our verification is itself constructive, the resolution proof could in principle be generated at run-time. But run-time proof-production imposes undesirable time and memory overhead on SAT solving. So it is preferable to have a static guarantee of soundness for the solver, at least for applications that do not need the actual proof artifact, but only require a trustworthy result.

Main contribution. What makes our work distinctive is that it is, to the best of our knowledge, the first to statically verify soundness of a SAT solver implemented using efficient low-level data structures. These include 32-bit machine integers for literals and mutable C arrays for many solver data structures (e.g., clauses and look-up tables), which are manipulated using machine arithmetic/bitwise operations, and low-level pointer managements. In GURU, machine integers and their operations are precisely modeled as bit vectors and vector operations, including overflow situations. This does increase the burden of proof, but is necessary for performance. We demonstrate (Section 5) that `versat` can solve large benchmarks, including some on the order of those used in the SAT Competition. While further work would be required to achieve levels of performance closer to the current state of the art in SAT solving, `versat` is already valuable as being the first high-performance SAT solver that can deliver trustworthy results without the overhead of proof production (and subsequent proof checking). Furthermore, as `versat` already includes verified implementations of many of the standard modern solver data structures including those for watched literals and efficient conflict analysis, we hypothesize that our approach will scale to additional solver optimizations.

Verification approach. This project also represents a major case study of a verification approach which is gaining importance, particularly within the Programming Languages community. The `versat` code has been developed and verified in a so-called *dependently typed* programming language called GURU [20]. The basic methodology of dependently typed programming is to express rich specifications through types, and include proofs (when needed) only internally, inside program code. Such proofs establish properties externally of functions used in expressing the specification. Such specificational functions are typically much smaller and more tractable than the programs they specify, thus reducing the burden of proof. GURU implements this approach to program verification, and also provides a static analysis for statically tracking memory. Thanks to this analysis, GURU does not require a garbage collector for memory management at runtime.

Paper outline. We begin with a brief summary of dependent types for verified programming in GURU (Section 2). We then describe in more detail the specification we have statically verified for `versat` (Section 3). Next, we describe the actual implementation, and how we verify that it meets our specification (Section 4). We present empirical results supporting our claim that `versat`'s performance is within the realm of modern SAT solving (Section 5). We next cover important related work (Section 6),

and then reflect a little on the experience of implementing an efficient verified SAT solver (Section 7), before concluding (Section 8).

2 Verified Programming in Guru

By way of background for the sections on the specification and implementation of `versat` below, we begin with a quick introduction to GURU. GURU is a functional programming language with rich types, in which programs can be verified both *externally* (as in traditional theorem provers), and *internally* (cf [1]).[1] For a standard example of the difference, suppose we wish to prove that the result of appending two lists has length equal to the sum of the input lengths.

External verification of this property may proceed like this. First, we define the type of `append` function on lists. In GURU syntax, the typing for this `append` function is:

```
append : Fun(A:type)(l1 l2 : <list A>). <list A>
```

This says that `append` accepts a type `A`, and lists `l1` and `l2` holding elements of type `A`, and produces another such list. To verify the desired property, we write a proof in GURU's proof syntax of the following formula:

```
Forall(A:type)(l1 l2:<list A>).
  { (length (append l1 l2)) = (plus (length l1) (length l2)) }
```

The equality listed expresses, in GURU's semantics, that the term on the left-hand side evaluates to the same value as the term on the right-hand side. So the formula states that for all types `A`, for all lists `l1` and `l2` holding elements of that type, calling the `length` function on the result of appending `l1` and `l2` gives the same result as adding the lengths of `l1` and `l2`. This is the external approach.

With internal verification, we first define an alternative *indexed* datatype for lists. A type index is a program value occurring in the type, in this case the length of the list. We define the type `<vec A n>` to be the type of lists storing elements of type `A`, and having length `n`, where `n` is a Peano (i.e., unary) number:

```
Inductive vec : Fun(A:type)(n:nat).type :=
  vecn : Fun(A:type).<vec A Z>
| vecc : Fun(A:type)(spec n:nat)(a:A)(l:<vec A n>).
              <vec A (S n)>.
```

This states that `vec` is inductively defined with constructors `vecn` and `vecc` (for `nil` and `cons`, respectively). The return type of `vecc` is `<vec A (S n)>`, where `S` is the successor function. So the length of the list returned by the constructor `vecc` is one greater than the length of the sublist `l`. Note that the argument `n` (of `vecc`) is labeled "`spec`", which means specificational. GURU will enforce that no run-time results will depend on the value of this argument, thus enabling the compiler to erase all values for that parameter in compiled code.

