Int J STTT (2000) 3: 1–34

Using an induction prover for verifying arithmetic circuits

Deepak Kapur[1], **M. Subramaniam**[2]*

[1] Department of Computer Science, University of New Mexico, Albuquerque, NM 87131, USA; E-mail: kapur@cs.unm.edu
[2] Microprocessor Division, HAL Computer Systems, Fujitsu Inc., Campbell, CA 95014, USA; E-mail: subu@hal.com

Abstract. We show that existing theorem proving technology can be used effectively for mechanically verifying a family of arithmetic circuits. A theorem prover implementing: (i) a decision procedure for quantifier-free Presburger arithmetic with uninterpreted function symbols; (ii) conditional rewriting; and (iii) heuristics for carefully selecting induction schemes from terminating recursive function definitions; and (iv) well integrated with backtracking, can automatically verify number-theoretic properties of parameterized and generic adders, multipliers and division circuits. This is illustrated using our theorem prover *rewrite rule laboratory* (*RRL*). To our knowledge, this is the first such demonstration of the capabilities of a theorem prover mechanizing induction.

The above features of RRL are briefly discussed using illustrations from the verification of adder, multiplier and division circuits. Extensions to the prover likely to make it even more effective for hardware verification are discussed. Furthermore, it is believed that these results are scalable, and the proposed approach is likely to be effective for other arithmetic circuits as well.

Key words: CE[a]

1 Introduction

Ever since Intel admitted to a bug in the division algorithm of its Pentium chip in November 1994, interest in the use of formal methods and tools supporting them, especially for enhancing the reliability of hardware circuits, has increased considerably both in the industry and in academia. In early 1995, we decided to try our theorem prover *rewrite rule laboratory* (*RRL*) for verifying properties of arithmetic circuits. We started with adder circuits [29], then moved on to multiplier circuits [28]. Most of this work was completed in 1995. In late 1996 and early 1997, we used *RRL* for the analysis of the SRT division circuit. The invariant properties of the SRT division circuit have been verified using *RRL* in three different ways [24, 31, 34].

In this paper, we discuss our efforts, focusing on the features of the theorem prover *RRL* found useful for this application. The main goal of our investigations has been to determine the extent to which a rewrite rule based theorem prover can be used to automatically verify arithmetic circuits, much like BDD-based software and the related Boolean equivalence checkers frequently used in the hardware design community. We propose some extensions to and future work on theorem provers such as *RRL* which would make them better suited for the application of hardware design analysis and verification.

Three classes of commonly used arithmetic circuits including adders, multipliers and divider circuits are described in this paper. The adder circuits, being the simplest, are used to illustrate the basic capabilities of *RRL*. Then, we show how *RRL* can be used to give a common specification and a fully automatic proof of an important class of multiplier circuits including several commonly used multipliers. The identification of such a class and exhibiting their commonality is one of the contributions of this paper. These circuits bring out the effectiveness of the induction and lemma generation capabilities of *RRL*. Finally, divider circuits illustrate the power of integration of linear arithmetic decision procedure and contextual rewriting implemented in *RRL*.

In the next section, we briefly outline the approach taken for verifying properties of arithmetic circuits. In the

* Partially supported by the National Science Foundation Grant no. CCR-9712366

third section, we review *RRL* and its capabilities. In subsequent sections, we discuss the main features of *RRL* found useful for this application, using the case studies of verification of properties of various arithmetic circuits.

1.1 Verification approach

Our verification attempts are quite modest and limited in the sense that we never consider real circuits or their descriptions in some well-known hardware description language. Instead, circuits are described in a functional equational language with simple data types acceptable by *RRL*. This is much in the spirit of the earlier work on CE[b] hardware verification using Boyer and Moore's prover and ACL2, as well as PVS work, although the language used by *RRL* is simpler in contrast to the powerful specification language of PVS, which is based on higher-order logic and syntax and allows parameterization and dependent types.

Adder and multiplier circuits in this paper are described algorithmically using recursive definitions on bit vectors (which are represented as lists of bits). Properties of these circuits are typically expressed in terms of number-theoretic properties. In the case of SRT division, we have followed the approach taken in [13], where the algorithm is given using numbers instead of bit vectors. Writing such a specification in terms of bit vectors should not add significantly to the complexity of verification process.

2 Related work

Different approaches have been proposed in the literature for verifying arithmetic circuits. These can be classified into the following three main techniques: state-based techniques using BDDs and their variants and model checkers [8, 11]; induction-based techniques adapted from software verification [7, 21]; and techniques based on modeling hardware circuits using higher-order logics [12, 14]. The latter two approaches have often collectively been referred to as theorem proving based approaches. Papers on these approaches have appeared in recent conferences such as *CAV* and *FMCAD*.

These approaches have traditionally been compared and contrasted in terms of their automation capability and their expressiveness. The state-based approaches have been espoused as the most automated of the three approaches. Induction-based approaches provide some degree of automation aided by several built-in heuristics in theorem proving tools. Finally, the approaches based on higher-order logic tend to be highly interactive. State-based approaches are typically applicable only over finite domains whereas induction-based approaches are applicable to both finite as well as unbounded domains. The approaches based on higher-order logics are the most expressive of the three.

State-based approaches based on symbolic manipulation of Boolean functions using binary decision diagrams *BDDs* [8] as canonical representations for Boolean functions are perhaps the most popular for verifying hardware circuits of fixed word size. A hardware circuit is specified using a Boolean function that can be succinctly represented using a *BDD*. *BDDs* provide a fast mechanism for comparing Boolean functions. Since the size of a Boolean function and hence the associated *BDD* is dependent on the word size, these approaches are well-suited for verifying non-parametric circuits. Even for linear circuits, in which the output is a linear function of the inputs, this approach has two major limitations: (i) it is unclear how circuits of arbitrary word size can be verified; and (ii) verification is limited to showing that a circuit implements a Boolean function, and not a function on numbers.

Verification of parametric (generic) descriptions of circuits has been typically carried out using theorem-proving approaches. Not only is it possible to prove equivalence of Boolean functions of arbitrary word sizes, but more importantly, it is possible to verify that a Boolean function indeed implements a given number-theoretic function. Another advantage of these approaches is that a single proof suffices to establish the behavioral correctness of a generic circuit which stands for a family of circuits of different word sizes. A major criticism of the theorem proving approaches is that they are semi-automatic based on heuristics, and often require expert user guidance. This is especially so for the approaches based on higher-order logics that primarily support an interactive mode of operation.

The approach used in this paper is in the spirit of Hunt's work [21], in which circuits are described as recursive functions, and their properties proved using a theorem prover mechanizing induction. Our goal in this paper, however, has been to demonstrate how the induction-based approach can achieve a degree of automation comparable to state-based approaches for arithmetic circuit verification. We have shown how the existing rewriting technology along with induction techniques implemented in *RRL* can automatically discharge correctness proofs of arithmetic circuits while retaining all the advantages provided by the general framework underlying theorem provers.

The case studies performed in *RRL* are described in Table 1. All of these case studies were done on a Sparc-5 workstation with 32 Mb of memory.

We briefly review the literature on verification of adders, multipliers and division circuits in the rest of this section.

2.1 Adder and multiplier circuits

A linearly specified ripple carry adder where an adder of size n is recursively specified in terms of an adder of size CE[c] n - 1 has been verified by a number of theorem

CE[b] Please add ref cite after 'Boyer and Moore's' below.
CE[c] Please change hyphen in line below to minus sign.

Editor's or typesetter's annotations (will be removed before the final TEX run)

Table 1. Arithmetic circuit case studies in RRL

Circuits	Definitions	Lemmas	Time(Secs)
Ripplecarry-Carrylookahead Adders (numeric repres.)	15	2	14.00
Ripplecarry(linear)-Ripplecarry (div.& conq.)Adders (numeric repres.)	23	3	17.73
Ripplecarry(linear)-Ripplecarry (div.& conq.)Adders (bit repres.)	21	3	14.40
Ripplecarry(linear) Adder (numeric repres.)	11	0	10.25
Ripplecarry(linear) Adder (bit repres.)	9	0	7.73
CarrySave Adder (bit repres.)	7	0	6.25
Linear Array Multiplier	12	0	2.48
Wallace Tree Multiplier	12	0	2.45
7-3 Multiplier	12	0	6.22
Radix 4 SRT Divider (abstract table)	0	0	900
Radix 4 SRT Divider (explicit table)	12	0	60

proving systems including *PVS, Nqthm, HOL, SPIKE-AC, Clam-Oyster*. However, verification efforts for the ripple-carry adder with a divide-and-conquer representation and the carry-lookahead adder have been very few, perhaps because these circuits are complex. Most verification efforts involving the carry-lookahead adder have been done in the context of verifying different forms of ALUs and processors. In [49] and [1], the verification of a parameterized ALU is reported using *Nqthm* and the *HOL* system respectively. The correctness proof described in [49] requires around 22 user-suggested intermediate lemmas to establish the correctness of the ALU with respect to addition.

In [7], Brock et al., describe the use of *Nqthm* for a comprehensive case study on the verification of the FM9001 microprocessor that includes a proof establishing the equivalence of a carry-lookahead adder and a ripple-carry adder. The proof is rather involved, using a number of built-in library functions and requires significant user intervention. In many of these proofs, the user has to explicitly provide the induction scheme.

In [29] we discussed the correctness proofs of several adder circuits including ripple-carry and carry-lookahead adders using *RRL*. As described in the table above, the correctness proof of the ripple-carry adder can be established automatically in *RRL*. The correctness of a carry-lookahead adder is done by exhibiting its equivalence to a ripple-carry adder. This requires only two intermediate lemmas. The carry-lookahead scheme used by us [29] is regular and is similar to the lookahead scheme *propagate-generate* scheme reported in [7].

Fixed size adder circuits are easily verified using BDD-based techniques. This is typically done by specifying both the behavior of the adders and the circuit in terms of Boolean functions and checking for the equivalence of these functions.

Unlike adder circuits, verification efforts involving multiplier circuits have been relatively few. It is well known that BDD-based techniques do not work well for multiplier circuits. Bryant and Chen introduced a new data structure *multiplicative binary moment diagram (BMD)* for modeling the functionality of circuits in terms of data at the word level [9]. Using this approach, a number of integer multiplier designs with word sizes up to 256 bits have been verified. However, such verifications are not fully automatic as stated in [9].

