Experimentation with Proving VCs Using No Special Theory Solvers

Technical Report RSRG-15-02

Nabil M. Kabbani, Joan Krone, and Murali Sitaraman
School of Computing
Clemson University
SC, 29634, USA

June 2015

Experimentation with Proving VCs Using No Special Theory Solvers

Nabil Michael Kabbani[1], Murali Sitaraman[1], and Joan Krone[2]

[1] School of Computing, Clemson University
Clemson, SC 29634, USA
[2] Department of Mathematics and Computer Science, Denison University
Granville OH 43023, USA
nkabban@clemson.edu, msitara@clemson.edu, krone@denison.edu

Abstract. The expression of specifications for a broad spectrum of software in a reasonably concise and readily comprehensible fashion requires the employment of a range of mathematical theories, many of which lack specialized decision procedures. Even if no decision procedures are available, if software component specifications and corresponding implementations are well-designed and annotated appropriately, the resulting verification conditions for correctness become "obvious," i.e., their correctness may be seen via a simple deduction by humans and established formally via the discovery of a short proof by an automated prover – without recourse to special-purpose solvers.
To explore just how obvious the program correctness VCs need to be and to understand the limitations of automated provers for proving VCs involving arbitrary theories, this paper presents results from experiments with two provers. One of them is Z3, with only the solver for uninterpreted functions and constants enabled. The other is a version of a prover based on congruence closures that we have developed to prove sequent-style VCs (as opposed to arbitrary mathematical assertions). Results from the experimentation are promising.

1 Introduction

Construction of a verifying compiler is a *Grand Challenge* [10]. A key component of the compiler is a prover to discharge verification conditions of correctness from programs. Ideally, proving of the VCs would be automated; Beyond supplying code annotations, such as invariants for loops or progress metrics for termination as required by most verification systems [13], programmers would not be involved in proofs of the VCs themselves. Such is the goal of a range of verification systems, including Dafny [12], KeY [4] and RESOLVE [17] among others.

Significant progress has been made in developing automated provers, especially in the form of decision procedures for a variety of theories and fragments [5,16,20]. These decision procedures are efficient solvers. So when VCs are within the scope of decision procedures, they are often the first choice.

Automated verification challenge takes on an additional dimension of complexity when software components are specified using a combination of theories

appropriate for the domain of discourse–including new theories conceived to simplify specifications. The theories may involve the use of higher order logic as well. In such cases, there are unlikely to be viable decision procedures. While interactive provers such as Coq [2] and Isabelle [15] may be used and they provide the additional benefit of being able to handle higher order logic as input, they are strictly proof assistants, so they often require user input to direct the proof. Use of such proof assistants are unavoidable for proving arbitrary theorems, but the VCs arising from programs can be made "obvious" to automated provers and these VCs are the topic of this paper.

One important way to simplify the VCs is to use proof rules for VC generation that limit and perhaps minimize the use of logical symbols in the VCs; i.e., rules that avoid clutter. More importantly, if software development is supported with well-engineered mathematical units, specifications, and implementations [18] can be proved automatically and no "deep thinking" is necessary [11].

There is an implied division of labor in proving VCs. The mathematical theorems (used in proving VCs) that reside in the reusable mathematical units that capture the theories may be proven using an external tool that can handle higher order logic, such as Coq or Isabelle. If the theories are well engineered, then specifications and code annotations that lead to VCs can be written in such a way that the consequent of the input sequent (VC) is always a predicate with constants as arguments. Given this setting, it is sufficient for an automated prover to use only instances of previously proven universally quantified statements (these are the reusable mathematical units above) to construct a proof, assuming that the assertions leading to VCs make such a proof relatively obvious.

Proving of VCs arising from establishing correctness of well-designed programs with well-designed supporting mathematical units–even when the VCs span theories for where no suitable decision procedures are available–is the topic of this paper. How hopeful are automated provers in this realm? To answer this question, we have experimented with two provers. The first one is MP, a prover integrated with our VC generator (detailed in [9]) and it is discussed in Section 3. MP is based on a congruence closure algorithm ([14]) and it is just as fast as Z3's uninterpreted function (UF) solver. The second one is a reduced version of Z3, that only includes its UF solver. For this experiment, we have developed a translator from our internal VC representation to SMT-LIB, a standard format for use as input to SMT solvers; among the interesting aspects of this translation is a representation of our type system in the many-sorted first-order logic required by SMT-LIB [3]. This is the topic of Section 4.

