ECOLE POLYTECHNIQUE
PROMOTION 2007
DAN Andrei Marian

RAPPORT DE STAGE DE RECHERCHE

Kernel of Truth based SAT proof checker

NON CONFIDENTIEL

Option: INFORMATIQUE
Champ de l'option: Qualite du logiciel
Directeur de l'option: Olivier Bournez
Directeur de stage: Natarajan Shankar
Dated du stage: 05/04/2010 - 27/08/2010
Adresse de l'organisme:
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
USA

Abstract

SAT solvers can generate proofs that justify their final result. This report presents a verified program that checks if the generated proofs are valid. The checker is written and proved in PVS. In order to execute it, Lisp code is automatically generated. We describe a trusted kernel called the Kernel of Truth. Finally, we prove that it exists a valid kernel proof corresponding to any valid checker proof. This demonstrates that the checker is a safe extension of the kernel.

Contents

1 Introduction

Before starting, I would like to thank Dr. Natarajan Shankar for giving me the chance to work on this project and for helping me throughout all my internship. I also had great help from Sam Owre and from Dr. Bruno Dutertre.
The correctness of SAT solver programs can be achieved by several approaches. One possibility is to specify the solver using a proof asistant and to prove its correctness. In this scenario, the software that needs to be trusted contains the proof assistant and the compiler from specification language to programming language. The approach presented here is based on a trusted kernel called Kernel of Truth (KoT). KoT contains first-order deduction rules that can be composed to create more complex proofs. The proof generated by the solver has an equivalent kernel proof in KoT logic. The advantage of this approach is that we must only trust the KoT rules and the program that checks a KoT proof, instead of the entire proof assistant.

Modern implementations of DPLL using clause learning are difficult to formalise and verify using proof assistants. This imposes the compromise between formalisation and performance. To avoid this choice, S accepts his uncertain status and generates proofs (noted Pr) that justifies his answers. The Pr for a formula can have the following forms:

1. An assignment for all the variables that makes the formula is satisfiable
2. A list of clauses and resolution steps that resolve to the empty clause (other forms?)

2 Overview of PVS

PVS (Prototype Verification System) is an environment for specification and proving. The main purpose of PVS is to provide formal support for conceptualization and debugging in the early stages of the lifecycle of the design of hardware or software systems. In these stages, both the requirements and designs are expressed in abstract terms that are not necessarily executable. The best way to analyze such an abstract specification is by attempting proofs of desirable consequences of the specification. Subtle errors revealed by trying to prove the properties are costly to detect and correct at later stages of the design lifecycle.
The specification language of PVS is built on higher-order logic (functions can be treated like primitive types: functions can take functions as arguments and return them as values, quantification can be applied to function variables. Specifications can be constructed using definitions and axioms [4].

2.1 The first proof

Types declaration is done by declaring type constructors with associated parameters:

```
list [T: TYPE]: DATATYPE
BEGIN
 null: null?
 cons (car: T, cdr:list):cons?
END list
```

Functions require the type specification for all the parameters and for the return value. Recursive functions need a measure that decreases with each call. For instance:

```
  member(x, l): RECURSIVE bool =
   CASES l OF
     null: FALSE,
     cons(hd, tl): x = hd OR member(x, tl)
   ENDCASES
  MEASURE length(l)
```

Properties of the function's behavior can be proven using theorems or lemmas:

```
member_null: LEMMA member(x, l) IMPLIES NOT null?(l)
```

This lemma states that if `x` is a member of the list `l`, then the list is not empty. When we try to prove this lemma, the proving environment displays a sequent. For this case, the sequent is the body of the lemma, and universal quantifiers are added for the variables `x, l`.

```
  |-------
{1}   FORALL (l: list[T], x: T):
member(x, l) IMPLIES NOT null?(l)
```

The lines above `-------` are the known elements, and below are the objectives. It is considered that the known elements are in conjunction and the objectives are in disjunction. In this case there are no hypothesis. We explain step by step the proof because proving properties was an important part of the work. We apply proof commands that modify the objective using the hypothesis. In the example we use `(skosimp)` that does skolemization, eliminating the universal quantifiers, and simplifies the resulting formula. We obtain:

```
{-1}  member(x!1, l!1)
{-2}  null?(l!1)
  |-------
```

