Validity Checking for Combinations of Theories with Equality

Clark Barrett, David Dill, and Jeremy Levitt

Computer Systems Laboratory, Stanford University, Stanford CA 94305, USA

Abstract. An essential component in many verification methods is a fast decision procedure for validating logical expressions. This paper presents the algorithm used in the Stanford Validity Checker (SVC) which has been used to aid several realistic hardware verification efforts. The logic for this decision procedure includes Boolean and uninterpreted functions and linear arithmetic. We have also successfully incorporated other interpreted functions, such as array operations and linear inequalities. The primary techniques which allow a complete and efficient implementation are expression sharing, heuristic rewriting, and congruence closure with interpreted functions. We discuss these techniques and present the results of initial experiments in which SVC is used as a decision procedure in PVS, resulting in dramatic speed-ups.

1 Introduction

Decision procedures are emerging as a central component of formal verification systems. Such a procedure can be included as a component of a general-purpose theorem-prover [8, 11], or as part of a domain-specific verification system [1]. Domain-specific systems for formally verifying the correctness of hardware designs in particular are becoming increasingly important, and they are beginning to be incorporated into industrial electronic design automation tools. As part of a general design tool used by non-experts, it is important that a verification system be automatic. To be useful in practice, it is important that the system be flexible and efficient.

Complex hardware designs are hierarchical; a decision procedure to verify such a design needs to have the flexibility to abstract away lower levels of the hierarchy. While BDDs have been applied to the verification of hardware, their use has been limited to certain application domains. Although automatic and very efficient for certain problems, they do not easily support abstraction. Furthermore, they have difficultly dealing with common operations, such as multiplication. Other procedures use a system of rewrites based on Knuth-Bendix completion to attempt to prove the validity of a formula [7, 3]. However, these systems are limited and often inefficient.

For several years, we have been developing a decision procedure for use in hardware verification. The data structure we use resembles a BDD in that expressions are represented as directed acyclic graphs (DAGs) and reused whenever possible. However, by extending our domain from Boolean formulas to decidable fragments of first-order logic, we obtain a procedure which is well suited for the verification of hardware. We maintain the benefits of automation and efficiency, while overcoming the drawbacks of BDDs. Lower levels of a design hierarchy can be easily abstracted through the use of uninterpreted functions, while linear arithmetic, inequalities and other functions and relations can be naturally represented and reasoned about.

Furthermore, rather than relying on Boolean case-splitting or syntactic rewriting alone, we use a hybrid approach: we begin with a formula α which is an arbitrary Boolean combination of atomic formulas. As in previous procedures [6], we maintain a data structure (called a *context*), which is a database containing a set of logical expressions partitioned into equivalence classes. The initial context contains α and all of its subexpressions, each in its own equivalence class. The procedure begins by extracting an atomic formula β from α; it then asserts β and simplifies α in the new context (i.e. substitutes **true** for β). During this simplification, a set of heuristic rewrites are applied which reduce the size of the DAG and increase the overall performance significantly. The procedure then recursively checks the simplified formula for validity. If the recursive call concludes that the formula is not valid, the procedure halts and produces a counterexample. Otherwise, the procedure restores the pre-assertion context, denies β (i.e. substitutes **false** for β), simplifies α in the new context, and then checks the simplified formula recursively as before.

To guide the simplification process and allow the integration of rewrite rules, we define a total ordering on expressions. Intuitively, the expression ordering determines which of two expressions is the *simplest*. Given any atomic formula, we can quickly find the simplest equivalent expression in the current context. This ordering is *monotonic* with respect to term structure so that simplification of an expression using substitution and rewriting can proceed from the bottom up, substituting the simplest equivalent formula for each sub-formula.

One of the problems that a context must deal with is detecting inconsistent combinations of atomic formulas, such as $x = y$ and $f(x) \neq f(y)$. *Congruence closure* is a well-known method for dealing with such phenomena [4, 12, 9]. Although it is true as was reported in [6] that congruence closure is often unnecessary, we have found practical applications where congruence closure is required. The addition of interpreted functions complicates the congruence closure algorithm. Nelson and Oppen propose a solution in which each theory containing interpreted functions has a separate decision procedure. These procedures communicate by propagating information about equality. Several systems have decision procedures based on Nelson and Oppen's methods for congruence closure [9], including STeP [8], and the Extended Static Checker (ESC) being developed by Nelson and Detlefs et al. at DEC [10]. Our implementation most closely resembles that of Shostak [13, 2]. It provides tightly-coupled decision procedures which interact through a single congruence closure algorithm and is thus more efficient than the scheme of Nelson and Oppen.