With both provers, proofs of most VCs for the examples in the experiment are more or less instantaneous. It is important to emphasize that the goal here is to check the viability of automated proving at all (and not how fast). For example, MP is integrated with our system, whereas use of Z3 goes through a translation process and invocation of an external piece that is outside our system. So speed considerations are not meaningful. MP is faster in some cases and reduced Z3 in others.

To exercise the provers, while we could use software specified with some new theories (e.g., theories that we have found to simplify specification of a class of components such as nested lists or heap structures), such an approach would have rendered the central points of this paper inaccessible. So we have taken a different approach. We have simply specified theories that describe mathematical strings (a structure simpler than sequences or arrays because no positions are involved) and numbers, and used them in component specifications, but have avoided using any decision procedures (e.g., for arithmetic) to tackle them. So as far as the provers are concerned, these theories are non-native and no special-purpose solvers are available. We also note that the theories include only a small set of general theorems (no more than would be needed in standard mathematics) and that they are not in any way tuned for the particular experiments conducted. Both the reduced version of Z3 and MP are promising, and suggest further exploration of the idea so that automated verification can reach components specified using new theories.

2 Example Specification, Implementation, and VC

As an example of a specification developed without recourse to data types handled by specialized decision procedures, we present the generic, bounded Stack_Template in Appendix A in RESOLVE. Stacks in this specification are conceptualized abstractly as mathematical strings (constrained by a user-supplied maximum depth), and alternative implementations may use arrays or linked structures to represent them. For modular verification of components that use stacks, internal details are irrelevant.

Mathematical operators on strings used in the specification include concatenation (denoted by `o`), string containing a single entry (`<x>`), length (`|alpha|` for a string alpha), and reverse (`Reverse`). Use of this model (instead of arrays, for example) in specifications of concepts such as stacks, queues, or lists can simplify a class of assertions and avoid the need for quantifiers.

Fig. 1 shows the specification of an *enhancement* operation to flip a `Stack`. Enhancements describe ancillary operations. In the `ensures` clause of `Flip`, input parameter value is denoted with `#`.

```
Enhancement Flipping_Capability for Stack_Template;
  Operation Flip(updates S: Stack);
    ensures S = Reverse(#S);
end Flipping_Capability;
```

Fig. 1. Specification of a Stack Flip Operation

VCs for the code in Fig. 2 are generated as detailed in [9] in a modular fashion, using only the specification of Stack_Template. Six VCs are generated

and they correspond to proving (i) the `ensures` clause, (ii) that the `maintaining` clause (invariant) holds before the loop, (iii) that it holds at the end of an arbitrary iteration, (iv) that the loop terminates with the aid of the `decreasing` clause, and (v) that the requires clauses of called operations `Pop` and `Push` hold before the calls. Fig. 3 shows the VC (iii) above for the first conjunct of the invariant. While dedicated solvers have been created for decidable fragments of the theory of strings (e.g., [1,5]) to prove such VCs, the present experiment concerns verification in the absence of special theory solvers. The VC's goal can be proven with just three instantiations of universally quantified statements in string theory.

```
Realization Obvious_Flipping_Realiz
        for Flipping_Capability of Stack_Template;
  Procedure Flip(updates S: Stack);
    Var Temp: Stack;
    Var Next_Entry: Entry;
    Var D: Integer;
    D := Depth(S);
    While (Less_Or_Equal(1, D))
      maintaining #S = Reverse(Temp) o S and D = |S|;
      decreasing |S|;
    do
      Pop(Next_Entry, S);
      Push(Next_Entry, Temp);
      D := Depth(S);
    end;
    S :=: Temp; -- swap operator
  end Flip;
end Obvious_Flipping_Realiz;
```

Fig. 2. Code for Flip

3 The Integrated Prover

This section summarizes the operation of the integrated prover, MP, which was designed expressly for the task of showing VC validity given a set of previously proven theorems in reusable math units. A technical description of the prover is not possible within the space limits (of even a longer paper).