This means that there are now two hypothesis and no conclusion, meaning that there is a conflict between the two formulas above the line. The skolemization transformed the variables `x, l` in `x!1, l!1`. The simplification that followed the skolemization eliminated `IMPLIES` by moving `member(x!1,l!1)` above the line. Also, the `NOT null?(l!1)` formula below the line was transformed in `null?(l!1)` above the line.
To continue the proof, we expand the definition of the member function by using the command `(expand member)`. We obtain:

```
 {-1}  CASES l!1 OF
  null: FALSE,
  cons(hd, tl): x!1 = hd OR member(x!1, tl)
        ENDCASES
[-2]  null?(l!1)
  |-------
```

The number of the formula can be in `{}`, which means that it was modified by the previous command, or in `[]`. This corresponds to the function's definition above. We use `(lift-if)` to replace the `CASES` form with an `IF`.

```
 {-1}  IF null?(l!1) THEN FALSE
       ELSE x!1 = car(l!1) OR member(x!1, cdr(l!1))
       ENDIF
[-2]  null?(l!1)
  |-------
```

In this situation we can use the hypothesis `[-2]` to eliminate the `IF` from `{-1}` by `(replace -2 -1)`. The proof simplifies to:

```
[-1]  null?(l!1)
  |-------
{1}   TRUE
```

`FALSE` in the hypothesis becomes `TRUE` in the conclusion. The prover automatically gives the following message:

```
 which is trivially true.
Q.E.D.
```

2.2 TCCs

The PVS typechecker analyzes the specification for semantic consistency. The type system of PVS is not algorithmically decidable. Theorem proving by the user might be required to establish the type-consistency of a PVS specification. These theorems are called type-correctness conditions (TCCs) [2]. For instance, the following function:

```
append(l1, l2): RECURSIVE list[T] =
 CASES l1 OF
   null: l2,
   cons(x, y): cons(x, append(y, l2))
 ENDCASES
 MEASURE length(l1)
```

generates the termination TCC:

```
append_TCC1: OBLIGATION
 FORALL (l1: list[T], x: T, y: list[T]):
   l1 = cons(x, y) IMPLIES length(y) < length(l1);
```

This TCC is created because the `append` function declares that the length of the first argument should decrease in the recursive call.

2.3 Code generation

PVS, like other specification languages, is designed to be expressive rather than executable. However, in order to test and validate models, executing specifications is useful. This is achieved by generating Lisp code from PVS. The fragment of PVS that can be executed can be viewed as a high-order functional programming language. Examples of specification elements that are not executable are: free variables, quantifications over infinite domains, terms with uninterpreted function symbols, equalities between terms of higher type [3].

3 Presenting the Kernel of Truth

> The best way to find out if you can trust somebody is to trust them. (Ernest Hemingway)

This text presents the implementation of the concepts presented in [5]. The KoT is written in PVS.

3.1 Formulas

A term is defined reccursively as a variable or an application of a function to a list of terms. The variables are identified by an index. The functions are defined by two natural numbers, an index and the arity. The length of the list of arguments should be equal to the function's arity.

```
term  : DATATYPE
 BEGIN
 v(v_index: nat): var?
 apply(fun: (fun?),
       args: {ss: list[term] |
length(ss) = arity(fun)}): apply?
END term
```

The distinction is made between interpreted or uninterpreted predicates and functions.

```
funpred: DATATYPE WITH SUBTYPES pred?, fun?
 BEGIN
  ipred(index, arity: nat): ipred? : pred?
  upred(index, arity: nat): upred? : pred?
  ifun(index, arity: nat): ifun?   : fun?
  ufun(index, arity: nat): ufun?   : fun?
 END funpred
```

A formula is either a predicate applied to its arguments, a negation, a disjunction or an existential quantification. The universal quantifier and the conjunction can be composed using the previous functions.

```
fmla: DATATYPE
 BEGIN
  atom(pred: (pred?), args: {ss: list[term] |
     length(ss) = arity(pred)}): atom?
  f_not(arg: fmla): f_not?
  f_or(arg1, arg2: fmla): f_or?
  f_exists(bvar: (var?), body: fmla): f_exists?
 END fmla
```

The substitution of a variable in a term with another term, term equality, parameter substitution, free variables search are also implemented in the kernel. A sentence is a formula that doesn't have any free variables (all variables are bound by the existential quantificator).

```
A, B, C: VAR fmla
sentence?(A): bool = null?(freevars(A))
```

KoT uses a list of sentences to represent one-sided sequents, which are used to build the rules.
The equality between two terms uses a predefined predicate:

```
f_eq(s, t): fmla = atom(ipred(0, 2), (:s, t :))
```

The equality predicate has the index 0, the argument 2 representing the number of parameters.