We can now define the type of `vec_append` function on vectors:

[1] GURU is freely downloadable from `http://www.guru-lang.org/`.

```
vec_append : Fun(A:type)(spec n m:nat)
                (l1:<vec A n>)(l2:<vec A m>).<vec A (plus n m)>
```

This type states that `append` takes in a type `A`, two specificational natural numbers `n` and `m`, and vectors `l1` and `l2` of the corresponding lengths, and returns a new vector of length `(plus n m)`. This is how internal verification expresses the relationship between lengths which we proved externally above. Type-checking code like this may require the programmer to prove that two types are equivalent. For example, a proof of commutativity of addition is needed to prove `<vec A (plus n m)>` equivalent to `<vec A (plus m n)>`. Currently, these proofs must mostly be written by the programmer, using special proof syntax, including syntax for inductive proofs.

GURU supports memory-safe programming without garbage collection, using a combination of techniques [19]. Immutable tree-like data structures are handled by reference counting, with some optimizations to avoid unnecessary increments/decrements. Mutable data structures like arrays are handled by statically enforcing a readers/writers discipline: either there is a unique reference available for reading and writing the array, or else there may be multiple read-only references. The one-writer discipline ensures that it is sound to implement array update destructively, while using a pure functional model for formal reasoning. The connection between the efficient implementation and the functional model is not formally verified, and must be trusted. This is reasonable, as it concerns only a small amount of simple C code (less than 50 lines), for a few primitive operations like indexing a C array and managing memory/pointers.

3 Specification

The main property of `versat` is the soundness of the solver on top of the basic requirements of GURU, such as memory safety and array-bounds checking. We encoded the underlying logic of SAT in GURU to reason about the behavior of the SAT solver. That encoding includes the representation of formulas and the deduction rules. For a "UNSAT" answer, our specification requires that there exists a derivation proof of the empty clause from the input formula. Note that most solvers can generate a model with a "SAT" answer and those models can be checked very efficiently. So, we do not think there is a practical advantage for statically verifying the soundness of "SAT" answers. Also, it is important to note that the specification is the only part we need to trust. So, it should be clear and concise. The specification of `versat` is only 259 lines of GURU code. The rest of `versat` is the actual implementation and the proof that the implementation follows the specification, which will be checked by the GURU type system.

3.1 Representation of CNF Formula

The `formula` type is defined using simple data structures: 32 bit unsigned integers for literals and lists for clauses and formulas. The lower 31 bits of the literal represent the variable number, and the most significant bit represents the polarity. The GURU definitions of those types are listed below. The `word` type is defined in GURU's standard library, and represents 32 bit unsigned integers. We emphasize that these simple data

structures are only for specification. Section 4 describes how our verification relates them to efficient data structures in the implementation.

```
Define lit := word
Define clause := <list lit>
Define formula := <list clause>
```

3.2 Deduction Rules

There may be different ways to specify the unsatisfiability of formula. One could be a model theoretic definition, saying no model evaluates a formula true or $\Phi \vDash \bot$. Another could be a proof theoretic one, saying the empty clause (False) can be deduced from the formula or $\Phi \vdash \bot$. In the propositional logic, the above two definitions are equivalent. In `versat`, we have taken a weaker variant of the proof theoretic definition, $\Phi \vdash_{res} \bot$ where only the resolution rule is used to refute the formula. Because $\vdash_{res}$ is strictly weaker than $\vdash$, $\Phi \vdash_{res} \bot$ still implies $\Phi \vDash \bot$. So, even though our formalization is proof theoretic, it should be possible to prove that our formalization satisfies a model theoretic formalization.

The `pf` type encodes the deduction rules of the propositional logic and `pf` objects represents proofs. Figure 1 shows the definition of `pf` type and its helper functions. `cl_subsume` is a predicate that means `c1` subsumes `c2`, which is just a subset function on lists defined in GURU's standard library. And `is_resolvent` is a predicate that means `r` is a resolvent of `c1` and `c2` over the literal `l`. Additionally, `cl_has` checks that the clause contains the given literal, and `cl_erase` removes all the occurrences of the literal in the clause. Also, `tt` and `ff` are Boolean values defined in the library. The `<pf F C>` type stands for the set of proofs that the formula `F` implies the clause `C`. Members of this type are constructed as derivation trees for the clause. Because this proof tree will not be generated and checked at run-time, the type requires the proper preconditions at each constructor. GURU's type system ensures that those proof objects are valid by construction.