The entire system is coded in C++ with due attention to low-level efficiency issues. Both the logic and the procedure are significantly extended from that described by Jones, Dill and Burch in 1995 [6]. Despite the addition of congruence closure, linear arithmetic and other extensions, the new system is as fast or faster than the previous version, depending on the examples used.

The remaining sections are organized as follows. We first give some background and present the basic algorithm in Section 2. Section 3 describes the ordering on expressions. Section 4 discusses our implementation of congruence closure, and Section 5 compares it with Shostak's method. Section 6 discusses extensibility, and Section 7 presents results and conclusions. The Appendix contains some of the more involved proofs.

2 Background

2.1 The Logic

Our logic has the following abstract syntax:

expr	::=	*const*
	|	*function symbol*(*expr*, ..., *expr*)
	|	**ite**(*expr*, *expr*, *expr*)
	|	(*expr* = *expr*)
	|	**add**(*rational const*,(*rational const*, *expr*),...,(*rational const*, *expr*))
const	::=	**false**
	|	**true**
	|	*rational constant*
	|	@*symbol*

Expressions are classified by *sort*: constants, uninterpreted functions, if-then-else expressions (**ite**s), equalities and **add** expressions. The addition of other interpreted functions such as array operations and linear inequalities is discussed in Section 6. Constants include the Boolean constants **true** and **false**, rational constants, and special user-defined constants distinguished by an initial "@". By definition, no two constants can be equal. Variables are uninterpreted functions of arity 0. Boolean expressions are represented in terms of **ite**s.

The syntax defines an abstract syntax tree; in the remainder of the paper whenever we refer to an expression, we are referring to its abstract syntax tree. If α and β are two expressions, we say that α is a subexpression of β if α is a subtree of β. We say that α is a child of β if α is a subtree rooted at a child of the root of β. In this case, we also say that β is a parent of α. The depth of an expression is defined to be the height of its tree. Formally,

Definition 1 (Depth) *The depth of an expression α, $D(\alpha)$, is defined recursively as: if α has no children, $D(\alpha) = 0$. Otherwise, $D(\alpha) = max\{D(c) \mid c \text{ is a child of } \alpha\} + 1$.*

Since **add** expressions are interpreted, they behave a little bit differently. We define the depth of an **add** expression to be the depth of its least simple child. In effect, the least simple child represents the **add** expression when it is compared with other sorts of expressions. More will be said about this in Section 3. It is also necessary to note that the child of an **add** expression in normal form cannot be an **add** expression itself.

The following lemma is easily shown by induction on $D(\beta)$.

Lemma 1. *If α is a subexpression of β and β is not an* **add** *expression, then $D(\alpha) < D(\beta)$.*

An important concept is that of an *atomic* expression. An atomic expression is defined as an expression containing no **ite** subexpressions.[1]

As was previously stated, expressions are represented by DAGs. Each expression has an entry in a hash table. This ensures that there is one unique copy for every expression and enables sharing of subexpressions in the DAG. Before inserting an expression into the hash table, certain rewrite rules are applied. An example of such a rewrite rule is:

$$\mathbf{ite}(\alpha, \alpha, \beta) \;\rightarrow\; \mathbf{ite}(\alpha, \mathbf{true}, \beta).$$

As mentioned above, a context consists of a set of expressions partitioned into equivalence classes. These classes are maintained using Tarjan's *union* and *find* primitives [2] [14]. Each equivalence class contains a unique representative element called the *equivalence class representative* which is an expression used to represent the class. If expressions α and β are in the same equivalence class in context C, we say that $\alpha \sim_C \beta$.

2.2 Validation Algorithm

A simple validation algorithm (based on the one used in [6]) is shown in Figure 1. Given an expression, we pick an atomic expression (called a *splitter*) that appears as the if-part of an **ite** subexpression and perform a case split. We repeat this recursively, reducing the **ite** to its then-part or else-part on each case

[1] Ideally, an atomic expression would be an expression containing no Boolean subexpressions. However, since we do not store explicit type information we can only identify a Boolean expression if it appears in the if-part of an **ite**. Though this theoretically limits the completeness of our algorithm, we have not found it to be a problem for practical applications. The issue can be resolved fairly easily by adding explicit typing or more sophisticated type inference.

