Theorem Proving Support for Hardware Verification
—Preliminary Draft—

Deepak Kapur[⋆]

Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
kapur@cs.unm.edu

1 Introduction

Automated reasoning is viewed as a critical technology in designing complex hardware chips, controllers, network protocols and embedded software in devices. Awareness about the limitations of simulation and testing techniques became acute especially after the reported bug in 1994 in the Intel Pentium chip, which is widely believed to be in the SRT division circuit, and which might have cost the company over 500 million dollars.

Hardware chips are becoming complex, because of which it is increasingly being felt that formal techniques and automated tools are needed to complement extensive testing and simulation already employed to ensure their reliability. System integration is emphasized in which chips from different vendors are put together to realize a more complex circuit. Issues faced in reasoning about hardware built from different components are related to the issues arising in reasoning about component-based software, even though hardware is typically regular and highly structured in contrast to software. Testing and simulation of such circuits made from different components become almost impossible as different vendors cannot be expected to provide a uniform behavioral specification for test vectors and simulation. With many hardware primitive operations implemented by microcode or by software, distinction between hardware and software verification may get increasingly blurred in the future.

There will be a growing demand for reliable system design and development methodologies in the future. There will be less tolerance for design bugs and crashes, not only in safety-critical system but also in commercial hardware and software systems. This will especially be true of commercial software. Just like there is growing awareness among hardware designers for the need of automated reasoning and formal methods to help design complex hardware, a similar design culture will evolve among software designers as well. Formal techniques will supplement traditional approaches including testing, simulation, code walk-throughs and design reviews for maximizing design reliability.

We have successfully demonstrated the use of a rewrite-rule based induction prover, *Rewrite Rule laboratory* (*RRL* [33, 45]) for automatically verifying properties of parameterized and generic arithmetic circuits of arbitrary data widths. This family of circuits includes the SRT division circuit as well. We have demonstrated that the existing theorem proving technology which efficiently integrates rewriting and automated mathematical induction with decision procedures for propositional logic, Presburger arithmetic, and data structures generated using free constructors (which includes lists, sequences, numbers), can be used to automatically verify many nontrivial properties of arithmetic circuits. As an example, the SRT division bug could have been detected easily using this approach. Most importantly, a theorem prover such as *RRL* can be run in a push-button mode for reasoning about such circuits.

Theorem provers are, however, not widely used by hardware designers in the design and verification of hardware circuits, even though there has recently considerable interest in formal methods in the hardware design community. Research laboratories associated with main chip designers have

⋆ Partially supported by the National Science Foundation Grant nos. CCR-9712396, CCR-9712366, CCR-9996150, and CDA-9503064.

been building groups in formal methods and investigating automated reasoning methods. In software design, theorem provers are used at best in safety critical applications.

In hardware design, ordered binary decision diagrams (OBDDs) are found useful for checking behavioral equivalence when the functional behavior of circuits as boolean functions is abstracted [7, 8]. Model checking techniques have been found useful for finding bugs in protocols, particularly cache coherence protocols for caches and memory, as well as protocols for networks [10, 11].

Like any methodology, OBDDs and model checking approaches have their own limitations. For instance, OBDDs cannot be used for verifying even simple arithmetic circuits such as multipliers, SRT division, and floating point arithmetic circuits; extensions to OBDDs in the form of OBMDs [8] as well as hybrid decision diagrams [13] have been proposed for handling number theoretic functions. As pointed out recently by Clarke et al [2], OBDDs almost always require exponential space for a large class of circuits, and OBDDs based tools consequently run out of space.

Theorem proving can complement OBDD and model checking. For instance, in the use of model checkers, coping with state explosion requires designing abstractions based on properties being attempted and ensuring their correctness. Many heuristics based on decomposing circuit structure, compositionality, partial order, and symmetry have been proposed, but the problem of huge number of states persists for complex circuits of large data widths. This problem seems to become even more severe when model checking techniques have been attempted for software verification and analysis. The designer developing the abstraction must verify the correctness of the abstraction with respect to the property being proved. A powerful theorem prover is thus needed.

There appears to be a strong relationship between abstraction methods used in model checking, loop invariant generation from loop-programs and intermediate lemma speculation needed during proofs of conjectures by induction. For tail-recursive descriptions as well as parameterized generic arithmetic circuits (e.g. multiplier circuits [26]), we have developed methods for automatically generating lemmas needed in proving properties from the descriptions [28].

In memory modeling and processor verification, the quantifier-free theory of equality with uninterpreted symbols is getting used within the model checking/CAV community [10]. There were at least three papers on this topic in the 1999 CAV conference [12]. Presburger theory of numbers has also been found useful in reasoning about arithmetic circuits. Dill's group has developed heuristics for establishing properties over bit vectors and their number representations [1].

Below, we review the key reasoning mechanisms found useful and effective in our work on using *RRL* for automatically verifying properties of arithmetic circuits. In the last section, we discuss challenges for the automated reasoning community which must be met for automated reasoning tools to become more effective in hardware design.

2 Simplification, Decision Procedures and their Integration

It has been widely observed that large expressions are typically encountered in reasoning about hardware circuits—in the context of modeling the input/output behavior of a boolean circuit, modeling a memory, pipeline architecture with k-stage lookahead, a state transition system as in coherence protocol, large tables, and a specification of a circuit under a restricted set of conditions. The ability to reason about such circuits is limited by the efficiency of simplification performed by automated reasoning tools. If some function symbols are known to have properties such as associativity and commutativity, or just commutativity, simplification is even more challenging since in general, both commutative matching as well as associative-commutative matching are NP-complete.

Unconditional (simple) simplification can be implemented well; there are well-known indexing methods for organizing a large rule set to identify a subterm in a large expression that can be simplified using a rule in the rule set. Typically, simplification of a subgoal is performed within a context using conditional rules which may represent parts of a definition or properties already proved. *Contextual rewriting* is the main primitive inference in *RRL*; it is however not efficient to implement because of reasons discussed below. To establish a conjecture, which is typically a conditional equation (or a clause), the proof is attempted by contradiction. The conjecture is negated and free variables are skolemized (i.e., replaced by constants).