The `pf_asm` constructor stands for the assumption rule, which proves any clause in the input formula. The `member` function looks for the clause C in the formula, returning `tt` if so. The `pf_sub` constructor stands for the subsumption rule. This rule allows to remove duplicated literals or change the order of literals in a proven clause. Note that the constructor requires a proof `d` of `C'` and a precondition `u` that `C'` subsumes `C`. Finally, `pf_res` stands for the resolution rule. It requires two clauses (`C1` and `C2`) along with their proofs (`d1` and `d2`) and the precondition `u` that `C` is a resolvent of `C1` and `C2` over the literal `l`.

3.3 The `answer` Type

In order to enforce soundness, the implementation is required to have a particular return type, called `answer`. So, if the implementation type checks, it is considered valid under our specification. Figure 2 shows the definition of the `answer` type. The `answer` type has two constructors (or values): `sat` and `unsat`. The `unsat` constructor holds two subdata: the input formula `F` and a derivation proof of the empty clause, `p`. The formula

```
Define cl_subsume := fun(c1:clause)(c2:clause).
  (list_subset lit eq_lit c1 c2)

Define is_resolvent := fun(r:clause)(c1:clause)(c2:clause)(l:lit).
  (and (and (cl_has c1 (negated l))
            (cl_has c2 l))
       (and (cl_subsume (cl_erase c1 (negated l)) r)
            (cl_subsume (cl_erase c2 l) r)))

Inductive pf : Fun(F : formula)(C : clause).type :=
  pf_asm : Fun(F : formula)(C:clause)
              (u : { (member C F eq_clause) = tt }) .<pf F C>
| pf_sub : Fun(F : formula)(C C' : clause)
              (d : <pf F C'>)
              (u : { (cl_subsume C' C) = tt }) .<pf F C>
| pf_res : Fun(F : formula)(C C1 C2 : clause)(l : lit)
              (d1 : <pf F C1>)(d2 : <pf F C2>)
              (u : { (is_resolvent C C1 C2 l) = tt }) .<pf F C>
```

Fig. 1. The `pf` data type and helper functions

`F` is required to make sure the proof indeed proves the input formula. The term `(nil lit)` means the empty list of literals, meaning the empty clause. By constructing a value of the type `<pf F (nil lit)>`, we know that the empty clause is derivable from the original formula. (Note that the proof `p` is marked as specificational using the `spec` keyword) The type checker still requires the programmer to supply the `spec` arguments. However, those arguments will be erased during compilation. We only care about the existence of such data, not the actual value. By constructing proofs only from the invariants of the solver, GURU's type system confirms that such proofs could always be constructed without fail. So, making them specificational, hence not computing them at run-time, is sound.

```
Inductive answer : Fun(F:formula).type :=
  sat : Fun(spec F:formula).<answer F>
| unsat : Fun(spec F:formula)(spec p:<pf F (nil lit)>).<answer F>
```

Fig. 2. The definition of the `answer` type

3.4 Parser and Entry Point

The formula type above is still in terms of integer and list data structures, not a stream of characters as stored in a benchmark file. The benchmark file has to be translated to GURU data structure before it can be reasoned about. So, we include a simple recursive

parser for the DIMACS standard benchmark format, which amounts to 145 lines of GURU code, as a part of our specification. It might be possible to reduce this using a verified parser generator, but we judge there to be more important targets of further verification. Similarly, the main function is considered a part of the specification, as the outcome of the solve function is an `answer` value, not the action of printing "SAT" or "UNSAT". The main function simply calls the parser, passes the output to the solve function, and prints the answer as a string of characters.

4 Implementation and Proof

The specification in Section 3 does not constrain the details of the implementation very much. For a trivial example, a solver that just aborted immediately on every input formula would satisfy the specification. So would a solver that used the naive data structures for formulas, or a naive solving algorithm. Therefore, we have imposed an additional informal constraint on `versat`, which is that it should use efficient low-level data structures, and should implement a number of the essential features of modern SAT solvers. The features implemented in `versat` are conflict analysis, clause learning, backjumping, watched literals, and basic decision heuristics. Also, each of these features is implemented using the same efficient data structures that can be found in a C-language implementation like `tinisat` or `minisat`. However, implementing more features and optimizations make it more difficult to prove the soundness property.

Modern SAT solvers are driven by conflict analysis, during which a new clause is deduced and recorded to guide the search. Thus, the critical component for soundness is the conflict analysis module, which can be verified, to some extent, in isolation from the rest of the solver. Verifying that every learned clause is a consequence of the input clauses ensures the correctness of UNSAT answers from the solver, in the special case of the empty clause. Using the internal-verification approach described in the previous section, the conflict analysis module enforces soundness by requiring that with each learned clause added to the clause database, there is an accompanying specificational proof (the `pf` datatype described in Section 3). In this section, we explain some of the run-time clause data structure along with the invariants, and the conflict analysis implementation.