3.2 Rules

- The first induction rule defined in KoT is the axiom:

$$\frac{}{\vdash A, \neg A, \ldots} \tag{1}$$

- Subset rule:

$$\frac{\vdash A_1, \ldots, A_n}{\vdash B_1, \ldots B_m}, A_1, \ldots, A_n \subset B_1, \ldots B_m \tag{2}$$

- Or rule:

$$\frac{\vdash A, B}{\vdash A \bigvee B} \tag{3}$$

- Not-or rule:

$$\frac{\vdash \neg A \quad \vdash \neg B}{\vdash \neg(A \bigvee B)} \tag{4}$$

- Negation rule:

$$\frac{\vdash A}{\vdash \neg\neg A} \tag{5}$$

- Cut rule:

$$\frac{\vdash A, A_1, \ldots, A_n \quad \vdash \neg A, B_1, \ldots B_m}{\vdash A_1, \ldots, A_n, B_1, \ldots B_m} \tag{6}$$

- Exists rule:

$$\frac{\vdash [s/x]A}{\vdash \exists x : A} \quad , freevars(s) = \emptyset \tag{7}$$

- Equivalent forall rule:

$$\frac{\vdash \neg[f/x]A}{\vdash \neg\exists x : A} \quad , \neg f \in freesymbols(A) \tag{8}$$

- Axiom schema instantiation rule, appliable for function and predicate instantiation.
- The reflexivity rule, where the equality is the predicate described above:

$$\frac{}{\vdash s = s} \tag{9}$$

- Function congruence rule:

$$\frac{}{\vdash \neg(a1 = b1), \ldots, f(a1, \ldots) = f(b1, \ldots)} \tag{10}$$

- Predicate congruence rule:

$$\frac{}{\vdash \neg(a1 = b1), \ldots, \neg p(b1, \ldots), p(a1, \ldots)} \tag{11}$$

4 The trace checker

SAT solvers take as an input a logic formula and return a boolean value. To justify their output, SAT solvers can also return a proof. If the formula is satisfiable, the proof is an assignment for each variable that make the formula true. If the formula is not satisfiable, then the solver provides a list of resolution steps that starts with the initial clauses and leads to the empty clause [6]. This is the proof that the initial clauses lead to a conflict, so a satisfying assignment does not exist. The trace checker presented in this text verifies that the list of resolution steps actually leads to the empty clause. The checker can generate a tree proof containing only KoT rules for each list of resolution steps justifying the resulted clause.

4.1 Input data

The Picosat solver generates a trace (tP) to justify the result. The line input format for the trace checker (tC) is similar to tP:

1. $$< lineNumber >< literalsList > s \;\; s, \tag{12}$$

   where $lineNumber \geq 0$, $literalsList$ is a list of integers representing the normal or negated form of the variables in the initial CNF formula. s has the role of separator.

   For instance:

```
18 -40 -55 90 s s
```

is the 19th line of the file and represents the clause

```
(-40 -55 90)
```

2. $$< lineNumber > * < lineNumbers > s, \tag{13}$$

   where $lineNumbers$ is a list of naturals representing the lines of the clauses on which resolution must be applied. Each element of $literalsList$ is smaller than $lineNumber$, meaning that only previous clauses can be used. An example is

   ```
   57 * 32 29 46 45 56 s,
   ```

   a line that requires resolution to be applied to the clauses obtained at the lines

   ```
   32, 29, 46, 45, 56
   ```

The differences between tC and tP are:

1. In tP the separator is 0, and the line numbers begin at 1.
2. In tP the line numbers can have gaps, consecutive lines can be numbered n and $n + 1 + \delta$, where $\delta > 0$. In tC the gaps are eliminated, modifying the $lineNumber$, and its appearances in the $lineNumbers$ lists.
3. tP presents a general type of line, that contains both $literalsList$ and $lineNumbers$. In this case, the clause that should be obtained by resolution on the list of clauses is given. The role of the trace checker would be only to verify that this clause and the computed clause are the same. This type of line was not generated by Picosat during the tests and it is not treated by tC.
4. In tP, the order of $lineNumbers$ is not necesarly the correct one to obtain at the end the empty clause. In tC, the order in which the resolutions are applied is the order indicated by $lineNumbers$.