Consider a goal, which is a conjunction of subgoals

$$L_1 \wedge \cdots \wedge L_k,$$

where each L_i, for simplicity, is either a literal of the form $s = t$, or $(s = t) = false$. A (conditional) rewrite rule

$$l \rightarrow r \; if \; cond,$$

where $cond$ is also a conjunction of literals $c_1 \wedge \cdots \wedge c_j$, is said to simplify the goal at the subgoal L_i, to

$$L_1 \wedge \cdots \wedge L_{i-1} \wedge L_i[p \leftarrow \sigma(r)] \wedge L_{i+1} \wedge \cdots L_k,$$

if there exists a substitution σ such that for some position p in L_i, (i) L_i/p[2] $= \sigma(l)$, and (ii) each of $\sigma(c_m)$ (recursively) simplifies to $true$ assuming the context $L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots L_k$ (i.e., every subgoal other than the subgoal L_i being simplified can be assumed to be true).

In case the rewrite rule is unconditional (i.e., there is no $cond$), then the above definition simplifies to checking whether the subterm of L_i at p matches the left side l. In the presence of associative-commutative (or commutative) function symbols, matching above is done modulo such properties.

This is a powerful inference mechanism, but is quite complicated and potentially inefficient since it recursively involves establishing the conditions under the context surrounding the subgoal L_i. As discussed in [43], it subsumes many inference mechanisms in a first-order theorem prover based on resolution, such as demodulation, subsumption and tautology deletion. The completeness of the inference mechanism vis a vis a simplification mechanism in which the simplified goal can replace the original goal, is proved in [43].

The power of contextual rewriting comes at an expense: if some condition in $cond$, when appropriately instantiated, cannot be established (i.e., simplified to $true$), then the particular rule being considered is not applicable. This may not be discovered until deep in the recursive calls. It is important for practical applications that instantiations are not unnecessarily tried. And, further, whenever attempted, available knowledge for establishing the conditions of the possibly applicable rule should be fully utilized. Decision procedures for quantifier-free theories can be useful for this objective as shown below.

2.1 Decision Procedures

The use and effectiveness of decision procedures (especially decision procedures for quantifier-free theories) have been well-recognized in the verification literature since the mid-70's. Bledsoe, Shostak and Nelson and Oppen's influential papers were motivated by their attempts to make the verification of software less manual. SRI's PVS system as well as Stanford's STEP theorem provers rely heavily on decision procedures for a family of theories including propositional calculus, congruence closure on ground terms, and quantifier-free theory of Presburger arithmetic. Dill's verification system also attributes its success in handling large hardware verification examples to a good implementation of a variety of decision procedures including a heuristic for handling bit vectors.

A decision procedure for a given subtheory typically takes as input, a formula in the theory, and outputs a yes or no answer, depending upon the validity of the formula. Most implementations of decision procedures, however, can also deal with problems considered in the realm of a combination of subtheories. A formula which besides having symbols from decidable theories, may include uninterpreted symbols as well, cannot be handled by a single decision procedure. Using constant abstraction techniques, such a formula is decomposed into an equivalent satisfiable conjunction of subformulas so that every subformula has only symbols from a single theory except for equality and constants which are shared across theories. New constants are introduced to stand for subterms. One of the subformulas is a conjunction of equations, equating constants to subterms using uninterpreted symbols. If a subformula belonging to a particular theory can be decided, other subformulas may have to be considered based on how the original formula is decomposed. Otherwise, depending upon

[2] L_i/p denotes the subterm of L_i at position p.

the combination approach followed, a decision procedure may be used to determine equalities among constants entailed by a subformula, which may have to be propagated to other subformulas, or if the decision procedure of a subtheory can solve equations, these solutions in the form of substitutions for variables are propagated to other subformulas. Through such cooperation and interaction among decision procedures of disjoint subtheories, a formula from a combination of theories can be decided.

2.2 Integrating Decision Procedures with Contextual Rewriting

The above definition of contextual rewriting is applicable more than conditional rewriting, but is still restricted. The context of L_i is, however, not used to simplify L_i itself. In this sense, literals in a goal are not normalized with respect to each other. It may be the case that L_i does not match against the left side of a conditional rewrite rule, but if other equality subgoals in the context are used to rewrite L_i, then the rewritten subgoal could have a successful match. This interaction between equality reasoning and contextual rewriting can be achieved by integrating the (ground) congruence closure mechanism with contextual rewriting. For computing congruence closure, a simple naive algorithm for ground completion can be used; for an efficient implementation, the reader may consider Shostak's algorithm implemented as a completion procedure as described in [23]. This algorithm generalizes to the case when some of the function symbols are commutative or associative-commutative.

Instead of checking whether L_i has a subterm at position p which matches the left side of a rewrite rule $l \rightarrow r\ if\ cond$, one can check whether $L_i/p =_c \sigma(l)$, where $=_c$ stands for the congruence closure generated by equality literals in the context (an alternative could be some $L'_i/p = \sigma(l)$, where $L'_i =_c L_i$). Such a check can be prohibitively expensive (it is NP-hard), so a compromise is arrived in the implementation of contextual rewriting in *RRL*. The equality literals in the context are used to compute a canonical rewrite system (which is equivalent to computing congruence closure and generating canonical forms). The canonical rewrite system is then used to normalize L_i (as well as other subgoals in the context). If a contradiction is established this way, the goal is unsatisfiable. Otherwise, the normalized subgoal L'_i is then matched against the left side of a rewrite rule.

For mechanizing verification of properties of arithmetic circuits, two data structures, numbers and bit vectors, play an important role. Their properties realized through decision procedures can also be integrated into the simplification mechanism. In general, the requirements on a decision procedure for integration with contextual rewriting are:

- the decision procedure should be able to detect unsatisfiability of a quantifier-free formula with uninterpreted symbols in the theory, and
- implicit equalities, if any, on terms can be deduced.