Our approach for verifying multiplier circuits is similar to the one suggested using BMDs in the sense that the circuit is decomposed into two components, and the number-theoretic correctness of the individual components is established. The overall proof then follows by the composition of these two components. However, the lemma generation heuristics in *RRL* automatically generate the required specification of these components, and the composition is also automatically done based on the circuit structure. Due to the generality afforded by theorem provers like *RRL*, it was also possible to obtain a common proof for a family of multiplier circuits of arbitrary size (parametric circuits) which would be infeasible otherwise.

A linear array multiplier has also been verified using the theorem prover PVS [42]. The proof is interactive and requires user guidance.

2.2 Division circuits

Since Intel's Pentium bug was reported in the media, there has been a lot of interest in automated verification of the SRT divider circuits [10, 13, 19, 36, 43].

For the SRT division circuit, Bryant [10] discussed how BDDs can be used to perform a limited analysis of some of the invariants. As reported in [43], Bryant had to construct a checker-circuit much larger than the verified circuit to capture the specification of the verified circuit.

Taylor's description of the SRT division circuit has been formalized by [13, 19] using the languages of Maple, a computer algebra system, and Analytica, a prover built over Mathematica, another commercially available computer algebra system. A correctness proof of the SRT divider circuit was then done using Analytica. The main feature of the proof was an abstraction of the quotient selection table using the six boundary value predicates. This abstraction had to be manually provided. The proof of invariants using this intensional representation of quotient selection table involves reasoning about inequalities, which can become quite tedious and involved. Even though it is claimed in [13] that the proof is "fully automatic" (p. 111 in [13]), the proof (especially, the proof of the second invariant regarding the boundedness of partial remainders) had to be decomposed manually and the two assumptions had to be discharged manually.

Our first proof attempt of SRT division discussed in [24] was essentially an exercise to determine how much of the Analytica proof [13] could be done automatically by *RRL* without having to use any symbolic computation algorithms of computer algebra systems. We mimicked the proof in [13] but by making data dependency of various circuit components explicit on different data paths, as well as by identifying different assumptions made in the proof reported in [13]. Much to our surprise, once we succeeded in translating the Analytica specification to *RRL*'s equational language (which, by the way, was the most nontrivial part of this proof), *RRL* was able to find proofs of all the formulas (the first invariant and the second invariant with the assumptions, as well as the discharging of assumptions) automatically, without any interaction.

In [31], we gave another proof of the SRT division circuit using an explicit representation of the quotient digit selection table, thus getting rid of an aspect of the specification development where human guidance is used for abstracting table entries as predicates. Further, since the abstract representation of the quotient selection table using boundary value predicates in [13, 24] just considers the minimum and maximum values of a partial remainder for every quotient digit, thus losing information on other relations among entries, it is possible to certify erroneous tables correctly. Even though the proof using explicit table representation has nearly 1536 subcases in contrast to 96 subcases in the proof in [24], the proof of each subcase is much easier and a lot quicker to obtain, as a subformula typically involves numeric constants that can be easily simplified.

During the course of this proof, we observed that the proofs of many of the subgoals share a common structure. We have described how this commonality can be automatically exploited in the theorem prover *RRL* by formalizing tables as a special data type and by exploiting the sparsity of the quotient digit selection table in SRT division. This leads to a compact proof with only 12 top-level cases.

The proof reported in [36, 43] using the PVS system is more general than the above proofs of the SRT division circuit. It includes a general theory of SRT division for arbitrary radix r and an arbitrary redundant quotient digit range $[-a, a]$. The theory is instantiated for the radix 4 SRT division. The specification is developed with considerable human ingenuity, and the resulting proof is manually driven, even though parts of the proof can be done automatically using previously developed PVS tactics. As reported in [43], the correctness proof of the table implementation itself took 3 h of cpu time, with the whole proof taking much longer even with user's help.

Miner and Leathrum's work [36] generalizes the proof in [43] to IEEE floating point numbers and establishes the correctness of an IEEE compliant SRT division algorithm. Moore et al. [35] reported a proof of correctness of the kernel of a microcoded floating point division algorithm implemented in AMD's 5K86 processor. The proof is done CE^{d} using ACL2, a descendant of Boyer and Moore's prover. No claim is made about making the proof automatic, but rather the main emphasis is on formalizing the IEEE floating point arithmetic to verify the division algorithm based on Newton-Raphson's method.

3 Rewrite rule laboratory

The theorem prover *rewrite rule laboratory* (*RRL*) supports equational and inductive reasoning using rewrite techniques. The specification language of *RRL* is equational, with support for defining abstract data types using constructors. *RRL* transforms its input into equations and conditional equations to be universally quantified. For instance, circuits and their behavioral specifications are transformed as equations and conditional equations. Definitions are distinguished from properties (lemmas) using := for definitions and == for properties to stand for the equality symbol. The correctness of circuit descriptions is established by proving various properties about these descriptions, and showing that they meet the behavioral specifications.

RRL is different in its design philosophy from most proof checkers such as *PVS, HOL, Isabelle, NUPRL, LP*, in the sense it attempts to perform most inferences automatically without user guidance. In this sense, it is closer in spirit to the Otter and EQP provers developed at Argonne National Laboratory. Many proofs in *RRL* can be generated automatically; *RRL* can be used in such cases as a push-button theorem prover. In fact, that is how we typically use *RRL* for finding proofs, starting without having any clue about how a proof can be done by hand.

RRL has built-in heuristics for:

1. Orienting equations into terminating rewrite rules
2. Identifying the next rewrite rule to apply for simplification, and for that, determining the instantiation of the free variables, and discharging conditions, if any, of the rewrite rule
3. Invoking decision procedures for numbers (quantifier-free Presburger arithmetic), bits, data types with free constructors, and propositional logic
4. Selecting the next inference rule
5. Automatic generation of case analysis
6. Choosing induction schemes based on the definitions of function symbols appearing in a conjecture and interaction among these definitions

7. Generating intermediate lemmas needed
8. Automatic backtracking to try an alternative proof attempt when one proof attempt fails.

Much like Boyer and Moore's theorem prover *Nqthm* [5] all the heuristics are applied in the same order using the same strategy in *RRL*.

The user is thus relieved of the task of having to determine the sequence in which rewrite rules should be applied, when decision procedures should be invoked, how rewrite rules should be instantiated, when induction is performed, variables to be used for induction, and what induction scheme should be used. Below, we briefly review some of these heuristics.

Each rewrite rule used by *RRL* must be terminating, which is ensured by an algorithm implementing a well-founded reduction ordering, called lexicographic recursive path ordering, for comparing terms based on precedence relations among function symbols [16]. It is possible to override this feature of *RRL* but then there is no guarantee that simplification using manually-oriented rules terminates. Terminating rewrite rules are automatically used for simplification as well as for constructing induction schemes for mechanizing proofs by induction.

Simplification with respect to a context (called *contextual rewriting*) is the main inference mechanism used by *RRL*. The simplification algorithm in *RRL* automatically determines the applicable rewrite rule on a given conjecture. This is done by first determining the possible instantiation for the variables in the rewrite rule, and then ensuring that the conditions in the rewrite rule, if any, are satisfied. Discharging of conditions is done taking into account the context of the formula being simplified and using other rewrite rules and the decision procedures which are tightly integrated with rewriting.

RRL attempts to prove a conjecture by normalizing its two sides using contextual rewriting and the decision procedures for discharging any hypotheses, if any, and checking whether the normal forms of the two sides of the conjecture are identical. If it succeeds, then the proof is said to have been obtained using equational reasoning and decision procedures.

If an equation cannot be proved by simplification, then a proof by induction is attempted. Variable(s) to perform induction on CE[e] the induction scheme are automatically selected using heuristics implemented to support the *cover set* method. The definitions of function symbols appearing in a conjecture are analyzed. An induction scheme is generated from the definition of one (or more) function symbol(s) selected, based on well-founded ordering used to establish termination of these function definitions. This scheme is often successful in determining the truth-value of the conjecture. The conjecture would be split into many cases, each corresponding to a subgoal to be proved in order to prove the original conjecture. Each subgoal is then tried just like the original conjecture.

If a proof attempt based on a particular induction scheme does not lead to a counter-example, but also does not succeed, *RRL* automatically backtracks to pick another induction scheme (and perhaps different induction variables) for attempting the conjecture. Additional inductions may be necessary to establish the induction subgoals. The depth of permissible inductions is provided as a parameter in *RRL* that can be modified by the user. The proof attempt of a subgoal fails once the number of inductions exceed this depth. The number of inductions in establishing a subgoal can be iteratively increased until a diverging pattern and/or need for additional lemmas is apparent.

RRL supports a variety of heuristics for automatically generating intermediate lemmas based on formulas generated during a proof attempt. We consider the intermediate lemma speculation research to be the most critical for automating proofs by induction. *RRL* implements a simple heuristic for conjecture speculation by abstracting common subexpressions appearing in a conjecture to new variables using certain criteria as well as by weakening a condition in a conditional conjecture. Another heuristic found especially useful for proving properties of tail-recursive definitions (which is indeed the case for arithmetic circuits including adders and multipliers) is that of generating bridge lemmas which facilitate the use of induction hypotheses in a proof attempt of an attempted valid conjecture. A constraint-based approach is CE[f] developed to speculate about intermediate conjectures as well as guess instantiations for non-induction variables in a conjecture [30, 46]. In verifying properties of arithmetic circuits, some of the intermediate lemmas needed can be generated from the circuit structure and component specifications, as illustrated later for multiplier circuits.

Lemmas which cannot be generated automatically by *RRL* must be provided by the user. This is where *RRL* needs guidance from the user.

When a proof attempt fails and a proof cannot be found automatically, the transcript is looked at, which may reveal a variety of things. The conjecture may have to be modified, a definition may have to be fixed, or perhaps, an intermediate lemma needs to be proposed.

Below, we list the main features of *RRL* found useful for hardware verification. These features are illustrated in subsequent sections where different arithmetic circuits are discussed in more detail.

3.1 Useful features of RRL for arithmetic circuit verification

Four major features seemed to have contributed to *RRL* being effective in our mechanization attempts in verifying properties of arithmetic circuits:

1. Fast and automatic contextual rewriting and reasoning about equality

CE[e] The word 'and' eliminated in line below, OK?
CE[f] The word 'about' added in line below, OK?

Editor's or typesetter's annotations (will be removed before the final TEX run)

2. Decision procedures for numbers and freely built recursive data structures such as lists and sequences, and their effective integration with contextual rewriting
3. The cover set method for mechanization of proofs by induction, and its integration with contextual rewriting and decision procedures
4. Intermediate lemma speculation.

In Sect. 4 we review contextual rewriting, its interaction with decision procedures for equality on ground terms and quantifier-free theory of numbers, as well as the cover set method for mechanizing proofs by induction. A ripple-carry adder is used for illustrations.