The transformation from tP to tC is done by a Lisp program and uses a slightly modified Picosat solver that generates the $lineNumbers$ in the correct order.
The PVSio library [1] was used as an interface between file-input operations and the PVS structures.

4.2 Checker structure

The literals of the logic expression are KoT formulas: uninterpreted predicates of arity 0. `prop_atom?` represents a variable, and literals are variables or negated variables:

```
prop_atom?(A): bool = (atom?(A) AND
                       upred?(pred(A)) AND
  arity(pred(A)) = 0 AND
  index(pred(A)) > 0)

literal?(A): bool = (prop_atom?(A) OR
             (f_not?(A) AND prop_atom?(arg(A))))
```

The clause type is defined by a predicate that describes a strict sorted list of literals. The lists are sorted to accelerate the resolution step and to avoid duplicate literals in a clause.

```
lAA : VAR list[(literal?)]
clause?(lAA) : bool = sorted?(lAA)
```

The function `nclause2fmla` takes a non-empty clause and returns an equivalent formula built as a disjunction of the literals in the clause.

```
    nclause2fmla(nclAA): RECURSIVE fmla=
     IF null?(cdr(nclAA))
     THEN car(nclAA)
     ELSE f_or(car(nclAA), nclause2fmla(cdr(nclAA)))
     ENDIF
   MEASURE length(nclAA)
```

The function `not_or_reduction` transforms a non-empty clause into a sequent(a list of formulas with no free variables) containing the negated formula returned by `nclause2fmla`:

```
not_or_reduction(nclAA): sequent =
     cons(f_not(nclause2fmla(nclAA)), null)
```

The previous function will be used in the equivalence theorems presented in the next section. The resolution is based on a pivot, a variable that is found in a clause and its negation in the other clause. The pivot is not specified in the trace that is being verified, so the checker must find it. The function that verifies if a pivot exists takes as arguments the two clauses:

```
exist_pivot?(ck, cl) : RECURSIVE bool =
   CASES ck OF
     null: FALSE,
     cons(k, cm): member(tr_neg(k), cl)
   OR exist_pivot?(cm, cl)
   ENDCASES
 MEASURE length(ck)
```

The signature of the function that returns the pivot, once it is established that it exists is:

```
find_pivot(nck, (ncl:{ncm | exist_pivot?(nck, ncm)})) :
  RECURSIVE {k | member(k, nck) AND member(tr_neg(k), ncl)}
```

where `ck, cl` are clauses and `nck, ncl, ncm` are non-empty clauses. Using the sorted list representation for the clauses, the resolution step becomes a merge of the two clauses, followed by removing the pivot. The checker allows the situation where a pivot doesn't exist. In this case, the resolution result is only the merged list. The function `tr_clause_true?` returns true if a clause contains a literal and its negation. The resolution function is:

```
resolution(nck, ncl) : (tr_clause?) =
   IF tr_clause_true?(nck) THEN ncl
   ELSIF tr_clause_true?(ncl) THEN nck
   ELSE
    LET merged = merge(nck, ncl) IN
    IF exist_pivot?(nck, ncl) THEN
      LET pivot= find_pivot(nck, ncl) IN
      delete_pivot(merged, pivot)
    ELSE
      merged
    ENDIF
   ENDIF
```

The `resolution_list` function is equivalent to a fold applied on the list of clauses with the `resolution` function.
For each line of the certificate, in succesive order, there is either no change (the case of input clauses) or `resolution_list` is applied, in order to calculate the line's clause. A certificate is correct is at the final line the empty clause is obtained.