Even though the decision procedure should be sound and complete for both the requirements, an incomplete but sound procedure can still be useful. Below we outline how a decision procedure for quantifier-free theory of Presburger arithmetic (over rationals, integers or natural numbers) can be tightly integrated with contextual rewriting.

Contextual Rewriting aided by Presburger Arithmetic

Consider again a goal, which is a conjunction of subgoals $L_1 \wedge \cdots \wedge L_k$, with L_i being the focus subgoal being considered for simplification. It is assumed that literals in the goal may include symbols of Presburger arithmetic. A decision procedure for Presburger arithmetic can be used on the context to simplify the focused subgoal L_i for determining the applicability of a rewrite rule, as well as for simplification.

First, it is checked whether the entire goal is unsatisfiable, in which case, the task is over. Otherwise, if not, then additional implicit equalities, if any, are deduced using the decision procedure from the goal. These equalities are then used by the congruence closure algorithm to simplify the whole goal, including the context as well as L_i. When such simplification caused by the interaction of the decision procedure and the congruence closure stabilizes (i.e., does not yield any further simplification), rewrite rules are analyzed for possible application by considering a maximal literal;

a well-founded ordering is assumed to be defined on the literals which is compatible with the ordering used to establish the termination of rewrite rules.

The applicability of a particular rewrite rule can be determined in two possible ways. Matching of the left side of a rewrite rule against a subterm of a subgoal can be done (modulo Presburger arithmetic) which can, in general, be quite expensive. An alternative is to analyze the left side of a rewrite rule and L_i to determine whether it may be useful to instantiate the rewrite rule for possible interaction with L_i, and its context. One useful heuristic is to exploit the well-founded ordering on terms. If the outermost symbol of the left side of a rule is an interpreted symbol, and there is a maximal subterm of the left side that matches a subterm in the literal L_i of the goal by a substitution σ, then augment the goal with the instantiation of the rule using σ provided the condition of the rule, when appropriately instantiated, is satisfied using the context. Otherwise, if the outermost symbol of the left side of a rule is an uninterpreted symbol, then check for match in L_i of the whole left side.

The above approach for integrating a decision procedure with rewrite rules as implemented in *RRL*, is influenced by Boyer and Moore's work [4] in integrating Fourier's decision procedure in their prover. *RRL*'s approach is more general, however. The main distinction is that whereas Boyer and Moore would convert linear equalities into a conjunction of inequalities, equalities are kept as they are in *RRL*. Additional equalities are deduced from linear inequalities, if any, so that they can be used as rewrite rules for discharging hypotheses/conditions in conditional rewrite rules using ground completion implementing the congruence closure.

Most verification systems including PVS and STEP do not integrate rewriting with decision procedures. Consequently, the user is burdened with having to determine the rule(s) in the rule base that may be applicable for simplifying a goal as well as the precise instantiation. This can be highly laborious and nontrivial task to do manually. In contrast, *RRL* can automatically check applicability of rules as well as find appropriate instantiations when rules are applicable. This feature of *RRL* has been instrumental in automatically finding proofs of properties of arithmetic circuits, including the SRT division circuit.

We believe that in order to make theorem provers more effective, further research must be emphasized in developing new decision procedures for data structures found useful in modeling application domains, including hardware and software, as well as the integration of these decision procedures with simplification in efficiently searching for applicable definitions and lemmas.

3 Induction and Interaction with Decision Procedures

One major distinction between our work on hardware verification and the work using tools such as OBDD and model checkers is that *RRL* can be used to verify properties of circuits of arbitrary widths. This is in contrast to OBDD based approaches where proofs of only fixed data path widths can be done. Even for similar circuits with different data path widths, proofs must be redone using OBDD based tools as these circuits represent different boolean formulas and hence have different behavior. In contrast, a proof by induction in *RRL* can handle a potentially unbounded family of related circuits in one shot.

Unlike control dominated hardware circuits, the behavior of arithmetic circuits can generally be succinctly expressed in terms of number-theoretic properties. Such properties are typically established using induction. In contrast, in OBDD based tools, the properties of arithmetic circuits are expressed using boolean functions. The correctness of the circuit is established by exhibiting the input-output behavioral equivalence of the two boolean functions - one representing the circuit and the other representing the property.

Methods for mechanizing induction play a critical role in automatically proving number-theoretic properties of arbitrary width arithmetic circuits. The mechanization of inductive proofs in *RRL* is done using the *cover set induction* method, the main technique in *RRL* for generating induction schemes from terminating function definitions. The induction schemes are generated using complete function definitions given as terminating rewrite rules. The main steps of the cover set induction method are reviewed below.

1. *Generating cover sets from terminating function definitions:* The definitions of function symbols are first preprocessed by RRL, oriented into terminating rules, and are analyzed for completeness using built-in heuristics. A *cover set* is generated from a complete function definition given as a finite set of terminating rewrite rules.
   A cover set is a finite set of triples, with each rule in the definition contributing a triple. For a rule $l \to r \; if \; cond$, where $l = f(s_1, \cdots, s_n)$ and $f(s_1^i, \cdots, s_n^i)$ is the i^{th} recursive call to f in the right side r, the corresponding triple is $\langle\langle s_1, \cdots, s_n\rangle, \{\cdots, \langle s_1^i, \cdots, s_n^i\rangle, \cdots\}, cond\rangle$. The second and the third components of a cover set triple are the empty set if there is no recursive call to f in r and no condition *cond* associated with the rule.
2. *Generating Subgoals Using Induction Schemes:* Corresponding to a subterm $f(x_1, \cdots, x_n)$ appearing in a conjecture, an induction scheme is generated from the cover set of f: There is one induction case represented as a triple, generated from each cover set triple. The first element of the triple is the substitution σ_c, the second element is the set of substitutions $\{\theta_i\}$ and the last element is the condition, $cond_c$, governing the induction subgoal. The substitution σ_c is used to generate the induction conclusion of the induction subgoal, each substitution in $\{\theta_i\}$ is used to generate an induction hypothesis for the subgoal.