Section 5 focuses on specification and verification of a family of multiplier circuits. It discusses how a theorem prover such as *RRL* can be used for generic verification of a family of multipliers of arbitrary width, implementing the same generic algorithm but using components of different numbers of signals (adding a partial sum vs 3 partial sums vs 7 partial sums), and using different components realizing the same behavior.

Section 6 provides details about intermediate lemma speculation heuristics using examples from adder and multiplier circuits.

Section 7 is concerned with the handling of large tables and theextensive case analysis often needed in verifying circuits including radix-4 SRT division circuits, that use tables implemented as PLAs (programmable logic arrays).

4 Inference mechanisms: contextual rewriting, decision procedures, and cover-set induction

In this section, we discuss two key primitive inference mechanisms of *RRL* which turned out to be the most effective in verifying properties of arithmetic hardware circuits. We first review *simple* contextual rewriting (integrated with a decision procedure for reasoning about Horn clauses); then we show its interaction and integration with a decision procedure for equality on ground terms. Finally, it is shown how a decision procedure for a quantifier-free theory of numbers is integrated with contextual rewriting. Features/properties of a decision procedure for tight integration with conditional rewriting are described.

Later, we discuss the cover set method for generating induction schemes as implemented in *RRL*, and contrast it with the structural induction method. We also discuss heuristics for choosing an appropriate induction scheme.

The examples of ripple-carry adder circuits are used for illustration.

4.1 Contextual rewriting

Contextual rewriting is the main primitive inference in *RRL*. To establish a conjecture, which is typically a conditional equation (or a clause), it is first simplified with the help of existing definitions and already proved lemmas using contextual rewriting. The proof is attempted by contradiction. The conjecture is negated and free variables are skolemized (i.e., replaced by constants). The negated conjecture, which is a conjunction of subgoals, is simplified by contextual rewriting, with an attempt to derive a contradiction in the form of a $false = true$ or a literal conjuncted with its negation. Tautology of the form $s = s$ is deleted from the conjunction of subgoals. Below, we show how contextual rewriting is different from rewriting.

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

$$l \to r\ if\ cond,$$

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

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

if for some position p in L_i, there exists a substitution σ such that

– L_i/p[1] $= \sigma(l)$, and
– 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 of the unconditional rewrite rule.

The reason such simplification is called contextual rewriting (in contrast to rewriting) is because the remaining subgoals in the goal are used as context and are assumed to be true while determining whether the conditions of the rewrite rule are true. This is illustrated using a simple example below.

Consider a simple conditional rewrite rule

$$p(x) \to true\ if\ (q(x) \wedge\ x \neq c).$$

Given a goal

$$p(a) \wedge q(a) \wedge a \neq c,$$

$p(a)$ by itself cannot be rewritten unless other subgoals in the goal are also used as its context. With the context $\{q(a), a \neq c\}$, $p(a)$ can be rewritten using the above rewrite rule to $true$, simplifying the goal to $q(a) \wedge a \neq c$, since the condition of the rewrite rule is satisfied by the context.

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

As the reader will notice, this is a powerful inference mechanism since it recursively involves establishing the conditions under the context surrounding the subgoal L_i. As discussed in [52], 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-à-vis a simplification mechanism in which the simplified goal can replace the original goal is also proved there. The power of contextual rewriting comes at a price: 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. So, it is important for practical applications that instantiations are not unnecessarily tried.

So far, we have discussed contextual rewriting in its simplest form. Below, we discuss how equality subgoals in a goal can be used to rewrite other subgoals, before attempting contextual rewriting on the conjecture. In a later section, we discuss how decision procedures can be integrated with contextual rewriting, using the example of quantifier-free theory of Presburger arithmetic.

4.2 Role of congruence closure in contextual rewriting

In the above definition of contextual rewriting, the context of a subgoal L_i is 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 constant congruence closure mechanism with contextual rewriting. In a later section, we show how a decision procedure for interpreted symbols other than equality can be used for normalizing a goal, much like the constant congruence closure algorithm.

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 (ground) congruence closure 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; trivial literals of the form $t = t$ are dropped from the normalized goal using tautology deletion. Otherwise, the normalized subgoal L'_i is then matched against the left side of a rewrite rule, as illustrated below using a simple example.

Consider again the rewrite rule discussed above:

$$p(x) \rightarrow true\ if\ (q(x) \wedge\ x \neq c).$$

Given another related goal

$$p(a) \wedge q(b) \wedge a = b \wedge a \neq c.$$

$p(a)$ cannot be rewritten using the above rule by simple rewriting or contextual rewriting since the condition $q(a) \wedge a \neq c$ cannot be satisfied. However, if the subgoal $a = b$ is used to simplify the rest of the subgoals in the goal using congruence closure, we get the sightly modified goal

$$p(b) \wedge q(b) \wedge a = b \wedge b \neq c.$$

Using the substitution $x \rightarrow b$ on the rewrite rule, $p(b)$ is rewritten to *true* by contextual rewriting. The goal simplifies to:

$$true \wedge q(b) \wedge a = b \wedge b \neq c.$$

The above analysis assumed $a \succ b$, but a similar analysis works if $b \succ a$, resulting in a different simplified goal:

$$q(a) \wedge a = b \wedge a \neq c.$$

For computing (ground) 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 [25]. Special care must be taken to handle equality literals of the form $x = t$, where x is a variable. If t does not include an occurrence of x, then this equality is oriented as $x \rightarrow t$, leading to elimination of x. This heuristic turns out to be very useful for simplification purposes.

4.3 Integration of a decision procedure with contextual rewriting

For mechanizing verification of properties of arithmetic circuits, two data structures, numbers and bit vectors, play an important role. In the previous section, it was discussed how equality on ground terms can be integrated with contextual rewriting. In this section, we discuss how to extend contextual rewriting further so that semantic information about a data structure encoded in a decision procedure can be efficiently exploited, leading to increased automation. Function symbols in the theory of the data structure being considered will be called *interpreted,* whereas other symbols are assumed to be uninterpreted.

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
- 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.

We will illustrate this integration of a decision procedure into contextual rewriting using the example of the quantifier-free theory of Presburger arithmetic, involving $0, s, +$ and $\leq, =$ for the data structure of numbers; more details can be found in [26].

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 include symbols of Presburger arithmetic. Recall that $L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \cdots \wedge L_k$ serve as the context of L_i.

Above, it was shown how the context can not only be used to discharge conditions in rewrite rules used for simplification, but equality literals in the context can also be used to simplify the focused subgoal L_i using ground congruence closure. Now the decision procedure for the theory of the data structure 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.

Using a decision procedure for a quantifier-free theory of numbers as an example, first it can be checked whether the context of L_i is unsatisfiable; if so, then the goal is unsatisfiable. Otherwise, if not, then additional equalities, if any, can be deduced using the decision procedure from the context, which can then be used by the constant congruence closure algorithm to simplify the context as well as L_i. When such simplification using the decision procedure for numbers and the constant congruence relation stabilizes (i.e., does not yield any further simplification), then rewrite rules are analyzed for possible application.

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 with respect to the theory of the data structure (also known as E-matching in the literature) which, in general, is 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 properly instantiate the rewrite rule for interaction with L_i. A useful heuristic is based on a 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 L_i 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. Otherwise, if the outermost symbol of the left side of a rule is an uninterpreted symbol, then check for a match in L_i of the whole left side.

Suppose the following conjecture over the integers is attempted:

$$\begin{aligned}&(p(x) \wedge (x \leq max(x,y)) \wedge (z \leq f(max(x,y)))\\&\quad \wedge (0 < min(x,y)) \wedge (max(x,y) \leq x))\\&\qquad \supset (z < g(x) + y).\end{aligned}$$

Assume that among other rules, the following rewrite rules for max, f, g, p are already in the data base.

1. $min(x,y) \rightarrow y \quad if \quad max(x,y) = x,$
2. $f(x) \leq g(x) \rightarrow true \quad if \quad p(x).$

When the conjecture is attempted using RRL, it is first negated and skolemized to give:

$$\begin{aligned}&p(A) \wedge (A \leq max(A,B)) \wedge (L \leq f(max(A,B)))\\&\quad \wedge (0 < min(A,B)) \wedge (max(A,B) \leq A))\\&\quad \wedge \neg(L < g(A) + B).\end{aligned}$$

The resulting goal is simplified. Equality literals in the goal are used to generate a canonical rewrite system that can then be used for simplifying all literals in the subgoal. Since the only equality literal is $p(A)$, nothing happens on this account. Now the decision procedure for numbers is invoked on the linear literals (those involving operations on numbers) to check for unsatisfiability or, if satisfiable, deducing implicit equalities. (A literal is linear if its atom is an inequality (with $\leq$) or an equality.) This leads to the generation of the equality literal $max(A,B) = A$, which can, in turn, be used to modify the congruence relation represented as a canonical rewrite system. This in turn may further simplify the goal; in this case that indeed happens, as $max(A,B)$ is simplified to A in other literals. The resulting subgoal, after tautology deletion (such as $A \leq A$), is then:

$$\begin{aligned}&p(A) \wedge (A = max(A,B)) \wedge (L \leq f(A))\\&\quad \wedge (0 < min(A,B)) \wedge \neg(L < g(A) + B).\end{aligned}$$

Now rule 1 is applicable on literal $(0 < min(A,B))$ since the condition $max(A,B) = A$ of the instantiated rule is in the context of $(0 < min(A,B))$. The resulting goal is:

$$\begin{aligned}&p(A) \wedge (A = max(A,B)) \wedge (L \leq f(A)) \wedge (0 < B)\\&\quad \wedge \neg(L < g(A) + B) \wedge (min(A,B) = B).\end{aligned}$$

The new constant congruence relation, as generated by the equality literals does not lead to any simplification, nor does the arithmetic decision procedure detect any unsatisfiability or new equality literals.

Now we illustrate another interesting aspect of the integration of the decision procedure with contextual rewriting. The second rule has the linear literal $f(x) \leq$

CE^g 'and' added in line below, OK?

Editor's or typesetter's annotations (will be removed before the final TEX run)

MS ID: STTTN032

31 May 2000 14:59 CET

$g(x)$ (since its outermost symbol is $\leq$) as its left side, in which the maximal term is CE^{g} $f(x)$, assuming $f \succ g$ in the precedence used to order terms for termination. The literal $L \leq f(A)$ in the simplified goal has a subterm that can match the maximal term in the left side of a linear rule. Given that the condition of this rule can be discharged from the context, rewriting now means conjoining the literal from the instantiated rule to the goal, giving:

$$
\begin{aligned}
&p(A) \wedge (A = max(A, B)) \wedge (L \leq f(A)) \wedge (0 < B)\\
&\quad \wedge \neg(L < g(A) + B) \wedge\\
&\qquad (min(A, B) = B) \wedge (f(A) \leq g(A)).
\end{aligned}
$$

The arithmetic decision procedure now detects a contradiction, thus showing the goal to be unsatisfiable, implying that the original conjecture is proved from the two rules along with the theory of equality, propositional calculus and the theory of numbers. Note that without instantiating rule 2 and using it, it would not have been possible to establish the goal above, even though no instance of the left side of rule 2 appears in the original goal or its intermediate simplified forms.