4.3 Kernel of Truth equivalent proof

The type `rule` corresponds to the induction rules presented above. Each rule has a constructor associated. The proofs are stored as a finite sequence of proof steps:

```
proof_step : TYPE
  = [# sequent: sequent,
       rule: rule,
subs: list[nat] #]

proof_seq: TYPE =  finseq[proof_step]
```

The `sequent` is the conclusion of the `proof step`. The `rule` is used to prove the `sequent`. `subs` is a list of indices, each element of the list pointing to an element in the proof that is used as a premise for the `rule` that is applied.
The conclusion of a proof is the `sequent` of the last proof step in the list:

```
RR: VAR proof_seq
ne_proof_seq: TYPE = {RR | RR`length > 0}
nRR: VAR ne_proof_seq

conclusion(nRR): sequent =
   (nRR`seq(nRR`length - 1))`sequent
```

An important function of KoT is `checkProof` which takes as input a finite sequence of lemmas that can be used and the proof. The function checks the correctness of each element of the proof. The signature of this function is:

```
checkProof(lemmas: finseq[sequent])(RR): bool
```

The objective is to build proofs for a resolution step and then for a chain of such steps that are validated by `checkProof`.
Auxiliary functions are used to build proofs. Let's describe the function that composes the proof of $A :\vdash (\neg a), \Delta$ with the proof of $B :\vdash (\neg b), \Delta$ to obtain a proof of $C :\vdash \neg(a \vee b), \Delta$. If the proof of A is a sequence of n proof steps and the proof of B has length m, then the proof of C will have the length $n+m+1$. The first $n+m$ steps are the concatenation of the first two proofs, and the last step is a $not - or$ rule. The function that builds the last proof step is:

```
concl_norr(sA, sB, sAA, (n, m: posnat)): proof_step =
         (# sequent:= cons(f_not(f_or(sA, sB)), sAA) ,
```

```
          rule:= norr,
     subs:= cons(n - 1, cons(n + m - 1,null)) #)
```

`sA` and `sB` are a and b from the formulas above, `sAA` is Δ, and `m,n` are the lengths of the proofs. `subs` is
The proof is built in PVS by:

```
RR_norr(sA, sB, sAA, RR1, RR2): ne_proof_seq =
   LET n = RR1`length IN
   LET m = RR2`length IN
   (# length := n + m + 1,
      seq := LAMBDA(j: below(n + m + 1)):
                COND
  j < n -> RR1`seq(j),
  j >=n AND j < n + m -> offset_proof_step(RR2`seq(j - n), n),
  j = n + m -> concl_norr(sA, sB, sAA, n, m)
 ENDCOND #)
```

`RR1, RR2` are the two finite sequences representing the proofs of A and B. The resulting sequence is a function $[0 \dots n+m] \to$ `proof_step`, identical with `RR1` on $[0 \dots n-1]$, with `RR2` on $[n \dots n+m-1]$, and in $n+m$ the value is the result of `concl_norr`.
The PVS lemma resuming all the description above is:

```
lemma_norr: LEMMA (conclusion(RR1) = cons(f_not(sA), sAA) AND
       conclusion(RR2) = cons(f_not(sB), sAA) AND
       checkProof(empty_seq)(RR1) AND
       checkProof(empty_seq)(RR2))
      =>
      (conclusion(RR_norr(sA, sB, sAA, RR1, RR2)) =
cons(f_not(f_or(sA, sB)), sAA) AND
       checkProof(empty_seq)(RR_norr(sA, sB, sAA, RR1, RR2)))
```

This shows that if the proofs `RR1, RR2` are validated by `checkProof`, then the proof returned by `RR_norr` will also be validated and it will have the required conclusion.

4.3.1 Resolution proof

First, a theorem about single step resolution is proved:

```
th: THEOREM
       conclusion(proof_th(ntcA, ntcB)) =
```

```
 append(
  not_or_reduction(translate_clause(ntcA)),
  append(
      not_or_reduction(translate_clause(ntcB)),
    translate_clause(resolution(ntcA, ntcB))))
AND checkProof(empty_seq)(proof_th(ntcA, ntcB))
```

The function `proof_th` builds a proof that has the following structure:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \neg p, \neg\neg p}\,axiom \quad \dfrac{}{\vdash \neg\Delta, \Delta}\,axiom}{\vdash \neg p, \neg(\neg p \bigvee \Delta), \Delta} \quad \dfrac{}{\vdash \neg\Gamma, \Gamma}\,axiom}{\vdash \neg(p \bigvee \Gamma), \neg(\neg p \bigvee \Delta), \Gamma, \Delta}}{\vdash \neg(p \bigvee \Gamma), \neg(\neg p \bigvee \Delta), \Gamma \bigvee \Delta} \tag{14}$$