[2] In our implementation, (as in [13]) *union*(a, b) is deterministic about which find pointer gets changed; in our case it always sets *find*(b) equal to a. This is different from Tarjan's *union* which sets the *find* of the tree with smaller rank. Although this decreases the theoretical worst-case performance of *find*, the actual impact is negligible since the program spends very little time in the *union*/*find* code. Also, we do use path-compression (in *find*) which has a much more significant effect than union-by-rank since there are many more calls to *find* than to *union*.

```
Validate(e)
  e := Simp(e);
  splitter := FindSplitter(e);
  IF e = true THEN
    RETURN true;
  IF splitter = NULL THEN
    RETURN false;
  PushContext;
  Assert(splitter);
  result := Validate(e);
  PopContext;
  IF result = true THEN BEGIN
    PushContext;
    Deny(splitter);
    result := Validate(e);
    PopContext;
  END
  RETURN result;

Simp(e)
  e := Signature(e);
  IF e is of the form a = b AND
    (a',b') ∈ diseq list
    where find(a')=a and
          find(b')=b THEN
    RETURN false;
  ELSE
    RETURN find(e);
```

```
Assert(e)
  CASE e OF
    a = b : Merge(a, b);
    ELSE  : Merge(true, e);
  ENDCASES

Deny(e)
  CASE e OF
    a = b : add (a,b) to diseq list
    ELSE  : Merge(false, e);
  ENDCASES

Merge(e1,e2)
  union(e1,e2);

NewExpr(e)
  Perform rewrites on e
  IF e not in hash table THEN BEGIN
    insert e in hash table
    find(e) := e;
  END
  RETURN e;

Signature(e)
  RETURN
    NewExpr(f(Simp($t_1$),...,Simp($t_n$)));
      where e = f($t_1$,...,$t_n$)
```

Fig. 1. Basic Validation Algorithm. Note that **true** indicates the constant expression, whereas true simply refers to the Boolean value.

split. If the expression reduces to **true** when all the splitters are exhausted, we return true, otherwise we return false.

The algorithm presented in this paper is a refinement of the one shown in Figure 1, so we will start by describing this simpler version first. The purpose of `Merge` is to join the equivalence classes of the two atomic expressions passed to it. Merge is always called with the first expression simpler than the second; in particular, equality expressions are always rewritten to ensure that the left-hand side is simpler than the right-hand side. We will discuss what it means for one expression to be simpler than the other in the next section. Recall that *union* makes the first argument the new equivalence class representative for the merged equivalence class. `Simp` recursively traverses an expression, replacing each subexpression with its equivalence class representative. It also detects equalities which are contradictory. This is done by maintaining a list of disequalities which is updated whenever an equality is denied. The significance of the `Signature` function will be discussed in Section 4. `NewExpr` allows for expression sharing by checking if an expression already exists before creating it. One important feature of our implementation is that the function `FindSplitter` is free to choose any atomic Boolean expression, making it easy to add or change splitting heuristics.

The algorithm presented in Figure 1 is sound and complete for Boolean tautology checking but is incomplete for logics which include interpreted and uninterpreted functions. For example, it cannot even validate

$$f(a) \neq f(b) \Rightarrow a \neq b.$$

We will show how to complete the algorithm in Section 4, but first we discuss what it means for one expression to be simpler than another and why this is important.

3 Expression Ordering

We use a set of rules to determine a total ordering $\prec$ on the expressions in our logic. If $\alpha \prec \beta$, we say that α is *simpler* than β. This ordering was designed to have two convenient monotonicity properties. First, it is monotonic with respect to subexpressions. That is, if α is a subexpression of β, then $\alpha \prec \beta$. Second, it is monotonic with respect to substitution, so that if we replace a subexpression with a simpler subexpression, the result is simpler. These properties aid intuition as well as implementation.

As demonstrated by Shostak [13], it is possible to efficiently implement tightly-coupled interacting decision procedures without an ordering on expressions. However, there are several significant benefits to implementing an ordering: without a monotonic ordering, it is possible to have $\mathit{find}(a) = b$ and $\mathit{find}(f(b)) = f(a)$. This kind of behavior is counterintuitive and can increase the difficulty and the complexity of the implementation significantly. On the other hand, monotonicity ensures that whenever we assert that two expressions are equal, the simpler of the two can be substituted for the other throughout the DAG resulting in a more intuitive representation. More importantly, such substitution increases sharing and leads to a more compact representation in the DAG.

Perhaps most importantly, by enforcing that any rewrites which are applied always result in a simpler expression, a monotonic ordering ensures that the heuristic rewrites cannot form an infinite loop. Such rewrites can have a significant impact on performance. On a set of test cases taken from actual verification work, the average speed-up with rewrites is about 2, excluding one large example which has a speed-up of 14.5.

The ordering we use is defined by the following rules (applied in order).