Unlike most other automated provers, MP does not employ the proof by refutation technique, in which given a statement $P \Rightarrow Q$, the prover tries to determine the satisfiability of its negation: $P \wedge \neg Q$. Rather, it tries to support the truth of Q given the included theorem library as well as P. In our experience,

```
VC 0_5
Inductive Case of Invariant of While Statement:
    Obvious_Flipping_Realiz.rb(10)
Goal(s):
(S = (Reverse((<Next_Entry'> o Temp')) o S'))

Given(s):
1. (S'' = (<Next_Entry'> o S'))
2. (1 <= |S''|)
3. (S = (Reverse(Temp') o S''))
4. Entry.Is_Initial(Next_Entry)
5. (|S| <= Max_Depth)
6. (1 <= Max_Depth)
```

Fig. 3. Verification Condition

instances of sound code resulting in the generation of VCs with false antecedents are not the norm. When they do occur, they are normally fairly easy to detect.

At the core of MP is a simple congruence closure algorithm, similar in spirit to the one described in [14]. An outer layer that incorporates pattern matching techniques for expressions containing universally quantified variables is engaged, similar to the *matcher* described in [7].

Internally, the VC is stored as a structure that represents a set of equations. These equations are used to discover congruence among the symbols contained in the VC (this includes symbols created to represent subexpressions). The internal representation normalizes many aspects of the proof object.

The matcher finds instances of quantified expressions (the contents of the theorem library) that are *likely* to assist in proving the input sequent. The matcher is designed for efficiency rather than completeness. Currently, the prover only supports universally quantified expressions written in prenex normal form.

The current implementation employs a two tier matching heuristic. A brute force approach that attempts to apply all possible matching instances of quantified expressions repetitively will time out before completion in all but the most simple cases. Heuristics are employed to select the information most likely to be useful. Symbols of the VC are ranked according to the closeness of their association to the goal. Since the search process is computation intensive, symbol rank is first used to choose an uninstantiated theorem, then the search results from the chosen theorem are ranked, and the top result is conjunctively appended to the antecedent of the verification condition.

MP has benefitted from lessons learned in building a proof-of-concept prototype prover based on backtracking and rewriting earlier [18], but other than some shared goals, the provers are unrelated.

4 VC Translation to SMT-LIB

For the purposes of this experiment, we added an automatic SMT-LIB script generator into the compiler. By renaming all symbols other than the logical symbols, the script prevents the use of any domain specific decision procedures. The prover is forced to use the theorems that are translated into assertions. Since using types as sorts requires changing parts of the the code, specifications, and theorem libraries (collectively called a *workspace*), we have devised a method to express our standard type system in SMT-LIB using only logic. Types differ from sorts in that types may have a subset relation between them, while sorts are necessarily independent of one another.

We declare two sorts, one for types, another for symbols other than types. A type is a name for a set in our system, and declaring a symbol as a member of that type means that it is an element of the set that the type represents. A type in our system is declared as a constant of sort Type in SMT-LIB. Constants of type τ are declared as sort Symbol with a corresponding assertion that the constant is an element of τ.

We declare a predicate in SMT-LIB, `ElementOf`, that represents symbol-type membership in our mathematical system. `ElementOf` is used in SMT-LIB wherever we would declare type membership in our system. This includes constant declarations, function declarations, and in the quantified statements that define the functions used in our reusable mathematical theorem library.

```
(declare-sort Type)
(declare-sort Syms)
(declare-fun ElementOf(Syms Type) Bool)
```

Types in our system are declared as constants in SMT-LIB.

```
(declare-const N Type)
(declare-const Z Type)
```

The following type relation in our theorem library is translated to an assertion in SMT-LIB.

```
  For all n : N, n : Z;

(assert (forall ((s Syms)) ( => (ElementOf s N) (ElementOf s Z))))
```

Constants are declared as sort Syms, and an assertion is added that they are elements of the type they are declared to be in the source.

```
(declare-const @!0 Syms )
(assert (ElementOf @!0 N))
(declare-const @!1 Syms )
(assert (ElementOf @!1 N))
```

Mathematical function declarations in our theorem library provide domain and range type information. Here, for example is how integer addition is declared.

```
Definition (m: Z) + (n: Z): Z;
```

Since the type N is declared to be a subset of type Z, this same function may also accept natural number parameters. This prevents us from using types as sorts. Instead, the sort Syms is used (unless the function has a type as a parameter or returns a type, in which case sort Type is used). The following example demonstrates the translation of this declaration into SMT-LIB format.

```
(declare-fun @!+ (Syms Syms ) Syms)
(assert( forall((s0 Syms)(s1 Syms))
  (=>(and (EleOf s0 Z)(EleOf s01 Z))
    (EleOf(@!+ s0 s1)Z))))
```

This says that when the arguments to the addition function are integers, the result of the addition is also an integer.