4.3.2 Chain resolution proof

We use several auxiliary functions that operate on sequents. `not_or_map` takes a non-empty clause as an argument and returns the sequent containing the negated literals of the clause.
The variable `lntcA` is a non-empty list of clauses. Since the checker must resolve all the clauses in a chain to obtain the resolvent, we build a theorem to prove the KoT equivalence for chain resolution. Similar to the resolution step, we prove:

```
   th_list:  THEOREM
   LET result: (tr_clause?) =
  resolution_list(lntcA) IN
        conclusion(proof_th_list(lntcA)) =
 append(
   not_or_map(lntcA),
   translate_clause(result))
AND checkProof(empty_seq)(proof_th_list(lntcA))
```

The variable `result` contains the resolvent obtained by resolution on the list of clauses. The function `proof_th_list` builds the proof that concludes that the chain resolution on a list of clauses is implied by the clauses.

5 Examples and tests

Picosat is the SAT solver used for proof generation. A modified version of Booleforce takes a proof generated by Picosat and reorders the antecedents of the learned clauses such that chain resolution can be applied.

5.1 A small example

We start with a simple unsatisfiable CNF formula:

```
p cnf 2 4
1 2 0
-1 2 0
1 -2 0
-1 -2 0
```

This file corresponds to $(x \bigvee y) \bigwedge (\neg x \bigvee y) \bigwedge (x \bigvee \neg y) \bigwedge (\neg x \bigvee \neg y)$ The solver calculates that the formula is unsat and produces a trace to prove this result. After the processing, the proof becomes:

```
0 1 2 s s
1 -1 2 s s
2 1 -2 s s
3 -1 -2 s s
4 * 1 3 s
5 * 4 2 s
6 * 4 0 5 s
```

This can be easily checked by hand. The first number of the line represents the clause identifier. The first four lines are the initial clauses. The clause associated with the fifth line is obtained by resolving clauses `1` and `3`, thus obtaining `(-1)`. The sixth clause is `|-2|`.For the last line, we resolve clause 4 with 0, and the result with clause 5. The result is the empty clause, so this is a valid unsatisfiability proof.

5.2 Complexity Analysis

The checker has a linear complexity $O(n)$, where n is the number of resolution steps specified by the trace. The complexity constant that multiplies n is the source of possible optimisations. Since the code generated from PVS comes from a specification, there is a compromise between the efficiency of the code and the ease to prove the required properties. Since both PVS and Lisp are functional languages, an important optimisation is tail-recursive functions. We should make sure that tail-recursive functions specified in PVS are translated in Lisp functions with the same property. Another method to improve the speed of the checker is to build a faster version of the resolution function and then prove only the equivalence with the already proved function. We used this method to accelerate the checking process.

Size of the trace (KB)	Time to check (s)
1.2	0.003
1.6	0.015
76	4.7
449	20
963	74
1800	132
15300	1006
37800	1863

5.3 Tests

We used CNF formulas from the SatLib benchmarks problems and other sources. The results are presented in the table below:

The current version that produced these results is not optimised for efficient resolution steps, the accent being on the clear specification and ease of proving. A new version, without fundamental modifications, could reduce times because currently the clauses are parsed several times in order do perform resolution, whereas a single parse could be enough.

6 Conclusion

In this report we present the structure of a trusted kernel (KoT) and the implementation of a cerfified SAT trace checker. We present the proof that it exists a valid kernel proof corresponding to any valid checker proof. This project required writing a preprocessor for the traces in Lisp, proving the checker in PVS and using a PVS extension, PVSio, to be able to write imperative code when needed. It is possible to continue the extension of the kernel by building verified checkers for SMT and rewrite traces.

References

[1] César Muñoz. Rapid prototyping in PVS. Contractor Report NASA/CR-2003-212418, NASA, Langley Research Center, Hampton VA 23681-2199, USA, May 2003.

[2] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[3] Shankar and Natarajan Shankar. Efficiently executing pvs. Technical report, Project report, ComputerScience Laboratory, SRI International, Menlo Park, 1999.

[4] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[5] Natarajan Shankar. Rewriting, inference and proof. International Workshop on Rewriting Logic and its Applications, 2010.

[6] Lintao Zhang and Sharad Malik. Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10880, Washington, DC, USA, 2003. IEEE Computer Society.