Induction schemes can be specialized using unification for conjectures in which an instantiation of $f(x_1, \cdots, x_n)$ occurs, particularly when induction variables are specialized. This is discussed in [25] as well as in a later subsection discussing interaction between generation of induction schemes and decision procedures.

In contrast to structural induction, there is no fixed inductive inference rule for a data type in the cover set induction approach. Different function definitions on the same data type can lead to different inductive inference rules for the same data type. For instance, a ripple-carry adder is specified by considering one bit at a time of the input bit vectors, whereas a carry-look ahead can be succinctly described using the divide and conquer strategy of dividing the input bit vectors into equal halves. Proving properties of ripple-carry adders requires structural induction on bit vectors, whereas an induction scheme based on halving the bit vectors is effective for proving properties of carry-lookahead adders. For showing equivalence of a ripple-carry adder to a carry-lookahead adder, these two different schemes need to be merged.

For certain conjectures about algorithms defined using different recursion schemes, the induction scheme based on the definition of any one of the algorithms may not lead to a proof of the conjecture. Instead, a scheme that combines induction schemes based on a subset of algorithms is preferred. Such a combination can be computed in certain easy cases using unification and decision procedures; however, generating an appropriate induction scheme is an interesting research problem, which must be further investigated. One possible heuristic, which guides the cover set induction method as well, is to develop an induction scheme which can lead to the applicability of the induction hypothesis(es) in induction subgoals.

Induction Schemes generated using Decision Procedures

Many data structures can be represented in many different ways; for example, numbers can be represented using 0 and s (to stand for the *successor* function) as constructors as well as in terms of $0, 1$ and $+$. Some functions, such as $gcd, divides, rem$, etc., are more convenient, and easier to define using $+$, instead of s. Finite sets can be represented using the *null* set, a constructor that *inserts* an element, or alternatively, using the null set, *singleton* sets and the *union* operation. Similarly, lists can be represented using the *empty* list, and *cons* constructor, or alternatively, the empty list, *singleton* lists and *append* operator on lists. While reasoning about functions defined on such data structures, there is a need to convert from one representation to the other. This could be due to a variety of reasons. A function definition may be given using a mixture of operators using different representations; a conjecture may be expressed using functions which are defined using two different representations, or a conjecture may be expressed using a function that is defined using one representation but its arguments in the conjecture use a different representation.

In [25], we proposed how decision procedures such as one for the quantifier-free theory of Presburger arithmetic can be used in generating appropriate induction schemes. The decision procedure can be used for reconciling the use of different representations of numbers while attempting a proof of a conjecture.

The algorithm for generating an induction scheme is generalized by performing semantic unification, instead of syntactic unification, on a selected subterm $t = f(t_1, \cdots, t_n)$ in a conjecture C, and the cover set associated with f. For a cover set triple, c $= \langle\langle s_1, \cdots, s_n\rangle, \{\cdots, \langle s_1^j, \cdots, s_n^j\rangle, \cdots\}, cond\rangle$, constraint equations $\{t_j = s_j \mid 1 \leq j \leq n\}$ are solved; the decision procedure for Presburger arithmetic generates a substitution σ_c of $Vars(t) \cup Vars(\mathrm{c})$ with context *cond* (if the third component of c is empty then the context is assumed to be *true*). Similarly, a substitution θ_i of $Vars(t) \cup Vars(\mathrm{c})$ for induction hypotheses are generated using the decision procedure on the constraint equations $\{t_j = \sigma_c(s_j^i) \mid 1 \leq j \leq n\}$ with context *cond*.

4 Reasoning about Tables and Case Analysis

Many hardware circuits use a large number of preprogrammed constants, which are typically specified as finite tables; the SRT division circuit, widely believed to be the buggy circuit in the Intel Pentium chip, for example, has a table for calculating the quotient digit based on truncated dividend and truncated divisor [41]. Implementations of arithmetic functions such as the square root and trigonometric functions are based on even much larger lookup tables [16, 46]. Many of these implementations use multiple lookup tables[16, 46]. These tables are realized in hardware using PLAs (programmable logic sumarrays)[37, 16]. Cache coherence protocols can also be specified as large tables, which specify state transitions under certain inputs.

We review extensions to *RRL* for performing extensive case analysis typically arising in the verification of such circuits as well as to handle large finite tables.

There are many ways to specify a finite table. One possibility is to specify it as a predicate (relation) that is a subset of the cartesian product of the index types and the entry type. Both index values for which there is a table entry as well as those index values for which there is no table entry, would have to be given. A lack of functional notation, however, is likely to lead to a cumbersome expression of properties involving a table. A second way to specify a table is as a partially specified function. Each index tuple for which the table has an entry, the function has that entry as its value. A special data type *table* is defined in *RRL* to avoid the burden of having to specify what index tuples must be excluded from its specification, as well have a functional notation for accessing table entries. Ideally, we would like a table to be input graphically to *RRL*.

With either of the above two representations, *RRL* would typically resort to brute force case analysis based on the table indices, regardless of the table structure. This may get prohibitively expensive for large tables. For example, the radix 4 SRT division uses a lookup table for quotient digit selection with up to 800 entries [41]. In order for the verification to scale up to these applications, it is necessary that the underlying structure in tables be better exploited by theorem provers.

In [30] we introduced *sparse* tables based on how frequently particular values appear as table entries. Sparsity in the tables is exploited in correctness proofs by *doing case analysis on the table entries rather on the indices.* The generated cases are used to deduce constraints on the table indices. Additional domain information about table indices can then be used to further simplify constraints on indices and check them. *Dontcare* entries are handled in a special way.

Mechanizing proofs of properties involving tables based on entries may lead to fewer cases with simpler proofs. If the number of distinct table entries is much less than the total number of distinct index tuples, proof attempts using case analysis based on table entries can be helpful. As an example, for the SRT division circuit, 1536 cases are needed in the correctness proof if case analysis is done based on table indices [29]. This is reduced to 12 top level cases by modeling tables as a special data type and performing case analysis on table entries. Each individual top level case generated is much simpler, even though it may have additional subcases.