1. Constant expressions are always simpler than non-constant expressions. For arbitrary Boolean, rational and user-defined constants b, r and c, we define $b \prec r \prec c$. For Booleans, we simply have **false** $\prec$ **true**. Rational constants are ordered numerically and user-defined constants are ordered lexicographically.

2. **add** expressions behave like their most complex child when compared with expressions of a different sort (if comparing directly with the most complex child, the child is simpler). When comparing two **add** expressions, their children are compared from most complex to least complex. The first pair of children which are not equivalent determines the ordering.
3. If two expressions have different depths, the one with the smaller depth is simpler.
4. We define uninterpreted functions to be simpler than equalities and equalities to be simpler than **ite**s. If two expressions are of the same sort, they are compared as follows: If the two expressions are uninterpreted functions with different function names, the expression with the lexicographically simpler name is the simplest; otherwise, the children of the expressions are compared in order, and the first pair of children which are not equivalent determine the ordering.

Lemma 2. *Rules 1-4 determine a total order on expressions.*

The proof is omitted, but is straightforward and can be accomplished by case splitting. We state the monotonicity properties in two theorems.

Theorem 3. *If α is a subexpression of β then $\alpha \prec \beta$.*

Theorem 4. *If $\alpha' \prec \alpha$, α is a subexpression of β, and β' is the result of replacing α with α' in β, then $\beta' \prec \beta$.*

The proof of the Theorem 3 is in the appendix, and the proof Theorem 4 is omitted.

4 Congruence Closure

A context C is said to be closed with respect to congruences if the following property holds for all expressions α and β in C:

Property 1 *If α and β are expressions of the same sort with the same number of children, and if each pair of corresponding children are in the same equivalence class, then α and β are in the same equivalence class.*

The notion of a *signature* [4] is helpful in ensuring that this property holds. The signature of an expression e denoted $signature(e)$ is defined to be the expression in which each child has been replaced with its equivalence class representative (notice that this is what the function `Signature` in Figure 1 does). Each expression u maintains a pointer to the expression which is its signature. We denote this expression by $sig(u)$. In order to ensure that Property 1 holds, we must enforce that $sig(u) = signature(u)$. Each equivalence class representative maintains a list (called the use-list) of expressions in which the equivalence class representative appears as a child.

```
Simp(e)
  IF e is atomic THEN
    RETURN find(e);
  ELSE
    RETURN find(Signature(e));

Merge(a, b)
  IF find(a) $\neq$ find(b) THEN BEGIN
    IF IsConst(a) AND IsConst(b) THEN
      Inconsistent := true;
    ELSE BEGIN
      union(a, b);
      FOREACH u IN use(b)
        IF sig(u) = u THEN BEGIN
          sig(u) := Signature(u);
          IF find(u) $\neq$ find(sig(u)) THEN
            IF u = find(u) THEN
              Merge(sig(u), u);
            ELSE
              Assert(NewExpr(find(u) = find(sig(u))));
        END
    END
  END
```

```
Deny(e)
  Merge(false, e);

NewExpr(e)
  Solve, normalize, and rewrite
  IF e is in hash table THEN
    RETURN e;
  ELSE BEGIN
    insert e in hash table
    sig(e) := e;
    find(e) := e;
    use(e) := {};
    FOREACH child c of e DO
      use(c) := use(c) $\cup$ e;
  END
  RETURN e;
```

Fig. 2. Modifications to the original algorithm.

Figure 2 shows the modifications needed to implement congruence closure. It also includes modifications necessary for dealing with **add** expressions. `NewExpr` now puts **add** expressions in a normal form with respect to their children, in addition to performing rewrites. Furthermore, if an equality involves one or two **add** expressions, it is solved so that the most complex child of the **add** expressions appears alone on the right-hand side. This guarantees that when we assert such an equality, the most complex child expression will be replaced with a sum of simpler expressions. Thus, variable elimination occurs automatically. In fact, the interaction between solving and congruence closure is fairly subtle and is one of the reasons that expression ordering was originally introduced; we wanted to guarantee that solving produces a simpler expression.

Whereas the algorithm in Figure 1 is unable to detect inconsistent assertions, detecting inconsistency in the new algorithm is surprisingly simple. A context is inconsistent if and only if there is an equivalence class which contains more than one constant. This is because any other inconsistency will eventually propagate to equating **true** and **false** which are both constants.

The purpose of the additional code in `Merge` is to maintain the following invariant which we show is equivalent to Property 1. Recall that $u \sim_C v$ means that u and v are in the same equivalence class (i.e. $\mathit{find}(u) = \mathit{find}(v)$).