The elements of our reusable mathematical library, called theorems, typically are universally quantified statements in prenex normal form. Quantified variables in the statement must be annotated with their types. The theorem becomes the succedent of an implication with the type restriction clause as the antecedent. The following example defines the length of a concatenated string.

```
;(|(alpha o beta)| = (|alpha| + |beta|))
(assert (forall((@!alpha  Syms)(@!beta  Syms))
  (=> (and (EleOf @!alpha  SStr)(EleOf @!beta  SStr))
    (= (@!l!l (@!o  @!alpha  @!beta ))
    (@!+ (@!l!l  @!alpha )(@!l!l  @!beta ))))))
```

The theorems and VC antecedent are added to the script along with the negation of the VC goal. The reduced Z3 performs quite well even with the additional complexity of the type system.

5 Experiment Results

For experimentation, we used VCs from a variety of examples involving stacks, queues, and preemptable queues (including several from an undergraduate software engineering class), such as those in the benchmarks in [19]. A representative sample is presented here.

For the first test, we used our standard workspace. This experiment produces VCs containing symbols that have types that are subsets of other types. The examples tested required only our String, Integer, and Natural Number theory. Four types are used: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{S}$ (the type of all Strings), and $Str(T)$ (the type of all Strings of type T, a generic type). It is asserted that $\mathbb{N} \subset \mathbb{Z}$ and $Str(T) \subset \mathbb{S}$. Type matching is a critical function of the component that finds valid applications of the universally quantified statements that comprise our theorem library. The integrated prover has the advantage of being able to call upon type-checking methods built into the compiler, whereas the SMT prover must use logic embedded in the SMT-LIB script to restrict instantiation based on type declaration as

	Integrated Prover	SMT Types in Logic	SMT Types as Sorts
	Proved/Total		
Inject_Front_Realiz	12/12	12/12	12/12
Iterative_Copying_Realiz	18/18	17/18	17/18
Obvious_CC_Realiz	34/34	32/34	34/34
Obvious_Flipping_Realiz	10/10	10/10	10/10
Obvious_Reading_Realiz	11/11	10/11	11/11
Recursive_Inverting_Realiz	5/5	5/5	5/5
Rotate_Realiz	3/3	3/3	3/3
Stack_Examples_Fac	12/12	9/12	12/12
Total Timeouts	0	7	1

Table 1.

described in Section 4. The results of tests conducted using our full type system are presented as columns 1 - 3 in Table 1.

The data in column 4 reports the results of a running an alternate SMT-LIB script generator with a workspace modified so that all types that appear in the VC are independent. The use of independent types allows types to be used as sorts in the script. This is accomplished by removing types $\mathbb{N}$ and $Str(T)$. Type $\mathbb{N}$ is replaced by $\mathbb{Z}$, and an assertion is added that the value previously declared as type $\mathbb{N}$ is non-negative. $Str(T)$ is simply replaced by $\mathbb{S}$, and we lose the type of the string contents (content type is not needed in these examples). The simpler assertions enable the verification of 6 more VCs using SMT than in the first test, however using this system limits the descriptive power of our specifications. Unprovable VCs were timed out at 20 seconds.

We conclude this section with an example to illustrate the extensibility of the approach for VCs using new notations. In the example below `Prt_Btwn` denotes the substring between two given indices. Verification of sample code that implements a Queue Rotate operation generates the VC below, a proof of which involves theorems from strings and numbers. Both provers had little difficulty. The proof uses the definition of `Q` in given 1 and theorems such as the following: `(Prt_Btwn(0, 1, (<x> o alpha)) = <x>)`.

```
VC 0_3
Ensures Clause of Rotate: Queue_Rotate_Realiz.rb(3)

Goal(s):
((Q' o <Temp'>) = (Prt_Btwn(1, |Q|, Q) o Prt_Btwn(0, 1, Q)))

Given(s):
1. (Q = (<Temp'> o Q'))
2. Other givens omitted.
```

6 Conclusions

The current trend in automated software verification is to use a combination of theory specific decision procedures in conjunction with a SAT solver to handle the logical structure of the problem; e.g. Z3 [6] and Yices [8]. Like MP, these SMT solvers have components that compute congruence closures and are able to perform quantifier instantiation, often based on heuristics. MP borrows the technique of using proven mathematical units from Isabelle and Coq efforts, and is not restricted to the many-sorted first-order logic of SMT.