Often, it is possible to use properties of index types to simplify cases based on entry values; an instantiation of a given conjecture based on a particular table entry value leads to constraints on indices, which can be checked explicitly. Another promising approach is to generate from the simplified conjecture, a constraint on index values that must be satisfied for the conjecture to be valid. It can then be checked whether the index values corresponding to a particular table entry indeed satisfy the constraint. The main idea in deriving constraints on index variables from a given conjecture for a particular table entry value is that of *projection* of the values of index variables. This can be obtained by eliminating non-index variables from the negation of the simplified formula.

5 Lemma Generation

A key distinguishing feature of mechanizing proofs by induction from first-order theorem proving is that often a proof by induction involves use of intermediate lemmas. These lemmas must either be discovered or speculated while attempting an inductive proof, or they have to be supplied by the user for a proof attempt to succeed. To minimize human intervention in guiding proof search, it thus becomes extremely important to develop heuristics for speculating intermediate conjectures that can help in discovering proofs. Further, while attempting proofs of certain conjectures, it is sometimes easier to attempt a proof of a generalization of a conjecture from which the conjecture itself follows. In this section, we discuss three techniques for speculating intermediate conjectures in *RRL* which have been found effective in mechanizing verification of arithmetic circuits.

5.1 Generalization Heuristic

The generalization heuristic implemented in *RRL* to speculate intermediate conjectures is quite effective in verifying number-theoretic properties of arithmetic circuits. Without a proper use of this heuristic, considerable human guidance is required in finding proofs of even simple theorems. In fact, *RRL* is one of the few theorem provers, which, from the recursive definitions of $+$ and $*$ for numbers in terms of 0 and s (the successor function), can automatically generate a proof of distributivity of $*$ over $+$. The theorem prover automatically generates lemmas such as associativity and commutativity of $+$ as well as *associativity* of $*$ needed for this proof. And, one of the main reasons *RRL* succeeds in this proof attempt, is due to the aggressive use of the generalization heuristic, and the backtracking facility which enables *RRL* to try alternatives when one particular attempt fails.

At least two kinds of generalizations are performed in *RRL*:

1. Abstracting a nonvariable subterm appearing in a conjecture to be a variable. Currently, the only requirement on the nonvariable subterm being abstracted is that it must have multiple occurrences in the conjecture (in the left side as well as the right side and/or the condition). It is possible to impose additional requirements, however.
2. Dropping an assumption from a conditional conjecture.

Semantic analysis aided by decision procedures can be useful in the implementation of the generalization heuristic since a subterm may have multiple occurrences semantically but it may not appear to have multiple occurrences syntactically; many such examples are given in [25]. Below, we restrict the discussion to the use of generalization based on syntactic analysis; for a discussion about a generalization heuristic based on semantic analysis, the reader may consult [25, 40].

The generalization heuristic is not only useful for generating stronger lemmas assuming that they are valid, but can sometimes be useful to generate an automatic proof using the cover set method. Proof attempts on the original conjecture may not succeed if all induction schemes are *flawed* [3, 40]. Generalizing a subterm may result in new induction schemes, some of which may be nonflawed. A judicious application of the generalization heuristic based on whether a particular generalization is likely to result in unflawed induction schemes, can be a good way to filter out many generalizations as well as avoid over-generalization.

Another way to avoid reckless generalization is to identify constraints on the subterm being considered for generalization. The subterm is matched against the data base of properties. If the subterm appears in the left side of an unconditional rule that is already proved as a theorem or is a part of a definition, then a *constraint* on the new variable being introduced for the subterm can be generated.

In case an attempted generalization in *RRL* turns out to be invalid (i.e., a counter example is generated) or does not lead to a successful proof attempt, the theorem prover backtracks, and tries a different generalization. At the end, if no generalization can be proved, then the original conjecture is attempted. It thus becomes very important that a invalid generalization is detected quickly; the role of examples to test a generalization becomes critical as a way to discard obviously invalid generalizations.

5.2 Speculating Conjectures guided by the use of Induction Hypotheses

Another heuristic for lemma speculation is to generate intermediate conjectures to serve as bridge lemmas so that the induction hypothesis(es) can be successfully applied in a proof attempt. In an induction subgoal, none of the induction hypotheses may be applicable on the simplified conclusion. Instantiations for non-induction variables in an induction hypothesis may have to be guessed; if a conjecture does not have any non-induction variables, they may have to be introduced by attempting a generalization. A constraint-based approach for speculating conjectures for this was proposed in [28]. Constraints are generated from the simplified conclusion and the induction hypothesis, and are aided by the definitions and the known properties of function symbols appearing in the conjecture. These constraints are also analyzed to speculate the missing side of an intermediate conjecture.

If an induction hypothesis cannot be applied to the conclusion, positions in the subgoal are identified where either side of the hypothesis can be potentially applied, and the corresponding subterm of the subgoal is equated with the appropriate side of the hypothesis. Such equations are called *difference equations*. The induction variables in the difference equations are separated from the non-induction variables in the hypothesis for which appropriate instantiations must be chosen.

The next step is to simplify/solve the difference equations to find substitutions for noninduction variables. Whenever the two sides of a difference equation have the same root symbol, a simple heuristic of *decomposing* the difference equation is used much like unification and matching, to generate simpler equations. The decomposition step fails if the result produces an equation without noninduction variable, whose two sides are not identical. Such an equation cannot be satisfied as no substitutions can be made to make the two sides equal.

This approach has turned out to be quite effective in proving properties from tail-recursive definitions. It can be used to generate intermediate lemmas while attempting proofs of properties of a circuit description given using a tail-recursive scheme, e.g., the ripple-carry adder, components in a multiplier circuit. Our work is similar in its motivation to that of [21] in which approaches for speculating intermediate lemmas and for discovering generalized forms of conjectures for fixing failed induction proof attempts are given using the *rippling* heuristic.