Theorem 5. *A context C satisfies Property 1 if for each expression u, $u \sim_C$ $signature(u)$.*

Proof: Suppose $u \sim_C signature(u)$ for each expression u in C. Let α and β be any two expressions which are of the same sort and for which corresponding children are in the same equivalence class. By definition, then, $signature(\alpha) = signature(\beta)$. But we know that $find(\alpha) = find(signature(\alpha))$ and $find(\beta) = find(signature(\beta)$, so $find(\alpha) = find(\beta)$ and thus $\alpha \sim_C \beta$. □

Since our algorithm creates new signatures as it goes, a concern is whether an infinite number of new signatures could be generated. For example, if we assert $f(x) = x$ and if we make $find(x) = f(x)$, then the signature of $f(x)$ becomes $f(f(x))$. This process could then repeat. Theorem 3 guarantees that this will not happen since it is impossible for the equivalence class representative of an expression to be one of its subexpressions. This is another benefit of monotonicity in our expression ordering.

We claim that with the modifications shown in Figure 2, every pair of expressions in the context will satisfy Property 1 on each call to `Validate`:

Theorem 6. *Each time* `Assert` *or* `Deny` *is called from* `Validate`*, the resulting context is closed with respect to congruences.*

The proof can be found in the appendix.

A fairly significant optimization is to only maintain congruence closure for atomic expressions that are not constants. We can do this because completeness only requires that we know whether an expression is true in a terminating case of `Validate`, and in all such terminating cases the final expression is atomic.

An additional benefit of implementing congruence closure is the ease with which disequalities can be handled. In Figure 1, disequalities between equivalence classes are maintained in a special-purpose disequality table. In the new algorithm, in order to deny $a = b$ (or conceptually, assert $a \neq b$), we simply merge $a = b$ with **false**. Now, if we ever try to equate a and b, the equality will become equal to **true**. As mentioned above, this will result in attempting to merge **true** and **false**, and the inconsistency will be discovered.

5 A Comparison to Shostak's Algorithm

As stated in the introduction, our implementation most closely resembles Shostak's algorithm for congruence closure [13]. Recently, Cyrluk et al. published a complete and rigorous analysis of Shostak's algorithm[2]. For ease of comparison, we have written the code in this paper in a similar fashion. Despite the similarities, there are some significant differences.

First, and perhaps most importantly, Shostak's method requires that expressions be converted to disjunctive normal form. Our algorithm does not require this. Not only does this eliminate the overhead of converting to DNF, but it also gives our technique a tremendous advantage when there is a large amount of sharing in the expression DAG.

Second, we allow the inclusion of heuristic rewrite rules which can significantly improve performance (as discussed earlier).

Third, we implement signatures as actual expressions, rather than as tags associated with expressions. This allows us to compute the congruence closure using only a single loop whereas Shostak's method requires a double loop. The reason for this is as follows. Suppose we merge expressions a and b so that *find*(b) becomes a. We want to guarantee that every expression whose signature contains b is updated and then merged with any other expressions which have the same signature. A single loop over all parents of b is necessary to update all the signatures. In Shostak's implementation, a second loop is required: if u is a parent of b whose signature has changed, then for each such u, all parents of a are checked to see if any have the same signature as u. Since we represent signatures as expressions, the second loop is unnecessary: we simply need to ensure that each expression is in the same equivalence class as its signature. Since the signatures of the parents of a have not changed, there is no need to revisit them. While compared to Shostak's procedure our algorithm generates many extra expressions when a signature changes multiple times, in addition to making the algorithm simpler these extra expressions are actually required in order to return to previous contexts. To avoid updating these old signatures in the congruence closure computation, we check each expression in the use list to see if it is its own signature. If it is not, we know it is an old signature and skip it.

The cost of checking old signatures is also offset by another advantage which comes from using expressions for signatures. Shostak's code requires that the use list of an equivalence class representative contains all expressions which have a child in that equivalence class. We only require that their signatures be in the use list. Thus if there are multiple expressions with the same signature, we have a single entry in the use list whereas an implementation of Shostak's algorithm will have an entry for each expression.

Finally, another difference is the aggressiveness with which we update and simplify expressions. Shostak's implementation waits until an expression is used in an assertion before adding it to use lists; we add expressions to use lists as soon as they are created and we have found it to be faster for many examples. This may be due to the difference between reducing a DAG and reducing a conjunct of formulas.

Other minor differences include the ability to handle disequalities, and support for user-defined constants.

6 Extensions

An advantage of Shostak's method is that his decision procedure easily accommodates new theories with interpreted functions, as long as they are canonizable and algebraically solvable [13]. While our procedure places slightly more stringent requirement on new theories, new theories are similarly easy to accommodate. We require that each new sort of expression be totally ordered and canonizable and that it be possible to solve an equation over interpreted functions for the most complex variable.[3] These are the requirements we meet to support **add** expressions.