4.1 Array-based Clauses and Invariants

In the specification, the data structure for clauses is (singly linked) list, which is easier to reason about. However, accessing elements in a list is not as efficient as an array. The elements of an array are more likely in the same cache line, which leads to a faster sequential access, as elements in a linked list are not. Also arrays will use less memory than lists, which need extra storage for pointers. So, at the implementation level, `versat` uses array-based clauses with invariants as defined in Figure 3. An `<array lit n>` object stands for an array of literals of size `n`. The variable `nv` represents the number of different variables in the formula `F`. It is also the maximum possible value for variable numbers as defined in the DIMACS file format (variables are named from 1 up to `nv`). The predicate `(array_in_bounds nv l)` used in the invariant `ul` means

every variable in the array `l` is less than or equal to `nv` and not equal to zero. The invariant `u1` is used to avoid run-time checks for bounds when accessing a number of look-up tables indexed by the variable number, such as the current assignment, reference to the antecedent clauses, and decision level for the variable. It also implicitly states that the array `l` is null-terminated. In array-based clauses, the word value zero is used as the termination marker, instead of keeping a separate run-time variable for the length of the array. The second invariant `u2` states that the clause `c`, which is proved by `pf_c`, is the same as the interpretation of `l`, where `to_cl` is our interpretation of an array as a list. Again, this interpretation is only specificational and not performed at run-time.

```
Inductive aclause : Fun(nv:word)(F:formula).type :=
  mk_aclause : Fun(spec n:word)(l:<array lit n>)
                  (spec nv:word)(spec F:formula)
                  (u1:{ (array_in_bounds nv l) = tt })
                  (spec c:clause)(spec pf_c:<pf F c>)
                  (u2:{ c = (to_cl l) })
                 .<aclause nv F>
```

Fig. 3. The `aclause` type for the array-based clauses and invariants

At the beginning of execution, `versat` converts all input clauses into `<aclause nv F>` objects. In order to satisfy the invariants, the conversion function checks that every variable is within bounds and internally proves that the interpretation of the output array is exactly the same as the input list-based clause. Then, every time a new clause is learned, a new `<aclause nv F>` object is created and stored in the clause database. Remember the soundness of the whole solver requires a `<pf F (nil lit)>` object, which is a proof of the list-based empty clause. Assume we derived the empty array-based clause at run-time. From the invariant `u2`, we know that there exists an interpretation of the array clause. And we proved a theorem which states that the only possible interpretation of the empty array is the empty list, `(nil lit)`. Now, we can conclude that the interpretation is indeed the empty list-based clause, which is proven valid according to another invariant `pf_c`. Thus, it suffices to compute the empty array-based clause to prove the empty list-based clause.

4.2 Conflict Analysis with Optimized Resolution

The conflict analysis is where a SAT solver deduces a new clause from the existing set of clauses by resolution. Usually, a series of resolutions are applied until the first unique implication point (UIP) clause is derived. In order to speed up the resolution step, advanced solvers like `minisat` use a number of related data structures to represent the intermediate conflict clauses and perform resolutions efficiently. In `versat`, we implemented this optimized resolution and proved the implementation is sound according to the simple definition of `is_resolvent` in the specification.

Figure 4 shows the data structure and invariants of intermediate conflict clauses, `ResState`, which are maintained after each resolution step over the course of conflict

analysis. Those invariants are sufficient to prove the soundness of versat's conflict analysis. Figure 5 summarizes the variables used in the ResState type. The conflict clause is split into the literals assigned at the previous decision levels (c1) and the literals assigned at the current level (c2) according to the invariant u5. So, the complete conflict clause at the time is (append c1 c2). Notice that c2 is declared as a specificational data with the spec keyword. During conflict analysis, versat does not build each intermediate conflict clause as a single complete clause. Instead, the whole conflict clause is duplicated in a look-up table (vt), and it keeps track of the number of literals assigned at the current level, which is the c2l, as stated by the invariants u1, u2 and u3. The u2 and u3 ensure that the conflict clause and the table contain exactly the same set of literals. The look-up table vt enables a constant time check whether a literal is in the conflict clause, which makes duplication removal and other operations efficient. And it also enables a constant time removal of a literal assigned at the current level, which can be done by unmarking the literal on the vt and decrementing the value of c2l by one. That also requires all literals in the list c2 to be distinct (u4), so that removing all occurrences of a literal (as in the specification) will decrease the length only by one (in the implementation). Note that, although the type of c2l is nat (the Peano number), incrementing/decrementing by one and zero testing are constant time operations just like the machine integer operations. Also, note that, some invariants, i.e. all variables are within bounds, are omitted in the figure for clarity.

```
Inductive ResState : Fun(nv:word)(dl:word).type :=
  res_state : Fun
    (spec nv:word)
    (spec dl:word)
    (dls:<array word nv>)
    (vt:<array assignment nv>)
    (c1:clause)
    (spec c2:clause)
    (c2l:nat)
    (u1:{ c2l = (length c2) })
    (u2:{ (all_lits_are_assigned vt (append c2 c1)) = tt })
    (u3:{ (cl_has_all_vars (append c2 c1) vt) = tt })
    (u4:{ (cl_unique c2) = tt })
    (u5:{ (cl_set_at_prev_levels dl dls c1) = tt })
    .<ResState nv dl>
```