5.3 Circuit Structure and Interpretation of Signals

From the input/output specification of a circuit, it is sometimes possible to generate requirements on its components by propagating specifications through interconnections. This is similar to reasoning about programs where the semantics of linguistic constructs is used to generate intermediate assertions from an input/output specification of the program. Interpretations of interconnections and signals (wires) used for connecting different subcomponents can be helpful.

For example, the correctness proof of a multiplier circuit is instructive in illustrating how interpretation of signals and the circuit structure can be helpful in identifying/speculating intermediate conjectures [26]. A multiplier circuit in the family of multiplier circuits patterned after Wallace tree multiplier circuit design [42] has four subcircuits. The first circuit, `p-all`, computes the partial sums; the second circuit, `k-repeat`, repeatedly performs addition of `k` bit vectors until the result is less than `k` bit vectors; the third circuit, `simple-adder` adds the bit vectors resulting from `k-repeat`. The fourth circuit is the adder, `k-once`, used within `k-repeat` for adding `k` bit vectors. From the whole circuit, we have:

```
k-mult(x, y) == simple_adder(k-repeat  (k, psum-all(x, y))),
```

The output of `gsimple-adder` is also the output of the whole multiplier circuit. Interpretations are used to express the correctness of the multiplier circuit as:

```
bton(k-mult(x, y)) =  bton(x) * bton(y).
```

where `x` and `y` are bit vectors and `bton` is a homomorphism from bit vectors to numbers. Similarly, a collection (list) of bit vectors is interpreted using `btonlist`, which is a homomorphism from a list of bit vectors to numbers.

```
bton(simple_adder(x)) = btonlist(x).
```

Intermediate lemmas relating `btonlist` to the functions `k-repeat, k-once` and `psum-all` with unknown templates are automatically generated by *RRL*.

```
1. btonlist(applst(Y, Z))   ==  G'(btonlist(Y), btonlist(Z))
2. btonlist(k-repeat(k, Z)) == H'(k, btonlist(Z)),
3. btonlist(k-once(k, Z))   == I'(k, btonlist(Z)),
4. btonlist(psum-all(x, y)) ==  J'(bton(x), bton(y)),
```

where `G', H', I'` and `J'` denote unknown term schemas, which must be discovered. Similar to constraint-based approach for lemma speculation discussed in the previous subsection, constraints on the term schemas are generated by attempting a proof by induction using the definitions of the function symbols. From these constraints, instantiations for the term schemas are speculated. More details can be found in [32].

6 Decidable Inductive Properties

In collaboration with M. Subramaniam and J. Giesl, we have recently been investigating an approach to identify conditions under which the validity of a given conjecture to be attempted by induction can be decided [31, 18]. This direction of research is motivated by the need for combining induction proof techniques with push-button approaches for automated reasoning. A drawback of conventional induction theorem provers is that they usually require considerable user intervention in order to perform successful induction proofs. This is perhaps the reason why induction provers have not gained acceptance by the users of applications, especially hardware and protocol verification.

The goal of this research has been is to find syntactical classes of conjectures where inductive validity is decidable. For those conjectures, induction proofs can be accomplished without any user interaction and they only fail, if the conjecture is indeed not valid. The concept of a *theory-based* algorithm is introduced.

A definition of a function symbol f is *$\mathcal{T}$-based* in a decidable theory $\mathcal{T}$ iff for each rule $f(t_1, \cdots, t_m) \rightarrow r$ in the definition, each t_i, $1 \leq i \leq m$, is an interpreted term in $\mathcal{T}$, recursive calls to f, if any, in r only have interpreted terms as arguments, and the abstraction of r defined as replacing recursive calls to f in r by variables is an interpreted term in $\mathcal{T}$.

It can be shown that a simple conjecture of the form, $f(x_1, \cdots, x_m) = r$, where f has a theory-based definition and r is a term in the decidable theory, can be decided using the cover set method. The main objective is to ensure that for each induction subgoal generated from a conjecture after selecting an induction scheme, the resulting formula after simplification and application of induction hypothesis(es) whenever possible, is in a decidable theory. The following decidable theories have been considered: quantifier-free theory of Presburger arithmetic, congruence closure, propositional calculus as well as the quantifier-free theory of constructors (mostly, free constructors as in the case of finite lists and finite sequences).

The approach is extended to more complex equations than just simple conjectures. Additional conditions on equational conjectures and interaction among the definitions of other functions appearing in algorithms that guarantee the simplification of each induction subgoal are identified. It is shown that the cover set method can be used as a decision procedure for a subclass of such equational conjectures.

In order to increase the applicability of the approach, these results have been extended from equations to arbitrary quantifier-free formulas [18]. Kapur and Giesl are currently working on syntactical characterizations of such general formulas where inductive validity is still decidable. The crucial concept for this extension is the use of so-called *correctness predicates.* A characterization of conjectures where these correctness predicates describe the exact set of instantiations for which the conjecture under consideration is true has been developed. The above results are being extended to quantifier-free theories of bit vectors, finite sets, and finite lists, as such data structures are extensively used in many application domains.

7 Challenges

We have shown that existing theorem proving technology can go a long way in ensuring reliability of a certain family of circuit descriptions. For simple adder, multiplier and division circuits, a theorem prover such as *RRL* which provides support for fast contextual rewriting, decision procedures for equality on ground terms and numbers, as well as mechanization of induction with intermediate lemma generation features, and backtracking, can be used as a push-button tool. Number-theoretic properties about such circuits can be proved automatically without any human guidance. This is in sharp contrast to the experience with other theorem provers including Boyer-Moore's prover, ACL2, PVS, Larch and NuPRL, which require considerable human guidance for finding proofs. Our experience with *RRL* gives hope that with additional enhancements and extensions to theorem proving tools, it is possible to automatically verify a larger class of circuits.

Can we claim that we have a tool that can be used by hardware designers for designing such circuits? The answer is no, as there are a number of issues which must be addressed and *RRL* must be enhanced to get to that level of use. In the rest of this section, we first address enhancements to address some peculiarities of *RRL*. Then, we discuss extensions to theorem provers (including *RRL*) to enhance their reasoning power to broaden their scope and make them widely applicable to a larger family of circuit designs. Finally, we discuss issues related to making theorem provers more effective for the application of hardware verification.