The reader will notice the interaction between the congruence closure and the arithmetic decision procedure through the deduction of implicit equalities, and between the arithmetic decision procedure and rewriting for discharging conditions as well as for deducing useful instances of rewrite rules for simplification.

Rewrite rules in which the outermost symbol of the left side is a function on the data structure whose semantics is being integrated (e.g., rule 2 above) using an associated decision procedure, must be handled especially for proper/tight integration. Applicability of such rules can be tested by making weaker requirements – by requiring that the maximal subterm with an uninterpreted symbol in the left side ($f(x)$ in the left side of rule 2), not just the whole left side ($f(x) \leq g(x)$), match against a subterm ($f(a)$) in the conjecture. If so, the instance of the rewrite rule thus generated can be augmented for further deduction and analysis. As illustrated in the above example, the left side of the rule does not have a match in the conjecture. The instance of the rule obtained from the maximal subterm match, $f(a) \leq g(a)\ if\ p(a)$, is simplified to $f(a) \leq g(a)$ and added to the conjecture.

The 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. The main distinction is that whereas Boyer and Moore convert linear equalities into a conjunction of inequalities, equalities are kept as they are. 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. Even though the procedure for checking unsatisfiability of linear inequalities over integers (as well as for deducing equalities) is incomplete, it has been found quite effective in using *RRL* for mechanical verification of arithmetic circuits.

4.4 Mechanizing induction

One major distinction between the use of *RRL* and BDD-based tools for verifying properties of arithmetic circuits is that *RRL* can be used to verify properties of circuits of arbitrary widths. This is in contrast to BDD-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 BDD-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 BDD-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 thus play a critical role in automatically proving number-theoretic properties of arbitrary width arithmetic circuits. In this section, we review the cover set induction method, the main technique for generating induction schemes in *RRL*. We discuss how induction schemes are automatically generated from terminating function definitions. This is illustrated by proving number-theoretic properties of an arbitrary width ripple-carry adder circuit. Later we discuss heuristics implemented in the theorem prover for choosing an appropriate induction scheme, and contrast the cover set induction approach with the structural induction method.

Cover set induction method

A proof of a given conjecture of the form $l = r\ if\ cond$, is always first equationally attempted in *RRL* by contextual rewriting and decision procedures. If a conjecture cannot be established or refuted by equational reasoning, then *RRL* automatically attempts to prove the simplified conjecture by induction.

RRL uses the *cover set induction* method for automating well-founded induction. The cover set method was proposed in [53], and it has been successfully used to prove many nontrivial theorems by induction on numbers, lists, arrays and other recursive data structures. Along with contextual rewriting, this is another powerful inference mechanism supported in *RRL*.

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.

The cover set method automatically generates induction schemes for a given conjecture. The induction schemes are generated using complete definitions of functions given as terminating rewrite rules. The induction subgoals are automatically generated from the induction scheme. We describe the main steps of the cover set induction method 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 several built-in heuristics. A *cover set* is generated from each complete terminating function definition.
   A cover set associated with a definition of a function f is a finite set of triples. There is one triple for each rule in the definition. The first element of the triple is derived from the left-hand side of the rule. The second element of the triple is a set whose elements are derived from the recursive calls to f on the right-hand side of the rule. The last element of the triple is derived from the conditions governing the rule. For a rule $l \to r\ if\ cond$, where $l = f(s_1, \cdots, s_n)$ and $f(s^i_1, \cdots, s^i_n)$ is the i^{th} recursive call to f in the right side r, the corresponding triple is $\langle\langle s_1, \cdots, s_n\rangle, \{\cdots, \langle s^i_1, \cdots, s^i_n\rangle, \cdots\}, cond\rangle$.
   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 if there is no condition *cond* associated with the rule.
2. *Generating subgoals using induction schemes:* an induction scheme is generated from a term $f(x_1, \cdots, x_n)$ appearing in the conjecture using the cover set of f. An induction scheme is a finite set of induction cases. There is one induction induction case corresponding to each cover set triple. Each induction case generates an induction subgoal which must be established in order to prove the given conjecture. Each induction case is a triple whose elements are generated from the corresponding elements in the 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 governing the induction subgoal $cond_c$. 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.

For an induction scheme based on a cover set to be sound, a cover set must have two properties. First, it must be complete i.e., for each induction variable, all possible values of a data type must be considered. Second, in the induction step, the substitutions for generating induction hypotheses must be lower in a well-founded order than the substitutions in the conclusion. The second property is automatically ensured in *RRL* since induction schemes are generated from terminating definitions. To ensure the first property, *RRL* supports algorithms for checking completeness of function definitions given as terminating rewrite rules.

The cover set method has been extensively used in all our arithmetic circuit verification efforts. The working of the cover set method on a simple example of a parameterized *ripple-carry* adder is described in detail below.

Specifying a ripple-carry adder

A ripple-carry adder is a simple hardware circuit that implements the conventional method of adding two binary numbers. The inputs to a ripple-carry adder of size n are two bit vectors of size n and an initial carry bit. The outputs is a bit vector of size $n+1$. Bits are added one by one from the least significant to the most significant, with the carry from the previous stage as input, and carry output at the current stage to serve as the input to the next stage.

Bit vectors can be modeled in *RRL* by using lists *freely* constructed with *nil* and *cons*. The elements of lists are bits modeled in *RRL* by an enumerated type with constructors 0 and 1.

A parameterized ripple-carry adder using lists can be specified in *RRL* as follows:

```
1. rca(x, nil, nil) := cons(x, nil),
2. rca(x,cons(y1,nil),cons(z1,nil)) :=
   cons(mod2(x,y1,z1), cons(half(x,y1,z1),nil)),
3. rca(x,cons(y1,y),cons(z1,z)) :=
   cons(mod2(x,y1,z1), rca(half(x,y1,z1), y, z))
   if {len(y) = len(z), not(y = nil),
   not(z = nil)}.
```

The first equation defines a ripple-carry adder of size 0 to be a wire propagating the input carry bit. The output carry bit is tied to value 0. The second equation defines a ripple-carry adder of size 1 to be a full adder. A full adder takes three bits as inputs and produces a lists with two bits as its output. The first bit in the list corresponds to the sum obtained by adding the three input bits. The other corresponds to the carry obtained by the addition. The function `mod2` in the equation is an abbreviation for the sum computation of a full adder. It stands CEh for the bit-level operation `xor`. The function `half` is an abbreviation for the carry computation of a full adder. `half(x, x1, x2)` stands for the bit-level operation `(x and x1) or (x and x2) or (x1 and x2)`.

The third equation defines an adder of size n recursively in terms of an adder of size *n - 1* cascaded with a full adder. The full adder is used for adding the initial carry bit with the least significant bits of the two input bit vectors. The carry bit from the full adder is taken to be the initial carry bit for the ripple-carry adder of size $n-1$. The function `len` above computes the length of a list.

CEh A hyphen inserted between 'bit' and 'level' twice in paragraph below. OK?

Editor's or typesetter's annotations (will be removed before the final TEX run)

These equations are oriented into terminating rules using lexicographic recursive path ordering implemented in *RRL*. By associating a right-left status with the function symbol `rca` (i.e., using a lexicographic ordering from right to left among the arguments of the function symbol `rca`), the recursive call `rca(half(x, y1, z1), y, z)` is smaller than the left-hand side `rca(x, cons(y1, y), cons(z1, z))` in a well-founded order on terms induced by a precedence relation on function symbols. A cover set is generated from the above definition of `rca`, which is then used to generate an induction scheme for conjectures in which `rca` appears. Induction hypotheses are generated from the smaller recursive call to `rca` on the right side of the definition.

Showing that ripple-carry adder implements addition

A proof of correctness of the ripple-carry adder `rca` (i.e., `rca` implements addition on bit representation of numbers) is mechanically generated by *RRL*. It is demonstrated that given any two bit vectors y and z and an initial carry bit x, the natural number derived from the output pair obtained as a result of the ripple carry addition of x and the bit vectors y and z is the same as the sum of the numbers corresponding to y and z along with x. A functions `bton` is defined to convert linear lists of bits into numbers.

```
bton(nil)          :=  0,
bton(cons(x, y)) :=  x + (2 * bton(y)).
```

Note that the bit `x` is overloaded and treated as a number in the above definition. Such overloading of bits and numbers is assumed in other function definitions as well.

The theorem expressing the correctness of the ripple-carry adder is:

```
C1: x + bton(y) + bton(z) == bton(rca(x, y, z))
     if len(y) = len(z),
```

where + is addition on natural numbers.

The theorem `C1` is first attempted by *RRL* using equational reasoning using contextual rewriting and the linear arithmetic decision procedure. Since `C1` cannot be established equationally, *RRL* automatically invokes the cover set induction to prove `C1`.

The cover sets for all the non-constructor function symbols are precomputed by *RRL*. In this case, *RRL* precomputes the cover sets of the function symbols `+, len, bton, rca`. For example, the cover set that is precomputed for the function symbol `rca` is

```
[<<x, nil, nil>, {}, {}>,
 <<x, cons(y, nil)>, cons(z, nil>>, {}, {}>,
 <<x, cons(y1, y), cons(z1, z)>,
 {<half(x, y1, z1), y, z>},
 {len(cons(y1, y)) = len(cons(z1, z)),
 not(y = nil), not(z = nil)}}>].
```

The first two triples above are derived from the first two equations defining `rca` for the empty and singleton bit vector inputs. The last triple is derived from the third equation defining `rca` recursively for bit vectors of size greater than 1. The first component of this triple comes from the left side of the equation. The second component is derived from the recursive call to `rca` on the right side, and the third component is derived from the condition in third equation. The second component of the cover set triple records distinct recursive calls.

In order to perform an inductive proof of `C1`, one of the function symbols `+, len, bton, rca` appearing in `C1` is chosen. In this case, the heuristics in *RRL* pick the function symbol `rca` to perform induction.

The rewrite rule for `rca` defines `rca` for all the values that the formula `C1` is being CE[i] proved (arbitrary carry bit and equal length bit vectors)[2].