Fig. 4. The datatype for conflict analysis state

For the resolution function, we have proved that the computation of the resolvent between the previous conflict clause and the antecedent clause follows the specification of is_resolvent, so that a new pf object for the resolvent can be constructed. At the end of the conflict analysis, versat will find the Unique Implication Point (UIP) literal, say l, and the ResState value will have one as the value of c2l. Because the UIP literal must be assigned at the current decision level, it should be in c2 and the

Variable	Description
nv	the number of variables in the formula
dl	the current decision level
dls	a table of the decision levels at which each variable is assigned
vt	a look-up table for the variables in the conflict clause
c1	the literals of the conflict clause assigned at the previous decision levels
c2	the literals of the conflict clause assigned at the current decision level
c2l	the length of c2 (the number of literals assigned at the current decision level)
u1	the length of the list c2 is the same as the value of c2l
u2	all the literals in the conflict clause are marked on the table
u3	all the literals marked on the table are in the conflict clause
u4	all literals in the list c2 are unique
u5	all variables in c1 are assigned at the previous decision levels

Fig. 5. Summary of variables used in ResState

length of c2 is one due to the invariant u1. That means actually c2 is a singleton list that consists of l. Thus, the complete conflict clause is (cons lit l c1). Then, an array-base clause can be constructed and stored in the clause database, just as the input list-based clauses are processed at the beginning of execution. Finally, versat clears up the table vt by unmarking all the literals to recycle for the next analysis. Instead of sweeping through the whole table, versat only unmarks those literals in the conflict clause. It can be proved that after unmarking those literals, the table is clean as new using the invariant u3 above. Correctness of this clean-up process is proved in around 400 lines of lemmas, culminating in the theorem in Figure 6, which states that the efficient table-clearing code (clear_vars) returns a table which is indistinguishable from a brand new array (created with array_new).

```
Define cl_has_all_vars_implies_clear_vars_like_new :
  Forall (nv:word)
         (vt:<array assignment nv>)
         (c:clause)
         (u:{ (cl_valid nv c) = tt })
         (r:{ (cl_has_all_vars c vt) = tt })
    .{ (clear_vars vt c) = (array_new nv UN) }
```

Fig. 6. The theorem stating correctness of table-clearing code

4.3 Summary of Implementation

The source code of versat totals 9884 lines, including proofs. It is hard to separate proofs from code because they can be intermixed within a function. Roughly speaking, auxiliary code (to formulate invariants) and proofs take up 80% of the entire pro-

gram. The generated C code weighs in at 12712 lines. The C code is self-contained and includes the translations of GURU's library functions being used. The source and generated C code are available at `http://cs.uiowa.edu/~duoe/versat`. All lemmas used by `versat` have been machine-checked by the GURU compiler.

Properties not proved. First, we do not prove termination for `versat`. It could (*a priori*) be the case that the solver diverges on some inputs, and it could also be the case that certain run-time checks we perform (discussed in Section 7) fail. These termination properties have not been formally verified. However, what users want is to solve problems in a reasonable amount of time. A guarantee of termination does not satisfy users' expectations. It is more important to evaluate the performance over real problems as we show in Section 5. Second, we have not verified completeness of `versat`. It is (again *a priori*) possible that `versat` reports satisfiable, but the formula is actually unsatisfiable. In fact, we include a run-time check at the end of execution, to ensure that when `versat` reports SAT, the formula does have a model. But it would take substantial additional verification to ensure that this run-time check never fails.