A more complicated extension is adding linear inequalities, since the data structures which exist for congruence closure are not sufficient to store all of the implications of asserting an inequality. We solve this problem by adding additional lists at each unasserted inequality which contain a set of expressions implied by the current context. This approach is slow for expressions which are dominated by inequalities, but quite satisfactory for expressions which contain a mix of inequalities, linear arithmetic, Boolean formulas, and array operations.

At the cost of making the procedure incomplete, it is possible to even add interpreted functions which do not satisfy the requirements for canonizability or solvability. A good example of this is the addition of **read** and **write** as new sorts of expressions which implement basic array operations. **read** takes an array and an address and returns the element at that address. **write** takes an array, an address, and a value, and returns the given array with the new value at the specified address. Instead of providing a complete theory for these operations, we treat them as uninterpreted functions and add an automatic rewrite which reduces $\mathbf{read}(\mathbf{write}(s, a1, v), a2)$ to $\mathbf{ite}(a1 = a2, v, \mathbf{read}(s, a2))$.[4] Our algorithm is now incomplete in that it is unable to deal with cases in which **write** expressions are directly equated with other expressions. However, it has been sufficient to deal with most of the verification examples that we have encountered. This demonstrates a further advantage of being able to include rewrites.

7 Results and Conclusions

As a point of comparison, we are experimentally using SVC to assist in proofs done using PVS. Where possible, SVC is used as a decision procedure in place of PVS's internal procedures which are an implementation of Shostak's algorithm. Often, SVC also replaces sequences of Boolean simplifications that are necessary in PVS to put formulas into a disjunctive normal form. Since SVC allows arbitrary Boolean formulas, such simplifications are not required.

So far, our experiments have included proving a fragment of a bounded retransmission protocol verified in PVS by Havelund and Shankar [5] and a simple three stage microprocessor pipeline. The results of running the proofs on a HyperSPARC with 128 MB memory are shown in Table 1 and demonstrate significant speedups even for these small examples.

The speedup in the actual decision procedure is even more significant than shown by the data. In the data for PVS with SVC, the time spent inside of SVC

[3] As in [13], equations involving interpreted functions from more than one theory solve the topmost interpreted functions by treating interpreted functions from other theories as variables.

[4] Adding this rewrite requires a slight adjustment to the expression ordering.

Example	PVS	PVS with SVC
protocol	78.77s	16.11s
pipeline	56.65s	19.71s

Table 1. PVS example with and without SVC decision procedure.

is very small (less than one second) compared to the time PVS spends preparing to send the data to SVC. Of course it is unfair to conclude from these data alone that our algorithm is significantly superior to Shostak's since PVS is coded in LISP and SVC is coded in C++. It does, however, provide some insight into how SVC compares with other verification tools.

We have presented an efficient and flexible algorithm for validity checking with equality and uninterpreted functions. The algorithm improves on previous work by combining the efficiency and speed of [6] with the completeness and extensibility of [13]. SVC has been successfully used both as a brute-force hardware verification tool, and as a fast supplemental decision procedure for more general theorem provers.

Some of the future work we envision is improving the implementation of linear inequalities, increasing the number of interpreted functions in the logic, and developing improved heuristics for choosing splitters. It is interesting to note that determining how to choose splitters is very similar to the problem of BDD variable ordering and so we expect there will be a significant advantage in finding good heuristics.

Acknowledgments

We would like to thank Jens Skakkebæk for his invaluable help in putting together and conducting the experiments with PVS, and for uncovering bugs in SVC. We would also like to thank Robert Jones whose initial work on SVC laid the foundation for our continuing effort and who provided much appreciated feedback as we prepared this paper. This research was supported by ARPA contract DABT63-95-C-0049, and by a National Defense Science and Engineering Graduate Fellowship.

Appendix

Proof of Theorem 3: We first show the following lemma.