Invariably, software specifications will involve newly-conceived theories and notations for ease of explanation and understanding, and there might not be viable decision procedures for proving VCs resulting from implementation of those specifications. Using experimentation with two provers, this paper explains that if supporting mathematics and specifications are suitably engineered, then the resulting VCs can be made sufficiently obvious and a lack of solvers might not be an impediment to automated verification.

7 Acknowledgements

This research has been funded in part by the NSF grants CCF-1161916 and DUE-1022941. Our thanks are due to Bill Ogden and other members of our research group.

A Stack Specification

```
Concept Stack_Template(type Entry; evaluates Max_Depth: Integer);
        uses String_Theory, Integer_Theory;
        requires 1 <= Max_Depth;

    Type Family Stack is modeled by Str(Entry);
        exemplar S;
        constraint |S| <= Max_Depth;
        initialization ensures S = Empty_String;
        end;

    Operation Push(alters E: Entry; updates S: Stack);
        requires 1 + |S| <= Max_Depth;
        ensures  S = <#E> o #S;

    Operation Pop(replaces R: Entry; updates S: Stack);
        requires 1 <= |S|;
        ensures #S = <R> o S;

    Operation Depth(restores S: Stack): Integer;
        ensures Depth = (|S|);
end;
```

References

1. Adcock, B.M.: Working towards the verified software process. Ph.D. thesis, The Ohio State University (2010)
2. Barras, B., Boutin, S., et al.: The coq proof assistant reference manual: Version 6.1 (1997)
3. Barrett, C., Stump, A., Tinelli, C.: The smt-lib standard version 2.0. SMT-LIB.org (2010), `http://SMT-LIB.org`
4. Beckert, B., Hähnle, R., Schmitt, P.H.: Verification of object-oriented software: The KeY approach. Springer-Verlag (2007)
5. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: An smt-lib format for sequences and regular expressions. In: Strings. p. 24 (2012)
6. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS 2008, pp. 337–340. Springer (2008)
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. Journal of the ACM (JACM) 52(3), 365–473 (2005)
8. Dutertre, B.: Yices 2.2. In: Computer Aided Verification. pp. 737–744. Springer (2014)
9. Harton, H.: Mechanical and Modular Verification Condition Generation For Object-Based Software. Ph.D. thesis, Clemson University (2011)
10. Hoare, T.: The verifying compiler: A grand challenge for computing research. In: Modular Programming Languages, pp. 25–35. Springer (2003)
11. Kirschenbaum, J., Adcock, B., Bronish, D., Smith, H., Harton, H., Sitaraman, M., Weide, B.W.: Verifying component-based software: Deep mathematics or simple bookkeeping? In: Proceedings of the 11th ICSR. pp. 31–40. Springer-Verlag (2009)
12. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR 2010. pp. 348–370. Springer (2010)
13. Muller, P., Shankar, N., Leavens, G.T., et al.: The 1st verified software competition, extended experience report (2011)
14. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. Journal of the ACM (JACM) 27(2), 356–364 (1980)
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
16. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using smt. In: Computer Aided Verification, pp. 773–789. Springer Berlin Heidelberg (2013)
17. Sitaraman, M., Adcock, B., et al.: Building a push-button resolve verifier: Progress and challenges. Formal Aspects of Computing 23(5), 607–626 (2011)
18. Smith, H.: Engineering Specifications and Mathematics for Verified Software. Ph.D. thesis, Clemson University (2013)
19. Weide, B.W., Sitaraman, M., et al.: Incremental benchmarks for software verification tools and techniques. In: Proceedings of VSTTE 2008. pp. 84–98. Springer (2008)
20. Wies, T., Muñiz, M., Kuncak, V.: An efficient decision procedure for imperative tree data structures. In: Automated Deduction–CADE-23, pp. 476–491. Springer (2011)