5 Evaluation

We compared `versat` to `picosat-936`[2] with proof generation and checking for certified UNSAT answers. `picosat` is one of the best SAT solvers and can generate proofs in the RUP and TraceCheck formats. The RUP proof format is the official format for the certified track[3] of the SAT competition, and `checker3` is used as the trusted proof checker. The TraceCheck format[4] is `picosat`'s preferred proof format, and the format and checker are made by the developers of `picosat`. We measured the runtime of the whole workflow of solving, proof generation, and checking in both of the formats over the benchmarks used for the certified track of the SAT competition 2007. The certified track has not been updated since then.

Figure 7 shows the performance comparison. The "versat" column shows the solving times of `versat`. The "picosat(R)" and "picosat(T)" columns shows the solving and proof generation times of `picosat` in the RUP format and TraceCheck format, respectively. Since `checker3` does not accept the RUP format directly, `rupToRes` is used to convert RUP proofs into the RES format, which `checker3` accepts. The "rupToRes" column shows the conversion times, and the "checker3" column shows the times for checking the converted proofs. The "tracecheck" column shows the checking times for the proofs in the TraceCheck format. The "Total(R)" and "Total(T)" shows the total times for solving, proof generation, conversion (if needed), and checking in the RUP format and TraceCheck format, respectively. The unit of the values is in seconds. "T" means a timeout and "E" means a runtime error before timeout. The machine used for the test was equipped with an Intel Core2 Duo T8300 running at 2.40GHz and

[2] `picosat` is available at `http://fmv.jku.at/picosat/`

[3] Information about the certified track, including the RUP/RES proof formats and `checker3`/`rupToRes`, is available at `http://users.soe.ucsc.edu/~avg/ProofChecker/`

[4] TraceCheck is available at `http://fmv.jku.at/tracecheck/`

3GB of memory. The time limits for solving, conversion, and checking were all 3600 seconds, individually.

Benchmarks	versat	picosat(R)	rupToRes	checker3	Total(R)	picosat(T)	tracecheck	Total(T)
itox_vc965	1.74	0.18	0.88	0.36	1.42	0.18	0	0.18
dspam_dump_vc973	3565	0.57	2.32	0.99	3.88	0.55	0.01	0.56
eq.atree.braun.7.unsat	15.43	2.63	42.13	2.78	47.54	2.92	1.1	4.02
eq.atree.braun.8.unsat	361.11	24.11	642.35	E	E	26.47	9.11	35.58
eq.atree.braun.9.unsat	T	406.94	T		T	356.03	62.68	418.71
AProVE07-02	T	T			T	T		T
AProVE07-15	T	93.94	T		T	103.95	20.44	124.39
AProVE07-20	T	262.05	T		T	272.39	95.87	368.26
AProVE07-21	T	1437.64	T		T	1505.24	E	E
AProVE07-22	T	196.28	T		T	239.59	116.8	356.39
dated-5-15-u	T	2698.49	E		E	2853.12	E	E
dated-10-11-u	T	T			T	T		T
dated-5-11-u	T	255.06	E		E	266.6	23.36	289.96
total-5-11-u	1777.26	91.27	E		E	109.94	32.42	142.36
total-5-13-u	T	560.96	E		E	629.53	151.23	780.76
manol-pipe-c10nidw_s	772.68	25.46	7.38	1.37	34.21	25.54	0.1	25.64

Fig. 7. Results for the certified track benchmarks of the SAT competition 2007

`versat` solved 6 out of 16 benchmarks. Since UNSAT answers of `versat` are verified by construction, `versat` was able to certify the unsatisfiability of those 6 benchmarks. `picosat` could solve 14 of them and generated proofs in both of the formats. However, the RUP proof checking tool chain could only verify 4 of the RUP proofs within additional 2 hour timeouts (1 hour for conversion and 1 hour for checking). So, `versat` was able to certify the two more benchmarks that could not be certified using `picosat` and the official RUP proof checking tools. On the other hand, `tracecheck` could verify 12 of 14 TraceCheck proofs. Note that the maximum proof size was about 4GB and the disk space was enough to store the proofs.

When comparing the trusted base of those systems, `versat`'s trusted base is the GURU compiler, some basic datatypes and functions in the GURU library, and 259 lines of specification written in GURU. `checker3` is 1538 lines of C code. `tracecheck` is 2989 lines of C code along with 7857 lines of `boolforce` library written in C. Even though `tracecheck` is the most efficient system, the trusted base is also very large. One could argue that GURU compiler is also quite large (19175 lines of Java). However, because the GURU compiler is a generic system, it is unlikely to generate an unsound SAT solver from the code that it checked, and the verification cost of GURU compiler itself, if needed, should be amortized across multiple applications.