Lemma 7. *If* $D(\alpha) < D(\beta)$ *then* $\alpha \prec \beta$

Proof: Let α' be α if α is not an **add** expression and the most complex child of α if α is an **add** expression. Define β' similarly so that the ordering of α and β is determined by the ordering of α' and β' in accordance with Rule 2. Since **add** expressions behave and have the same depth as their most complex child, we know that $D(\alpha') < D(\beta')$ and since **add** expressions cannot contain other **add** expressions as children, we know that neither α' nor β' are **add** expressions. Now β' cannot be a constant since its depth is greater than the depth of α' and constants have 0 depth. Thus if Rule 1 applies, it must be the case that α' is a constant so that $\alpha' \prec \beta'$. If Rule 1 does not apply, then the ordering is determined by Rule 3 which directly implies that $\alpha' \prec \beta'$. Thus $\alpha \prec \beta$. $\square$

We now proceed to prove Theorem 3. If β is not an **add** expression, then by Lemma 1, $D(\alpha) < D(\beta)$ and thus by Lemma 7, $\alpha \prec \beta$. If β is an **add** expression, then there are two cases. If α is a child of β, then by Rule 2, $\alpha \prec \beta$. If α is is a subexpression of a child (recall that the child cannot be an **add** expression), then α has a smaller depth than the child by Lemma 1. Now since the depth of the **add** expression is the depth of its most complex child, it must be the case that $D(\alpha) < D(\beta)$ so that by Lemma 7, $\alpha \prec \beta$. $\square$

Proof of Theorem 6: Initially, every expression is its own signature and its own equivalence class representative, so by Theorem 5, the context satisfies Property 1. Now we must show that this property still holds after a call to `Assert` or `Deny`.

Suppose that we have an arbitrary context C which satisfies Property 1. In the following discussion, we will subscript *find* and signature with the context to which we are referring. Thus we have:

$$\forall\, u \in C.\ u \sim_C \mathit{signature}_C(u).$$

Let $C_0 = C$ and let C_n be the context which results from calling `Assert` or `Deny` from `Validate` so that for $0 \leq i \leq n$, C_i represents an intermediate context, and all such intermediate contexts are represented by some C_i. We will use three lemmas:

Lemma 8. *If $u \sim_{C_i} v$ then $u \sim_{C_j} v$ where $0 \leq i \leq j \leq n$.*

Proof: Each context is derived from the previous context by either adding an expression or merging two equivalence classes. Thus, once two expressions are in the same equivalence class, they will always be in the same equivalence class.

Lemma 9. *If $i < j$ then $\mathit{signature}_{C_j}(\mathit{signature}_{C_i}(u)) = \mathit{signature}_{C_j}(u)$.*

Proof: Let c be an arbitrary child of u. We know that $c \sim_{C_i} \mathit{find}_{C_i}(c)$. By Lemma 8, we know that $c \sim_{C_j} \mathit{find}_{C_i}(c)$. This means that in context C_j, c and $\mathit{find}_{C_i}(c)$ have the same equivalence class representative. Since c and $\mathit{find}_{C_i}(c)$ are corresponding children in u and $\mathit{signature}_{C_i}(u)$ respectively, we thus conclude that u and $\mathit{signature}_{C_i}(u)$ have the same signature in context C_j. $\square$

Lemma 10. *For arbitrary expressions u and v, as a result of executing* `Assert(NewExpr(`*find*`(u) = `*find*`(v)))` *in context C_i, it will be the case that $u \sim_{C_j} v$ for some $j > i$.*

Proof: As mentioned above, `NewExpr` rewrites equalities into a normal form in which the left hand side is simpler than the right hand side. Let $\alpha_1 = \mathit{find}_{C_i}(u)$ and $\alpha_2 = \mathit{find}_{Ci}(v)$. These are already normal form expressions (i.e. no rewrites should apply), so as long as neither one is an interpreted function, the result of `NewExpr` will simply be an equality with the simpler of the two on the left and the other on the right. Thus the call to `Assert` leads directly to a call to `Merge` and thus $u \sim_{C_{i+1}} v$. If α_1 or α_2 is an interpreted function such as **add**, `NewExpr` will solve the equality to place the most complicated variable alone on the right hand side of the equation. Thus we will have a new equation of the form $\alpha' = \beta$ where β is a single variable which appears in either α_1 or α_2. Assume without loss of generality that it appears in α_1. When `Assert` is called on $\alpha' = \beta$, we will go through everything on the use list of β and eventually find α_1. We will replace β with α' in α_1 to get α_2 (or a signature which eventually gets put in the same equivalence class as α_2 if other simplifications on α_2 were pending when the assertion took place). Now, since α_1 and α_2 eventually end up in the same equivalence class in some context $j > i$, it must be the case that $u \sim_{C_j} v$. □