The induction scheme for `C1` is automatically generated by *RRL* from the subterm `rca(x, y, z)` in `C1` by using the cover set of `rca`:

```
Let P(x, y, z) be
bton(rca(x, y, z)) == (x + bton(y) + bton(z))
if {len(y) = len(z)}

Induction will be done on x, y, z in rca(x, y, z),
with the scheme:
[1] P(x, nil, nil).
[2] P(x, cons(x1, nil), cons(x2, nil)).
[3] P(x, cons(y1, y), cons(z1, z))
    if {(len(y) = len(z), not(y = nil),
    not(z = nil)),
    P(half((x, y1, z1)), y, z)}.
```

Three induction subgoals are generated based on this scheme. The first two subgoals correspond to the base cases and the third goal is the induction step case.
The first induction subgoal,

```
[1] bton(rca(x, nil, nil)) == x + bton(nil) +
    bton(nil) if {len(nil) = len(nil)},
```

reduces to true by the definitions of the function symbols `bton, rca` and `len`.
The second subgoal,

```
[2] bton(rca(x,cons(x1,nil),cons(x2, nil))) ==
    x + bton(cons(x1,nil)) + bton(cons(x2,nil))
    if {len(cons(x1,nil)) = len(cons(x2,nil))},
```

simplifies by the definitions of `rca, len` and `bton` to:

```
mod2(x, x1, x2) + half(x, x1, x2) +
half(x, x1, x2) == x + x1 + x2.
```

This subgoal is proved by *RRL* by case analysis on the values of the variables `x, x1, x2`.

[2] Note that `rca` is not completely defined for all possible input combinations. The equations define only cases where the bit CE[j] vectors are of equal lengths. In such cases the cover set induction method requires the `C1` to be relativized to be proved only over defined values [27]. The condition `len(y) = len(z)` in `C1` provides such a relativization.

CE[i] Could the word 'proved' be substituted with 'tested for'in line below? Alternatively, another option might be '..for all the values for which the formula..'above.
CE[j] 'are' added to line below. OK?

Editor's or typesetter's annotations (will be removed before the final TEX run)

The third subgoal [3] is

```
[3] bton(rca(x,cons(y1,y),cons(z1,z))) ==
    x + bton(cons(y1,y)) + bton(cons(z1,z))) if
    {(len(cons(y1, y)) = len(cons(z1, z))),
    not(y = nil), not(z = nil),
    (bton(rca(half(x, y1, z1), y, z)) =
    (half(x, y1, z1) + bton(cons(y1, y)) +
    bton(cons(z1, z)))) if len(y) = len(z)}
```

This subgoal is split into two intermediate subgoals by contextual rewriting based on the combination of the clauses. One of the intermediate subgoals generated is

```
[3.1] bton(rca(x,cons(y1,y),cons(z1,z))) ==
      x + bton(cons(y1,y))+bton(cons(z1,z)) if
      {len(cons(y1, y)) = len(cons(z1, z),
      not(y = nil), not(z = nil),
      not(len(y) = len(z))}.
```

Contextual rewriting of this subgoal with the definition of `len` combined with the linear arithmetic procedure, for the freeness of the successor function on numbers, gives conditions `len(y)` = `len(z)` and `not(len(y)` = `len(z))`. The subgoal is vacuously true due to contradictory conditions.
The other subgoal generated from [3] is

```
[3.2] bton(rca(x,cons(y1,y),cons(z1,z))) ==
      x + bton(cons(y1,y))+bton(cons(z1,z)) if
      {len(cons(y1,y)) = len(cons(z1,z)),
      not(y = nil), not(z = nil),
      bton(rca(half((x,y1,z1)),y,z)) =
      bton(y) + bton(z) + half(x, y1,z1))}
```

Note that the body of the above formula corresponds to the induction conclusion and the final conjunct in the condition corresponds to the induction hypothesis. The conclusion is simplified by the definition of `rca` and `bton` to,

```
[3.2.1] mod2(x, y1, z1) +
        2 * bton(rca(half(x, y1, z1), y, z) ==
        x + y1 + 2 * bton(y) + z1 + 2 * bton(z)
        if {len(y) = len(z), not(y = nil),
        not(z = nil)}
```

to which the hypothesis can be applied. The resulting formula

```
[3.2.1.1] mod2(x, y1, z1) + 2 * bton(y) +
          2 * bton(z) + 2 * half(x, y1, z1)) ==
          x + y1 + 2 * bton(y) + z1 + 2 * bton(z)
          if {len(y) = len(z), not(y = nil),
          not(z = nil)},
```

is simplified by *RRL* by using linear arithmetic decision procedure to,

```
[3.2.1.1.1] mod2(x, y1, z1) + half(x, y1, z1) +
            half(x, y1, z1) == x + y1 + z1
            if {len(y) = len(z), not(y = nil),
            not(z = nil)},
```

which is proved by case analysis on the variables `x, y1, z1`. The conditions in the above formula are actually dropped by `RRL` since none of the variables in the conditions occur in the body[3].

All of the above steps in proving the formula [3.2] are done using the combination of contextual rewriting and the arithmetic decision procedure. Application of the hypothesis is treated just like another instance of contextual rewriting. A single application of contextual rewriting integrated with the arithmetic procedure produces the formula [3.2.1.1.1] directly from the subgoal [3.2].

Choosing an appropriate induction scheme

The choice of an appropriate induction scheme is important for the success of a proof attempt by induction. For a given conjecture, there are many possible alternative induction schemes, corresponding to function symbols appearing in the conjecture. Further, a particular function symbol may appear more than once in a given conjecture with different variables as arguments. A choice among these subterms corresponds to choosing different induction variables and leads to different subgoals.

For instance, for the conjecture `C1`,

```
x + bton(y) + bton(z) == bton(rca(x, y, z))
if len(y) = len(z).
```

There are three possible induction schemes based on the function symbols `bton`, `len` and `rca`. For the schemes based on `len` and `bton` either `y` or `z` could be chosen as the induction variable. The induction scheme based on `rca` uniquely determines both `y` and `z` as induction variables. A proof of `C1` using the induction scheme suggested by the function symbol `rca` was described in the previous section.

Below, we first describe how the choice of the other alternative induction schemes leads to unsuccessful proof attempts. Then, we describe how such failures are avoided by the heuristics implemented *RRL* for choosing the most appropriate induction scheme and induction variables.

Consider proving `C1` using the induction scheme suggested by `bton` with `y` as the induction variable. Two induction subgoals are generated for `y = nil` and `y = cons(u1, u)`.
The base case with `y = nil` simplifies to

```
C1.1: x + bton(z) == bton(rca(x, nil, z))
      if len(z) = 0.
```

Since no further simplification is possible by contextual rewriting and/or the decision procedures, the proof attempt does not succeed.

In such a case, the formula is treated as an intermediate conjecture by *RRL*, and an inductive proof of this

[3] This is one form of *generalization* that *RRL* supports. Generalization and other forms of intermediate lemma generation support in *RRL* are discussed in detail in the next section.

intermediate conjecture is attempted. An inductive proof of `C1.1` could be performed based on schemes derived from the function symbols `bton` or `len` with the induction variable being `z`. An induction proof attempt based on the function symbol `bton` leads to two subgoals with `z = nil` and `z = cons(v1, v))`.
The base case with `z = nil` simplifies to

```
x == bton(rca(x, nil, nil)).
```

The above formula is reduced to true by the definitions of `rca, bton`.
The induction step case with `z = cons(v1, v)` is easily established since the condition `0 = len( cons(v1, v))` reduces to false using the definition of `len` and the linear arithmetic decision procedure.

The formula `C1.1` is thus an inductive theorem. Such intermediate inductive theorems are oriented into rules by *RRL*, and are added to CE[k] the rule database. These rules are used by *RRL* in subsequent as well other subgoals of the same proof. This is one way intermediate lemmas are generated, proved and stored by *RRL*. For further details on intermediate lemma generation, the reader can refer to the next section.

Coming back to the proof of `C1`, we now consider the induction step case generated based on the induction scheme based on the function symbol `bton` with the induction variable `y`.
In the step case, the conclusion with `y = cons(u1, u)` is,

```
x + bton(cons(u1,u)) + bton(z) ==
bton(rca(x,cons(u1,u),z))
if len(cons(u1,u)) = len(z),
```

with the hypothesis being,

```
x + bton(u) + bton(z) == bton(rca(x, u, z))
if len(u) = len(z),
```

The conclusion cannot be simplified any further. Thus the induction step case cannot be established by equational reasoning.

As done for the base case, an inductive proof of the step case can be attempted based on a scheme suggested by the function symbol `bton` with the induction variable `z`. Two subgoals are generated with `z = nil, z = cons(v1, v)`.
The base case with `z = nil` is reduced to true due to contradictory conditions involving `len`.
In the step case, the conclusion is

```
x + bton(cons(u1,u)) + bton(cons(v1,v)) ==
bton(rca(x,cons(u1,u),cons(v1,v)))
if {len(cons(u1, u)) = len(cons(v1, v)),
(x + bton(u) + bton(cons(v1,v)) =
bton(rca(x,u,cons(v1,v))) if
len(u) = len(cons(v1,v)))},
```

with the hypothesis being

```
x + bton(cons(u1, u)) + bton(v) ==
bton(rca(x, cons(u1, u), v))
if {len(cons(u1, u)) = len(v),
(x + bton(u) + bton(v) = bton(rca(x, u, v))
if len(u) = len(v)}.
```

The conclusion cannot be simplified any further[4]. Additional inductions based on the scheme `bton` are not likely to help either. So, the induction proof attempt of `C1.1.` based on the scheme suggested by the function symbol `bton` fails.

An induction proof attempt based on a scheme suggested by the function symbol `len` similarly fails as well.

To avoid such failures it is necessary that the heuristics in *RRL* choose the most appropriate induction scheme suggested by `rca` for the conjecture `C1`. *RRL* automatically backtracks whenever an induction proof attempt based on a scheme fails. The conjecture is retried with a different induction scheme. Attempts at proving a conjecture are abandoned by *RRL* once all the available schemes have been tried or at a user request.

Heuristics for choosing an induction scheme

Several heuristics are implemented in *RRL* to choose the most appropriate induction scheme from the given possible schemes. These are based on the heuristics developed for the prover *Nqthm*, described in [3]. Two heuristics that are frequently employed in the verification of arithmetic circuits are *subsumption* and *merging*.