General performance. We measured the solving times of `versat`, `minisat-2.2.0`, `picosat-936` and `tinisat-0.22` over the SAT Race 2008 Test Set 1, which was used for the qualification round for the SAT Race 2008. The machine used for the measurement was equipped with an Intel Xeon X5650 running at 2.67GHz and 12GB of

memory. The time limit was 900 seconds. In summary, `versat` solved 19 out of 50 benchmarks in the set. `minisat` solved 47. `picosat` solved 46. `tinisat` solved 49. `versat` is not quite comparable with those state-of-the-art solvers, yet. However, to our best knowledge, `versat` is the only verified solver at the actual code level that could solve those competition benchmarks.

6 Related Work

Verifying the correctness of each individual result of a solver is generally believed to be easier than verifying the solver itself. For this reason, fields like SMT, where solvers are typically on the order of several tens of thousands of lines of code, have usually relied on result verification rather than solver verification. For example, there are many recent works on producing proofs from SMT solvers, and several solvers, including VERIT, Z3, and CVC3, can produce independently checkable proofs of the formulas they claim are valid [3, 17, 15, 16, 8]. Challenges in this area are devising a common proof format for SMT solvers, minimizing the overhead of proof production, and efficient proof checking.

There have been a number of works aimed at verifying automated reasoning algorithms. Lescuyer and Conchon verified a modern DPLL-based SAT solver in COQ and extraced OCAML code to compile into machine binary [13]. Also, Shankar and Vaucher's work verifying a modern SAT solver is a noteworthy example [18]. They concede, though, that while their model in PVS can be extracted to executable code, that code would not be as efficient as an implementation intended for high-performance use.

More closely related work is Marić's formal proof of correctness for a modern SAT solver implementing some low-level optimizations like watched literals in ISABELLE/HOL [14]. He proves soundness, completeness, and termination for a modern SAT solver written in ISABELLE's pure functional programming language. This is a major achievement, requiring around 25,000 lines of ISABELLE proof scripts. Marić proves much more about the solver than we do. In particular, proving termination would require extensive additional work for `versat` (see Section 7 below). But like Shankar and Vaucher, Marić verifies a functional model of a solver. This model uses pure-functional lists to represent clauses, and Peano naturals for variables. In contrast, `versat` uses mutable C arrays to represent clauses, and 32-bit machine words for literals. Also, Marić's resolution code is not optimized using a look-up table like `versat` and other modern solvers. Our work can thus be viewed as taking up Marić's concluding challenge to verify high-performance SAT solvers with efficient low-level data structures. To our knowledge, our work is the first to verify a deep property of a high-performance implementation of a modern solver.

Armand et al. extend COQ with support for more efficient data structures, including imperative arrays. They use their extended language to implement and verify an efficient checker for proofs produced by an external (to COQ) SAT solver [2]. In contrast, we have verified soundness of the efficient SAT solver itself. Both approaches use an inefficient functional model of arrays for formal reasoning, which is then replaced during compilation with a more efficient implementation. In our case, thanks to GURU's

resource typing, this implementation is simply C arrays. Our readers/writers analysis ensures this is sound, even with destructive updates. Armand et al. use less efficient *persistent arrays* to combine destructive updating with persistence of old versions of the array [6]. Also, while GURU does not require run-time garbage collection, Armand et al. rely on compilation to OCAML (a garbage-collected language). GC overhead can be substantial in practice [21]. Also, it is noteworthy that Darbari et al. implemented an efficient TraceCheck proof checker in COQ [7].

7 Discussion

The idea for the specification was clear, and the specification did not change much since the beginning of the project. However, the hard part was formalizing invariants of the conflict analysis all the way down to the data structures and machine words, let alone actually proving them. Modern SAT solvers are usually small, but highly optimized as several data structures are cleverly coupled with strong invariants. The source code of `minisat` and `tinisat` does not tell what the invariants it assumes. As we discovered new invariants, we had to change our verification strategy several times along the development. Sometimes, we compromised and slightly modified our implementation. For example, the look-up table `vt`, used for resolution to test the membership of variable in the current conflict clause, could be an array of booleans. Instead, we used an array of `assignment`, which has three states of true, false, and unassigned. The other solvers assume the current assignment table already contains the polarity of each variable, which is an additional invariant. In `versat`, the table marks variables with the polarity, which duplicates information in the assignment table, avoiding the invariant above.

Unimplemented features. Some features not implemented in `versat` includes conflict clause simplification and restart. Conflict clause simplification feature requires to prove that there exists a certain sequence of resolutions that derives the simplified clause. Although the sequence can be computed by topologically sorting the removed literals at run-time, additional invariants would be required to prove it statically.