We now proceed with the proof of Theorem 6. Suppose that C_n does not satisfy Property 1. Then there exists some expression e such that it is not the case that $e \sim_{C_n} \mathit{signature}_{C_n}(e)$. Let i be the minimum for which $e \in C_i \wedge \neg(e \sim_{C_i} \mathit{signature}_{C_i}(e))$ (obviously $i > 0$). Consider C_{i-1}. It cannot be the case that $e \notin C_{i-1}$, because `NewExpr` ensures that each new expression is both its own signature and its own equivalence class representative. It must be the case, then, that $e \sim_{C_{i-1}} \mathit{signature}_{C_{i-1}}(e)$. Obviously $\mathit{signature}_{C_{i-1}}(e) \neq \mathit{signature}_{C_i}(e)$. This means that for some child e' of e, $\mathit{find}_{C_{i-1}}(e') \neq \mathit{find}_{C_i}(e')$. The only way for this to happen is if C_i is the result of calling $\mathit{union}(a, b)$ where $b = \mathit{find}_{C_{i-1}}(e')$ and $a = \mathit{find}_{C_i}(e')$. In this case, $\mathit{signature}_{C_{i-1}}(e)$ is on the use list of b. Assuming that we maintain sig pointers correctly so that sig(u) = signature(u)[5], the body of the loop will be executed with u set to $\mathit{signature}_{C_{i-1}}(e)$. This will occur in some context C_j where $i \leq j$. By Lemma 9, $\mathit{signature}_{C_j}(u) = \mathit{signature}_{C_j}(e)$. The result of the body being executed is either nothing, if $u \sim_{C_j} \mathit{signature}_{C_j}(u)$, or a call to `Merge` which merges u and $\mathit{signature}_{C_j}(u)$, or a call to `Assert` which results in $u \sim C_{j'} \mathit{signature}_{C_j}(u)$ for some $j' > j$ by Lemma 10. In each case, we know that $u \sim_{C_k} \mathit{signature}_{C_k}(u)$ where C_k is the context after executing the body of the loop (clearly $k \leq n$). But by Lemma 9, $\mathit{signature}_{C_k}(u) = \mathit{signature}_{C_k}(e)$. Thus, we have $\mathit{signature}_{C_{i-1}}(e) \sim_{C_k} \mathit{signature}_{C_k}(e)$. And by Lemma 8, we know that $e \sim_{C_k} \mathit{signature}_{C_{i-1}}(e)$. So $e \sim_{C_k} \mathit{signature}_{C_k}(e)$ in contradiction to our assumption that e was not equivalent to its signature in contexts C_i through C_n. □

[5] It is easy to see that this is true for atomic expressions, the only relevant case in our optimized algorithm.

References

1. J. R. Burch and D. L. Dill, “Automatic Verification of Microprocessor Control”, In Computer Aided Verification, 6th International Conference, 1994.
2. D. Cyrluk, P. Lincoln and N. Shankar, “On Shostak's Decision Procedure for Combinations of Theories”, Proceedings of the 13th International Conference on Automated Deduction, New Brunswick, NJ, July 1996, 463-477.
3. A. J. J. Dick, “An Introduction to Knuth-Bendix Completion”, The Computer Journal 34(1):2-15, 1991.
4. P. J. Downey, R. Sethi and R. E. Tarjan, “Variations on the Common Subexpression Problem”, Journal of the ACM, 27(4):758-771, 1980.
5. K. Havelund and N. Shankar, “Experiments in Theorem Proving and Model Checking for Protocol Verification”, In Proceedings of Formal Methods Europe, March 1996, 662-681.
6. R. B. Jones, D. L. Dill and J. R. Burch, “Efficient Validity Checking for Processor Verification”, IEEE/ACM International Conference on Computer Aided Design, 1995.
7. D. E. Knuth and P. B. Bendix, “Simple Word Problems in Universal Algebras”, In Computational Problems in Abstract Algebra, ed. J. Leech, 263-297, Pergamon Press, 1970.
8. Z. Manna, et al., “STeP: the Stanford Temporal Prover”, Technique Report STAN-CS-TR-94, Computer Science Department, Stanford, 1994.
9. G. Nelson and D. C. Oppen, “Simplification by Cooperating Decision Procedures”, ACM Transactions on Programming Languages and Systems, 1(2):245-257, 1979.
10. G. Nelson, D. Detlefs, K. R. M. Leino and J. Saxe, “Extended Static Checking Home page”, <URL:http://www.research.digital.com/SRC/esc/Esc.html>, 1996.
11. S. Owre, et al., “Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS”, IEEE Transactions of Software Engineering, 21(2):107-125, 1995.
12. R. E. Shostak, “An Algorithm for Reasoning About Equality”, Communications of the ACM, 21(7):583-585, 1978.
13. R. E. Shostak, “Deciding Combinations of Theories”, Journal of the ACM, 31(1):1-12, 1984.
14. R. E. Tarjan, “Efficiency of a Good but not Linear Set Union Algorithm”, Journal of the ACM, 22(2):215-225, 1975.