Subsumption chooses an induction scheme with a common set of induction variables. An induction scheme ϕ subsumes an induction scheme ψ if the substitutions for the induction variables in ϕ refine those in ψ. For example, the induction scheme suggested by `rca` subsumes that suggested by `bton, len` with respect to the induction variable `y` or the induction variable `z`. The function `rca` is defined in terms of bit vectors of size 0, 1 and those with size greater than 1 whereas the functions `bton` and `len` are defined in terms of bit vectors of size 0 and those with size greater than 0.

If a scheme ϕ subsumes a scheme ψ, the heuristics in *RRL* always choose the scheme ϕ. This is because choosing a scheme ψ leads to an induction case whose conclusion cannot be simplified using the definitions since the rules are more refined than the conclusion, and hence will not match the conclusion. We describe below how the scheme suggested by `rca` is chosen by these heuristics while proving the conjecture `C1`.

A proof attempt of `C1` based on the scheme suggested by `bton` or `len` fails for this reason. The subterm `rca(x, cons(u1, u), cons(v1, v))` cannot be simplified by the definition of `rca` as the rules in the definition are further

[4] Note that the third equation in the definition of `rca` cannot be applied to the conclusion. In order to apply this rule, it is necessary that `not(u = nil)` and `not(v = nil)`. These conditions do not follow from the assumptions of the step case.

CE[k] Would the word 'subgoals' be appropriate after 'subsequent' in line below?

Editor's or typesetter's annotations (will be removed before the final TEX run)

refined based on `u` and `v`. The case of `u = v = nil` is covered by the second rule and the third rule covers the case when these variables are not `nil`.

Using the subsumption heuristic, the schemes suggested by `bton` and `len` is discarded by *RRL* in favor of the scheme suggested by `rca`. This leads to a successful proof of the conjecture `C1` without backtracking.

Another heuristic implemented in *RRL* to pick an induction scheme is *merging.* Often the induction schemes suggested by the various subterms of a given conjecture share induction variables amongst each other. An inductive proof attempt of the conjecture based on one of these schemes only is not likely to succeed in such cases. For instance, if $t_1 = f(x,y)$ and $t_2 = g(z,x)$ are two subterms of a given conjecture where f and g are binary functions defined recursively on both of their arguments, then attempting a proof of the conjecture by induction based only on the scheme suggested by the term t_1 would result in an induction step with the conclusion containing $t'_1 = \sigma(f(x,y)) = f(\sigma(x),\sigma(y))$ and $t'_2 = \sigma(g(z,x)) = g(z,\sigma(x))$. The choice of induction scheme ensures that the term t'_1 can be simplified to match the induction hypothesis, but the same need not be true for the term t'_2 since the variable z in t'_2 does not get instantiated.

In such cases, the two competing induction schemes are merged into a single induction scheme that instantiates the induction variables simultaneously in all the terms. Merging reduces the number of alternative induction schemes, and also eliminates the need to arbitrarily choose from among competing schemes.

Coverset induction vs structural induction

In the cover set induction method, induction schemes are generated from the definitions of nonconstructor function symbols rather than being based on the constructors of the data type. Thus, different schemes can be generated for the same data type using the cover set method. Such schemes can involve nonconstructor symbols and can differ in the induction hypotheses to be used. In contrast, there is only one induction scheme corresponding to a data type in structural induction. In the cover set induction approach, each recursive call in the definition of a nonconstructor produces an inductive hypothesis, whereas only one induction hypothesis is generated by the structural induction approach. The capability of generating multiple induction schemes and additional hypotheses seems very beneficial in verifying properties of arithmetic circuits as discussed below.

Consider a ripple-carry adder `rcp` based on a divide-and-conquer representation. In this case, a ripple carry-adder of size n is recursively defined in terms of adders of size $n/2$. A divide-and-conquer representation of a ripple-carry adder is often used in hardware designs [39, 40] to construct arithmetic circuits of larger data widths by cascading similar smaller data width components. Smaller data width adders are often used as standard cells in industrial ASIC designs, and logic synthesis tools often provide these in their technology libraries.

A divide-and-conquer representation of a ripple-carry adder `rcp` can be used to exhibit the input-output equivalence of a ripple-carry and a carry-lookahead adder, which is much easier to specify using the divide-and-conquer strategy. We have found that such an equivalence proof is more easily done than the one in which a linear representation of the ripple-carry adder `rca` is used. The verification proofs of ripple-carry and carry-lookahead adders are discussed in more detail in [29]. Below, we illustrate how properties of a ripple-carry adder defined using the divide-and-conquer strategy can be verified easily using the cover set method, whereas proving these properties using structural induction is nontrivial.

The adder `rcp` can be specified in `RRL` as follows:

```
1. rcp(x, nil, nil) := cons(x, nil),
2. rcp(x,cons(y1,nil),cons(z1,nil)) :=
   cons(mod2(x, y1, z1),
   cons(half(x, y1, z1), nil)),
3. rcp(x,app(y1,y2),app(z1,z2))      :=
   app(sum(rcp(x, y1, z1)),
   rcp(carry(rcp(x,y1,z1),y2,z2)
   if {len(y1) = len(z1),
   len(z1) = len(y2), len(y2) = len(z2)}.
```

The first two equations defining `rcp` are identical to those defining `rca`. In the third equation, there are two recursive calls to `rcp` on the right side. The first adder performs addition of the corresponding lower halves of the input bit vectors and the second adder performs the addition of the upper halves. The function `app` denotes the appending of lists, and can be recursively defined in *RRL*. The functions `sum, carry` denote the outputs sum and carry of the first adder. The function `sum` is defined recursively as a list with the last element removed. The function `carry` is CE[1] defined recursively as the last element of a list.

The behavioral correctness of `rcp` can be stated in *RRL* as follows:

```
C2: bton(rcp(x, y, z))  == x + bton(y) + bton(z)
    if len(y) = len(z).
```

An inductive proof attempt of `C2` by structural induction over lists with induction variables `y, z` does not seem feasible. Such a proof attempt leads to two subgoals as before. The base case with `y = nil, z = nil` simplifies to true by contextual rewriting.
In the step case, the conclusion with `y = cons(u1, u), z = cons(v1, v)` is

```
bton(rcp(x,cons(u1,u),cons(v1,v))) ==
x + bton(cons(u1,u)) + bton(cons(v1,v)) if
(len(cons(u1, u)) = len(cons(v1, v))),
```

with the hypothesis being,

CE[1] 'in' eliminated from line below.OK?

Editor's or typesetter's annotations (will be removed before the final TEX run)

MS ID: STTTN032

31 May 2000 14:59 CET

```
bton(rcp(x, u, v)) == x + bton(u) + bton(v)
if len(u) = len(v).
```

The conclusion cannot be simplified as there is no rewrite that can match `rcp` whose second and third arguments are `cons` terms. The hypothesis cannot be applied to the conclusion, and therefore, the proof attempt fails. Additional inductions and/or case analysis are not likely to help in this case.

`C2` can, however, be established in *RRL* by the cover set induction scheme generated from the definition of `rcp`. Three subgoals are generated corresponding to the three equations in the definition of `rcp`. The first two equations lead to base cases, and are proved by contextual rewriting and case analysis as done for the conjecture `C1`. In the third subgoal, the conclusion is

```
[3] bton(rcp(x,app(y1,y2),app(z1,z2))) ==
    x + bton(app(y1,y2)) + bton(app(z1,z2)) if
    {(len(app(y1, y2)) = len(app(z1, z2)),
    len(y1) = len(y2), len(y2) = len(z1),
    len(z1) = len(z2))}.
```

Two hypotheses are generated by the cover set method from the two recursive calls of `rcp`. These are:

```
Hyp 1: bton(rcp(x, y1, z1)) ==
       x + bton(y1) + bton(z2) if
       {(len(y1) = len(y2), len(y2) = len(z1),
       len(z1) = len(z2))}.
Hyp 2: bton(rcp(carry(rcp(x,y1,z1)),y2,z2)) ==
       carry(rcp(x,y1,z1)) + bton(y2) + bton(z2)
       if {(len(y1) = len(y2), len(y2) = len(z1),
       len(z1) = len(z2))}.
```

Sixteen intermediate subgoals are generated by contextual rewriting corresponding to the each of the 5 clauses in the two hypotheses. 15 of these are established by contextual rewriting combined with the linear arithmetic decision procedure due to contradictory conditions over the function `len`.
The only remaining subgoal is

```
[3.16] bton(rcp(x,app(y1,y2),app(z1,z2))) ==
       x + bton(app(y1,y2)) + bton(app(z1,z2))
       if {len(app(y1,y2)) = len(app(z1,z2)),
       len(y1) = len(y2), len(y2) = len(z1),
       len(z1) = len(z2),
       bton(rcp(x,y1,z1)) =
       x + bton(y1) + bton(z2),
       bton(rcp(carry(rcp(x,y1,z1)),y2,z2)) =
       carry(rcp(x,y1,z1)) + bton(y2) + bton(z2))}.
```

This subgoal simplifies by the definition of `bton, rcp` to

```
[3.16] bton(app(sum(rcp(x, y1, z1)),
       rcp(carry(rcp(x, y1, z1)), y2, z2))) ==
       x + bton(app(y1, y2)) + bton(app(z1, z2))
       if {len(app(y1,y2)) = len(app(z1,z2)),
       len(y1) = len(y2), len(y2) = len(z1),
       len(z1) = len(z2),
       bton(rcp(x, y1, z1)) =
       x + bton(y1) + bton(z1),
       bton(rcp(carry(rcp(x,y1,z1)),y2,z2)) =
       carry(rcp(x,y1,z1)) + bton(y2) + bton(z2))}.
```

The subgoal is established in *RRL* by induction on `rcp` with the help of an additional intermediate lemma automatically generated by *RRL* by *generalizing* the subterm `rcp(x, y1, z1)` by a new variable. The intermediate lemma generated is

```
bton(app(sum(z), cons(carry(z), nil)) == bton(z).
```

The generation of this lemma using the generalization heuristic of *RRL* is discussed in detail in Sect. 6.1 on lemma generation.

5 Verifying properties of a family of circuits

In the previous section, we discussed the use of induction for verifying the number-theoretic properties of arithmetic circuits with arbitrary data widths. Besides the data width, arithmetic circuits are often parameterized in terms of

– the number of signals that they process in a single step
– the types of hardware components that they use.

We call this *component* parameterization to distinguish it from data width parameterization.

For example, most of the commonly used multiplier circuits employ the same algorithm but differ only in the number of partial sums considered at each step [28, 39, 40]. Similar parameterization of circuits is evident in the description of subtractive division algorithms such as the SRT [17]. In [17], the authors describe how four radix-2 iterations can be overlapped to perform one iteration of a radix-16 SRT division circuit. A radix-4 and radix-8 SRT divider can be similarly designed by overlapping two and three iterations of a radix-2 SRT divider, respectively.