As of now, *RRL* does not use an efficient representation of numbers. Naturals and integers are represented in unary notation. And, rationals can be modeled as pairs of integers. Operations on numbers are implemented using these representations. The efficiency of rewriting and other inference mechanisms used in the decision procedures can be significantly improved by adopting a more efficient/compact representation of numbers.

Given that most hardware circuits are dealing with bit vectors and signals (control and data), developing special data structures and reasoning mechanisms suited to handle these concepts are likely to help in increased automation of verification attempts of circuits.

RRL does not have any special data structure for representing state machines and reasoning about reachable states. For control-theoretic properties, it is more effective to use a state machine model, support reasoning algorithms for state machines and their properties. To deal with the state explosion problem faced by most model checkers, it is necessary to provide heuristics for designing and verifying abstractions to control the complexity of the state space.

In general, there is a need to combine the most effective ideas/strategies from diverse approaches to automated reasoning and integrate them without compromising the basic goals—automation, ease of use, and effectiveness in the application of system modeling and design, be it hardware, software or hybrid. Properties considered trivial in an application domain by domain experts should be decided automatically without burdening the expert. The theorem prover should be viewed as a real helper, enabling the expert to focus on interesting problems and design issues, instead of having to worry about aspects either considered trivial or unrelated to the application. A theorem prover must especially provide strong support for propositional reasoning, rewriting, induction, decision procedures, model checking, conjecture speculation, and backtracking. A few symbolic computation algorithms for solving/analyzing combinatorial identities, differential and polynomial equations found useful in modeling hybrid systems, and methods for manipulating polyhedra specified using linear inequalities should be integrated as well. Abstract interpretation techniques found useful for controlling the state explosion problem in model checking should be supported. Relationship between abstraction techniques, invariant generation and lemma speculation in proofs by induction need to be explored [28].

The model checker should support directly the description of transition systems, and validation of temporal formulas in transition systems. Explicit state enumeration methods and symbolic representation of states and transitions using boolean formulas (as well as other symbolic representations) should be explored as each technique has its strengths and weaknesses. Propositional theorem provers such as SATO [44] should be integrated into an induction/rewrite-based prover. Weak second-order theory of monadic successors should be incorporated. A major challenge to consider is the interaction of the decision procedures and induction heuristics with model-checking approaches for handling data values, and dealing with abstraction techniques. The main research issues which need considerable investigation and research are:

- Develop a good framework for combining decision procedures with rewriting and heuristics for mechanizing induction.
- Develop an efficient simulation/symbolic evaluator tool well integrated with a theorem prover.
- Develop an approach for generating counter-examples from a diverse set of approaches to automated reasoning.

There are also issues having to do with making automated reasoning tools more effective for use by hardware and system designers. For a theorem proving tool to be acceptable to designers, not only should it address a need felt by the designers, it also needs to be integrated into the overall design environment. The use of a theorem prover should not be perceived as a burden or something to be avoided unless becomes unavoidable.

Hardware description languages cannot be easily interfaced to automated reasoning tools even though there has recently been some progress in this direction. The input language accepted by most tools is quite different from the notation familiar to hardware designers. The input language of *RRL* is a functional programming language using recursion, and not a hardware description language. There is a need to develop an interface between *RRL* and a tool for describing hardware designs from which circuits can be synthesized. A theorem prover should be able to process descriptions used by hardware designers, instead of requiring descriptions in a special purpose language viewed unrelated to hardware designs. A good step in making progress in that direction would be to develop a translator from a subset of a commonly-used hardware description language by hardware designers, e.g. Verilog or VHDL, to the input language of *RRL*.

A theorem prover must provide feedback found useful by a designer when it is unable to prove a property about a design. This could be in the form of a counter-example (test) that establishes that the property is not true of the circuit. If the portion of the circuit can be localized because of which the conflict between the circuit description and the property may be arising, this can be helpful in understanding the problem possibly leading to a fix. Some believe that this can perhaps be the most useful role for automated reasoning tools.

References

1. C. W. Barrett, D. L. Dill, and J. R. Levitt, "A decision procedure for bit-vector arithmetic," Proc. *35th Design Automation Conference,* June 1998.
2. A. Biere, E. M. Clarke, R. Raimi, and Y. Zhu, "Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs," Proc. *CAV '99*, 60-71, LNCS 1633, Springer, 1999.
3. R.S. Boyer and J S. Moore, *A Computational Logic*. ACM Monographs in Computer Science, 1979.
4. R.S. Boyer and J S. Moore, Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic, *Machine Intelligence* 11 (1988) 83-157.
5. R.S. Boyer and J S. Moore, *A Computational Logic Handbook*. New York: Academic Press, 1988.