Run-time checks. Certain properties of `versat`'s state are checked at run-time, like assert in C. We tried to keep a minimal set of invariants and it is simply not strong enough to prove some properties. Run-time checks makes the solver incomplete, because it may abort. Also, it costs execution time to perform such a check. In principle, all of these properties could be proved statically so that those run-time checks could be avoided. However, stronger invariants are harder to prove. Some would require a much longer development time and may not speed up the solver very much. Thus, the priority is the tight loops in the unit propagation and resolution. However, one-time procedures like initialization and the final conflict analysis are considered a lower priority. We did not measure how much those run-time checks cost, however, `gprof` time profiler showed that they are not bottlenecking `versat`.

Verified programming in GURU. GURU is a great tool to implement efficient verified software, and the generated C code can be plugged into other programs. Optimizing software always raises the question of correctness, where the source code can get complicated as machine code. In those situations, GURU can be used to assure the

correctness. However, some automated proving features are desired for general usage. Because `versat` heavily uses arrays, array-bounds checking proliferates, which requires a fair amount of arithmetic reasoning. At the same time, when code changes over the course of development, those arithmetic reasonings are the most affected and need to be updated or proved again. So, automated reasoning of integer arithmetic would be one of the most desired feature of GURU, allowing the programmer to focus on more higher level reasonings.

8 Conclusion and Future Work

`versat` is the first modern SAT solver that is statically verified to be sound all the way down to machine words and data structures. And the generated C code can be compiled to binary code without modifications or incorporated into other software. This paper has shown that the sophisticated invariants of the efficient data structures used in modern SAT solvers can be formalized and proved in GURU. Future work includes proving remaining lemmas and tuning the performance of `versat`. And we envision that the code and lemmas in `versat` can be applied to other SAT-related applications.

References

1. T. Altenkirch. Integrated verification in Type Theory. Lecture notes for a course at ESSLLI 96, Prague, 1996. Available from the author's website.
2. M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. pages 83–98, 2010.
3. T. Bouton, D. Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In R. Schmidt, editor, *22nd International Conference on Automated Deduction (CADE)*, pages 151–156, 2009.
4. Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of sat and qbf solvers. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010.
5. Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
6. S. Conchon and J.-C. Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 37–46. ACM, 2007.
7. Ashish Darbari, Bernd Fischer, and João Marques-Silva. Industrial-strength certified sat solving through verified sat proof checking. In Ana Cavalcanti, David Déharbe, Marie-Claude Gaudel, and Jim Woodcock, editors, *ICTAC*, volume 6255 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2010.
8. L. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In B. Konev, R. Schmidt, and S. Schulz, editors, *7th International Workshop on the Implementation of Logics (IWIL)*, 2008.
9. George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In A. Cimatti and R. Jones, editors, *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (Portland, Oregon)*, pages 109–117. IEEE, 2008.

10. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In J. Matthews and T. Anderson, editors, *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220. ACM, 2009.
11. Nupur Kothari, Todd Millstein, and Ramesh Govindan. Deriving state machines from tinyos programs using symbolic execution. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 271–282, Washington, DC, USA, 2008. IEEE Computer Society.
12. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In G. Morrisett and S. Peyton Jones, editors, *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
13. S. Lescuyer and S. Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.
14. F. Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411:4333–4356, November 2010.
15. S. McLaughlin, C. Barrett, and Y. Ge. Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. *Electr. Notes Theor. Comput. Sci.*, 144(2):43–51, 2006.
16. M. Moskal. Rocket-Fast Proof Checking for SMT Solvers. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
17. D. Oe, A. Reynolds, and A. Stump. Fast and Flexible Proof Checking for SMT. In B. Dutertre and O. Strichman, editors, *Workshop on Satisfiability Modulo Theories (SMT)*, 2009.
18. Natarajan Shankar and Marc Vaucher. The mechanical verification of a dpll-based satisfiability solver. *Electr. Notes Theor. Comput. Sci.*, 269:3–17, 2011.
19. A. Stump and E. Austin. Resource Typing in Guru. In J.-C. Filliâtre and C. Flanagan, editors, *Proceedings of the 4th ACM Workshop Programming Languages meets Program Verification, PLPV 2010, Madrid, Spain, January 19, 2010*, pages 27–38. ACM, 2010.
20. A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified Programming in Guru. In T. Altenkirch and T. Millstein, editors, *Programming Languges meets Program Verification (PLPV)*, 2009.
21. F. Xian, W. Srisa-an, and H. Jiang. Garbage collection: Java application servers' Achilles heel. *Science of Computer Programming*, 70(2-3):89 – 110, 2008.