Multiplier circuits use adder circuits as components. Different multipliers in the same chip may be designed using different types of adder circuits, such as carry-save adders, ripple-carry adders and carry-lookahead adders, even though they provide the same functionality. The choice of adder circuits is often dependent on timing and area considerations [39]. Division algorithms are also parameterized based on the types of hardware adders and subtractors that they use. Based on the representation used for the partial remainder in each step, a carry-save adder or a ripple-carry adder may be used. A carry-save adder may be used to keep the partial remainder as a pair of vectors, denoting the sum and carry; otherwise a ripple-carry adder may be used [17]. The use of carry-save adder reduces the time required to compute the partial sum.

Typically, each such circuit has to be specified individually. Even though these circuits are related, their verification has to be redone all over again. Commonality in the circuit structure leading to commonality in verification proofs is not exploited. We believe this repetitive effort can be avoided by parameterizing arithmetic circuits such as multipliers and dividers based on the number of signals considered at each step as well as based on the hardware components used.

By exploiting these parameters in addition to data width, a common, hierarchical top-level specification of a family of related circuits can be developed. A proof can be generated for the whole family using such specifications. The hardware components in these circuits are abstracted in terms of their behavioral constraints. Individual instances of the components can be separately verified against these constraints. This approach avoids redoing the proof for each instance of the circuit, as is the case when BDD-based tools are used.

We illustrate this methodology for multiplier circuits below.

5.1 Generic specification and verification: a family of multiplier circuits

Most of the commonly used multiplier circuits are based on the CE^{m} grade school principle of multiplying any two given n-bit numbers consisting of two steps:

– computing the partial sums
– adding the partial sums to obtain the required result.

This basic underlying principle is often not evident in commonly found descriptions of these circuits. The computation of partial sums is done in the same manner in these circuits, but they differ in the number of partial sums that they consider for addition at any particular time.

For example, a linear array multiplier performs the multiplication of two n-bit numbers in linear time. The n partial sums are first obtained in constant time. Then, at each step one partial sum is added to the partial result computed so far.

Wallace introduced a new scheme for multiplying two n bit numbers in logarithmic time [50], which has popularly come to be known as the Wallace tree multiplier. Improved performance is achieved in this multiplier by considering three partial sums for addition together.

The multiplication scheme due to Wallace was generalized and improved upon by Dadda in [15] leading to a rich family of multipliers called the *Dadda multipliers*. In these multipliers, larger than three partial sums are taken up for addition at a particular time. Considering larger numbers of partial sums does not improve the asymptotic complexity but considerably reduces the number of stages required for multiplication resulting in reduced wiring delays. The 7-3 multiplier used in *IBM RS/6000* is based on this observation, and has been attributed [37] as one of the important features that contributes to its good performance.

A common top-level specification for this whole family of multiplier circuits using the common algorithm can be given. A multiplier circuit is abstracted in terms of two components: a component that computes the partial sums, called the *partial sum computation* component and another that adds these partial sums to compute the final product, called the *partial sum addition* component. We then describe a uniform approach for mechanically verifying properties of any multiplier in the family using *RRL*. It is shown that the correctness of any multiplier circuit in the family can be mechanically established from the behavioral correctness of the partial sum computation component and that of the partial sum addition component. The correctness of these components follow from the correctness of the adder circuits used in them.

In our specifications and proofs of various multiplier circuits, we abstract the adders in terms of generic hardware components with associated behavioral constraints. The correctness of a multiplier circuit is first established in terms of such generic components. It is shown later how a particular adder can realize a generic component by demonstrating that the adder satisfies the behavioral constraints of the generic component. Such a view provides a clear separation between specification and implementation aspects. The use of generic components aids the reuse of proofs and modularizes the correctness proofs, allowing verification to go hand in hand with the design process in a hierarchical fashion. Such modularization of proofs is crucial for any verification methodology to effectively scale up to larger and more complex hardware circuits.

The proposed approach is highly generic – it not only abstracts over the word size of multiplier circuits but also abstracts the common behavior of a variety of different multiplier circuits. The proofs of correctness are obtained for multiplier circuits of arbitrary word size. Second, seemingly different multiplier circuits share a common specification and a common proof of correctness using the same lemmas, with only a few different definitions for each multiplier circuit.

A major complaint against the use of theorem provers and proof checkers for hardware verification has been the semi-automatic nature of these systems. Verification efforts involve considerable user ingenuity. We believe that a common top-level proof for a family of multiplier circuits with well-characterized intermediate lemmas that are independent of the underlying prover, is a step forward in addressing this issue. It is also shown that the intermediate lemmas needed in the proofs of correctness of multipliers reported here correspond to formulas that specify the input output behavior expressed in terms of numbers, of different components of the circuits. Such lemmas can be generated systematically from the structure of the circuits as discussed in a later section.

5.2 Specifying a family of multiplier circuits

A common, top-level equational specification for the family of multiplier circuits is described. We first review how a popular multiplier circuit – the Wallace tree circuit –

CE^{m} Hyphen added to 'n bit' in lines below, OK?

Editor's or typesetter's annotations (will be removed before the final TEX run)

can be specified in *RRL*. A common top-level *RRL* specification for a family of multipliers is then described. From this generic specification, specifications of linear array multiplier and 7-3 multiplier can be generated by instantiation.

The overall structure of the Wallace tree multiplier can be described diagrammatically as in Fig. 1.

Given bit vectors x and y of equal length, a Wallace tree circuit first computes a list of partial sums ($P1, \cdots, P8$ for an 8-bit multiplier in Fig. 1) using a function such as *psum-all*. Each partial sum in the list is a bit vector that corresponds to a single bit of x, and is obtained by shifting y appropriately. The partial sums in the list are then added together by adding in parallel three partial sums at a time. Addition of any three partial sums is typically done using a carry-save adder (CSA) that has three bit vectors as its inputs and produces a pair of bit vectors as its output. The outputs correspond to the bitwise sum and the bitwise carry of the inputs[5]. The parallel addition of three bit vectors at a time is repeated on the outputs of the carry-save adders until there are only two bit vectors left. The final result is obtained using a ripple-carry (RCA in Fig. 1) or a carry-lookahead adder on these two bit vectors.

The computation of the partial sums can be achieved in constant time in parallel as the partial sums are independent of each other. The partial sum corresponding to a single bit z can be computed from the bit value, the bit position and y. While adding partial sums, at each level, $\lfloor n/3 \rfloor$ carry-save adders in parallel convert the addition of n partial sums to the addition of $\lceil 2n/3 \rceil$ partial sums resulting in a tree with depth bounded by $log(n)$.

Specifying Wallace tree partial sum computation in RRL. Partial sums in multiplier circuit are modeled in `RRL` as a list of bit vectors with `lnil` denoting the empty list of bit vectors and `consl` that adds a bit vector to a list of bit vectors[6].

The partial sum `psum` corresponding to a single bit `x1` of `x` is the same as `y` if `x1` is `1`; otherwise, it is the zero bit vector of the same length as `y`.

```
psum(x1, y) := cond(x1 = 0, mkzero(y), y),
```

where `mkzero` generates a zero bit vector of the same length as its input.

The list of partial sums corresponding to all the bits of `x` is computed by applying the function `psum` point-wise to each bit of `x` and shifting `y` to the right by appending a *trailing* zero.

```
psum-all(nil, y) := lnil,
psum-all(cons(x1, x), y)  :=
consl(psum(x1, y), psum-all(x, cons(0, y))).
```

Specifying Wallace tree partial sums addition in RRL. In a Wallace tree circuit, each level in the tree in Fig. 1 contains a list of bit vectors that have to be added to produce the final result. The root contains the list of

[5] Further details on the specification of the carry-save adder are given in Sect. 5.4.

[6] Recall that contrary to the usual convention, we assume that the bits increase in order from left to right in a bit vector i.e., the bit vector 01 stands for $0 * 2^0 + 1 * 2^1 = 2$.

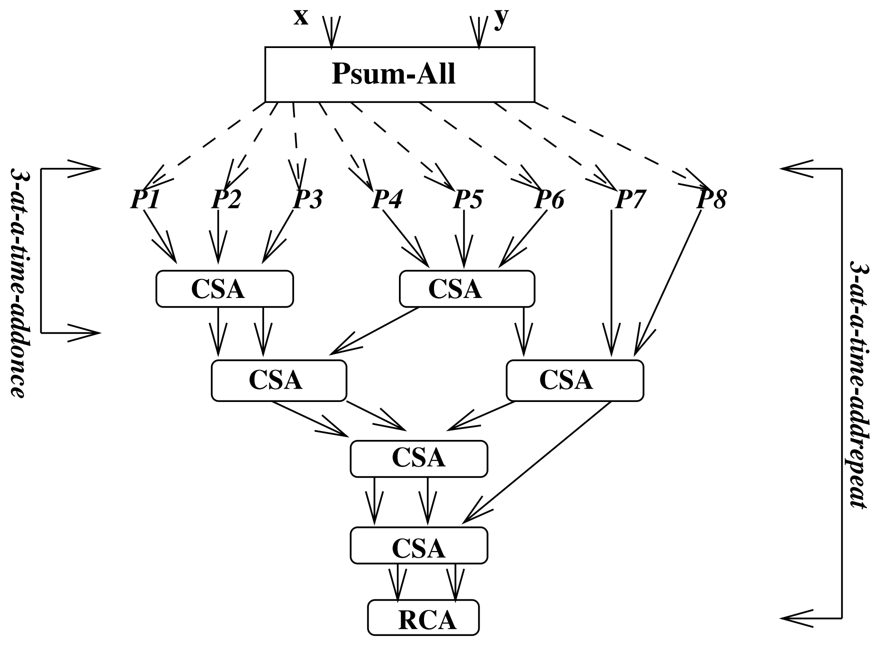

Fig. 1. Wallace tree multiplier

partial sums corresponding to each bit of the bit vector x. The successive levels of the tree are repeatedly constructed until there are less than three bit vectors at any given level(equations 1, 2 and 3 below). In the case of two bit vectors, addition using a ripple-carry adder rca, is performed(equation 3 below).