6. B C. Brock, W.A. Hunt and M. Kaufmann, *The FM9001 Microprocessor Proof.* Technical Report 86, CLI, Dec. 1994.
7. R.E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Computers*, C-35(8), 1986.
8. R. E. Bryant, and Y.-A. Chen, *Verification of Arithmetic Functions with Binary Moment Diagrams.* Tech. Rep. CMU-CS-94-160, June 1994.
9. R.E. Bryant, *Bit-level Analysis of an SRT Divider Circuit.* Tech. Rep. CMU-CS-95-140, Carnegie Mellon University, April 1995.
10. J. R. Burch, E. M. Clarke, K. L. Mcmillan and D. L. Dill, "Sequential circuit verification using symbolic model checking," Proc. *27th ACM/IEEE Design Automation Conference*, 1990.
11. A.J. Camilleri, M.J.C. Gordon and T.F.Melham, "Hardware verification using higher-order logic"", *HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione (editor) 43-67, N.Holland, Amsterdam 1987.
12. Proc. *Computer Aided Verification (CAV), 1999.* LNCS, Springer.
13. E. M. Clarke, M. Fujita, and X. Zhao, "Hybrid decision diagram: Overcoming the limitation of MTBDDs and BMDs", Proc. *International Conference on Computer-Aided Design (ICCDA)*, 1995, 159-163.
14. D. Cyrluk and S. Rajan and N. Shankar and M. K. Srivas, "Effective theorem proving for hardware verification", Proc. 2^{nd} *Conference on Theorem Provers in Circuit Design*, (eds. Kumar and Kropf), Sept. 1994.
15. N. Dershowitz, "Termination of rewriting," *J. of Symbolic Computation*, 3, 69-116.
16. M.D. Ercegovac and T. Lang, *Division and Square Root: Digit Recurrence Algorithms and Implementations.* Kluwer, 1994.
17. S. German, *Towards Automatic Verification of Arithmetic Hardware.* Lecture Notes, 1995.
18. J. Giesl & D. Kapur, *Decidable Classes of Inductive Theorems.* Draft, University of New Mexico, USA, 2000.
19. See a special issue on Hardware/Software Codesign, *Proc. of IEEE,* March 1997.
20. W.A. Hunt and B.C. Brock, "The verification of a Bit-Slice ALU," *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects,* Springer LNCS 408, 1989.
21. A. Ireland "The use of planning critics in mechanizing inductive proofs", Proc. *LPAR 92*, A. Voronkov (editor), LNAI 624 Springer Verlag, 1992.
22. D. Kapur, "Rewriting, decision procedures and lemma speculation for automated hardware verification," Proc. *10th Intl. Conf. Theorem Proving in Higher Order Logics,* LNCS 1275, 1997.
23. D. Kapur, "Shostak's congruence closure as completion," Proc. *Intl. Conf. on Rewriting Techniques and Applications* (*RTA-97*), Barcelona, Spain, June 1997.
24. D. Kapur and X. Nie, "Reasoning about numbers in Tecton," Proc. *8th Intl. Symp. Methodologies for Intelligent Systems, (ISMIS'94)*, Charlotte, North Carolina, October 1994, 57-70.
25. D. Kapur and M. Subramaniam, "New uses of linear arithmetic in automated theorem proving for induction," *J. Automated Reasoning,* 16(1-2), 1996, 39-78.
26. D. Kapur and M. Subramaniam, "Mechanically verifying a family of multiplier circuits," Proc. *Computer Aided Verification (CAV'96)*, New Jersey, Springer LNCS 1102 (eds. Alur and Henzinger), 1996, 135-146.
27. D. Kapur and M. Subramaniam, "Mechanical verification of adder circuits using powerlists," *J. Formal Methods in System Design,* 13(2), Sep. 1998, 127-158.
28. D. Kapur and M. Subramaniam, "Lemma discovery in automating induction," Proc. *Intl. Conf. on Automated Deduction, CADE-13*, LNAI 1104 (eds. McRobbie and Slaney), New Jersey, July 1996.
29. D. Kapur, M. Subramaniam "Mechanizing Reasoning About Arithmetic Circuits: SRT Division," Proc. *17th FSTTCS,* LNCS (eds. Sivakumar and Ramesh), 1997.
30. D. Kapur, and M. Subramaniam, "Mechanizing reasoning about large finite tables in a Rewrite based theorem prover,", Proc. *ASEAN-8,* LNCS 98, 1998.
31. D. Kapur & M. Subramaniam, "Extending decision procedures with induction schemes," Proc. *CADE-17*, LNAI, Pittsburgh, PA, USA, 2000.
32. D. Kapur and M. Subramaniam, "Using an induction prover for verifying arithmetic circuits," accepted for publication in *J. of Software Tools for Technology Transfer,* Springer Verlag, 2000.
33. D. Kapur, and H. Zhang, "An overview of Rewrite Rule Laboratory (RRL)," *J. of Computer and Mathematics with Applications,* 29, 2, 1995, 91-114.

34. J Moore, T. Lynch and M. Kaufmann, *A Mechanically Checked Proof of the Correctness of the AMD5K86 Floating Point Division Algorithm.* Technical Report, CLI, March 1996.
35. P.S. Miner and J.F. Leathrum Jr., "Verification of IEEE compliant subtractive division algorithm," Proc. *FMCAD'96*, LNCS 1166, Palo Alto, CA, 1996.
36. G. Nelson, and D.C. Oppen, "Simplification by cooperating decision procedures," *ACM Tran. on Programming Languages and Systems* 1 (2), 1979, 245-257.
37. A.R. Omondi, *Computer Arithmetic Systems: Algorithms, Architecture and Implementations,* Prentice Hall 1994.
38. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* MIT Press, 1991.
39. R.E. Shostak, "Deciding combination of theories," *Journal of ACM* 31 (1), 1984, 1-12.
40. M.Subramaniam, *Failure Analyses of Inductive Theorem Provers.* Doctoral Dissertation. State University of New York, Albany 1996.
41. G.S. Taylor, "Compatible hardware for division and square root," Proc. *5th IEEE Symp. on Computer Architecture,* May 1981.
42. C.S. Wallace, "A Suggestion for a fast multiplier," *IEEE Trans. Electron. Comput.*, EC-13:14-17, 1964.
43. H. Zhang, "Implementing contextual rewriting," Proc. *3rd Intl. Workshop on Conditional Term Rewriting Systems,* Springer LNCS 656, (eds. Remy and Rusinowitch), 1992, 363-377.
44. H. Zhang, *SATO: An efficient propositional prover,* Proc. *International Conference on Automated Deduction (CADE-97),* LNAI 1104, Springer-Verlag, 308-312.
45. H. Zhang, D. Kapur, and M.S. Krishnamoorthy, "A mechanizable induction principle for equational specifications," Proc. *9th Intl. Conf. Automated Deduction (CADE),* Springer LNCS 310, (eds. Lusk and Overbeek), Chicago, 1988, 250-265.
46. J.H.Zurawski, J.B. Gosling, "Design of a high speed square root multiply and divide unit", *IEEE Trans. on Computers*, vol. C-36, 1987.