The Wallace tree multiplier is specified by 3-mult below. The trace of a computation of 3-mult on input vectors of a specific length corresponds to a specific circuit:

```
1. 3-mult(cons(x1,nil), y) := psum(x1, y),
2. 3-mult(cons(x1, cons(x2, nil)), y) :=
   rca(0, pad(1,psum(x1,y)),
   psum(x2,cons(0,y))),
3. 3-mult(cons(x1,cons(x2,cons(x3,x))),y) :=
   3-repeat(psum-all(cons(x1,cons(x2,
   cons(x3,x))),y)).
```

The function 3-repeat repeatedly takes 3 bit vectors and adds them; it is specified as

```
1. 3-repeat(lnil)                    := nil,
2. 3-repeat(consl(x1,lnil))          := x1,
3. 3-repeat(consl(x1,consl(x2,lnil))) :=
   rca(0,pad(1,x1),x2)
   if len(pad(1,x1)) = len(x2),
4. 3-repeat(consl(x1,consl(x2,consl(x3,x)))) :=
   3-repeat(3-once(consl(x1,consl(x2,
   consl(x3,x))))),
```

where len denotes the length of a list (of bit vectors). The function pad(m, x) produces a bit vector by appending m *leading* zeroes to the bit vector x. Bit vectors are typically padded by leading zeroes in these specifications so that the input bit vectors to the adders are of equal length. The last equation (equation 4) computes the bit vectors at the successive level by the function 3-once.

The function 3-once is defined on a list of bit vectors. If the input list contains less than three bit vectors(equations 1, 2 and 3 below), then the bit vectors in the input list are carried over to the output list. Otherwise, the bit vectors in the input list are taken in groups of three, and added in parallel using a carry-save addercsa, (equation 4 below)[7]. The outputs of the carry-save adder and the bit vectors in the input list that were not considered for addition together constitute the bit vectors of the output list.

```
1. 3-once(lnil)                      := lnil,
2. 3-once(consl(x1, lnil))           :=
   consl(x1, lnil),
3. 3-once(consl(x1, consl(x2, lnil)))  :=
   consl(x1, consl(x2, lnil)),
4. 3-once(consl(x1,consl(x2,consl(x3,x)))) :=
   consl(fst(z1),consl(snd(z1), 3-once(x)))
   if {(z1 = csa(pad(2, x1), pad(1, x2), x3)),
   len(pad(2, x1)) = len(x3),
   len(pad(1, x2)) = len(x3)}.
```

[7] As said earlier, a carry-save adder takes three bit vectors of equal length as inputs, and produces two bit vectors of equal length as outputs. One output denotes the position-wise sum of the inputs, and the other denotes the position-wise carry of the inputs. For more details on carry-save adders the reader may refer to the end of this section.

A generic specification

The above specification of the Wallace tree multiplier can be generalized to specify any multiplier of the above family. A generic specification of a multiplier k-mult in which k (k >= 1) partial sums are added together at any time can be given. The specification of k-mult is described below. It uses generic hardware components gsimple-mult for multiplier and gsimple-adder and gkogk-adder for adder circuits. The specification of the linear array, Wallace and the 7-3 multiplier can be obtained by instantiating k to 1, 3, and 7, respectively. The generic hardware components are instantiated by multipliers and adder circuits with the appropriate data width. The complete specification of the linear-array and the 7-3 multiplier are available by anonymous ftp via *ftp.cs.albany.edu/pub/subu/Multipliers*.

The k-mult multiplier is abstracted in terms of a partial sum computation component psum-all and partial sum addition component k-repeat. k-mult is specified in *RRL* as follows.

```
k-mult(x,y,k) := cond(len(x) < k,
                 gsimple-mult(x,y,k),
                 k-repeat(k,psumall(x,y))) if
                 (len(x) = len(y)).
```

The function k-mult takes two equal length bit vectors x, y and the parameter k as its inputs. If the size of the input bit vectors is less than k, then a generic multiplier gsimple-mult is used. Otherwise, the partial sums are generated from x, y using the partial sum computation component psum-all, as before. The generated partial sums feed into the partial sum addition component k-repeat, which repeatedly adds k partial sums at a time. The partial sum computation component is specified just as in the case of the Wallace tree multiplier using the functions psum and psum-all. It does not depend upon the parameter k.

The partial sum addition component k-repeat has k as a parameter. The parameter determines the number of partial sums added at each step. This component is recursively specified in RRL using the function k-once that adds k partial sums at one level until there are fewer than k partial sums left. The functions k-repeat and k-once are analogous to the functions 3-repeat and 3-once used in the case of Wallace tree multiplier above.

```
k-repeat(k, y)              :=
gsimple-adder(y) if (lenlst(y) < k),
k-repeat(k, applst(y1, y)) :=
k-repeat(k, k-once(k, applst(y1, y))) if
(lenlst(applst(y1, y)) >= k),
k-once(k, x)                :=
x if (lenlst(x) < k),
k-once(k, applst(x1, x))   :=
applst(gklogk-adder(x1, k), k-once(k, x)) if
{(lenlst(applst(x1, x)) >= k), (lenlst(x1) = k)}.
```

The function `lenlst` computes the length of a list of bit vectors. The function `applst` takes two lists of bit vectors as its inputs and appends the first input list of bit vectors to the second list in the front to produce a list of bit vectors. Recall that a bit vector is itself represented as a list of bits.

The functions `gsimple- adder` and `gklogk- adder` above are specified as generic hardware adder components. The adder `gsimple-adder` takes a list of bit vector as inputs and produces a bit vector as output which is the result of adding the input bit vectors. The adder `gklogk- adder` takes `k` bit vectors as inputs and produces `log(k)` (rounded to the next highest integer) as its outputs, bit vectors corresponding to the bitwise sum and bitwise carry(s) of the input bit vectors. Adding k bits can result in a carry of size at most `log(k)`. For instance, for `k = 3`, the adder `gklogk-adder` can be realized by a carry-save adder that takes 3 bit vectors as inputs and produces 2 bit vectors as its outputs – a bit vector of bitwise sums and a bit vector of carrys.

As shown below, these generic components are specified abstractly only in terms of behavioral constraints; no definitions, implementation or realizations of their behavior are provided. Behavioral constraints must be subsequently checked from definitions when a generic component is instantiated. *RRL* automatically checks that the instantiations satisfy constraints by establishing the constraints as theorems from these definitions. The use of such generic components enables hierarchical structuring and reuse of correctness proofs.

5.3 *Verifying properties of multipliers in RRL*

We discuss how properties of multiplier circuits can be automatically verified using *RRL*. An important aspect of this verification is the generation of intermediate lemmas needed. In this section, we discuss how these lemmas are used in proofs. In a later section, we discuss how such lemmas can be automatically generated from circuit structure and specifications of components in a circuit using heuristics.

Verification of the Wallace tree multiplier circuit is presented first. Subsequently, we outline how a correctness proof with the same top-level structure as that of the Wallace tree multiplier can be generated for any multiplier circuit in the family.

To establish the correctness of these circuits with respect to multiplication over numbers, conversion functions from bit vectors and lists of bit vectors to numbers are needed. In addition to the function `bton` described in the previous section on adder circuits, the function `btonlist` is used to convert bit vectors to numbers. Given a list of bit vectors as input, the function `btonlist` performs a linear addition of numbers corresponding to each of the bit vectors.

```
btonlist(lnil)  :=  0,
btonlist(consl(x, y)) := bton(x) + btonlist(y).
```

Verification of the Wallace multiplier

The main theorem expressing the correctness of the Wallace tree multiplier is stated in *RRL* as follows:

```
C3: bton(3-mult(x, y)) == bton(x) * bton(y)
    if (len(x) = len(y)).
```

The equations defining functions `psum, psum-all`, `3-once`, `3-repeat` and `3-mult` are first oriented by *RRL* into rewrite rules[8]. The cover set corresponding to each of these functions is computed.

The theorem `C3` is proved in *RRL* by induction. The subterms `bton(x)`, `bton(y)` and `3-mult(x, y)` suggest three possible induction schemes. The induction suggested by the subterm `3-mult(x, y)` *subsumes* the one suggested by the subterm `bton(x)` since the left sides of the rules in the definition of `3-mult` refine the left sides of the rules in the definition of `bton`. The scheme based on the definition of the function `3-mult` is chosen to be the most appropriate one automatically by *RRL*. Here is the transcript generated in *RRL*.

```
Let P(x) be bton(3-mult(x, y)) == bton(x) * bton(y)
           if (len(x)  = len(y))

Induction will be done on x in 3-mult(x, y),
with the scheme:
[1] P(cons(x1, nil))
[2] P(cons(x1, cons(x2, nil)))
[3] P(cons(x1, cons(x2, cons(x3,  x))))
```

The subgoal corresponding to `[1]` is established by case analysis based on the definition of `psum`, using the definitions of `3-mult` and `bton` for simplification. The case analysis is automatically recognized by *RRL* based on the definition of `psum` given in terms of the ternary predicate `cond`.
The subgoal corresponding to `[2]` simplifies by the second rule in the definition of `3-mult` to

```
bton(rca(0,pad(1,psum(x1,y)),
psum(x2,cons(0,y)))) ==
bton(cons(x1,cons(x2,nil))) * bton(y).
```

After simplification, the above formula suggests the following lemma, which ensures that the ripple-carry adder correctly implements addition over numbers:

[8] All the equations except the last equation in the definition of `3-repeat` can be oriented left to right using term orderings such as *lrpo* implemented in *RRL*. In order to orient the last equation defining `3-repeat`, a semantic argument that the number of partial sums to be added decreases after one-level addition of partial sums by `3-once` has to be used. Such an argument can be proved by induction in *RRL*.

```
L1: bton(rca(x, y, z))  == bton(x) + bton(y)
    if len(y) = len(z).
```

This lemma is the same as the theorem C1 discussed in the previous section, and it can be directly proved in *RRL* by induction as discussed there. Using this lemma, by case analysis on the definition of the function psum, the subgoal [2] is easily established in *RRL*.
The subgoal [3], simplifies by the definition of 3-mult to

```
bton(3-repeat(psum-all(cons(x1,
cons(x2, cons(x3, x))), y)))) ==
bton(cons(x1,cons(x2,cons(x3,x)))) * bton(y)
if len(cons(x1,cons(x2,cons(x3,x)))) = len(y).
```

Similar to the subgoal [2], the above formula on simplification suggests the following lemma relating the number corresponding to the output of the partial sum addition component to the number corresponding to the input list of bit vectors to this component.

```
L2:  bton(3-repeat(x)) == btonlist(x).
```

Using L2, the subgoal [3] is simplified to:

```
btonlist(psum-all(cons(x1,cons(x2,
cons(x3,x))),y)) ==
bton(cons(x1,cons(x2,cons(x3,x)))) * bton(y)
if len(cons(x1,cons(x2,cons(x3,x)))) =
len(y)).
```

Just as in the case of lemmas L2 and L1, the above formula on simplification suggests an intermediate lemma L3 below. This lemma L3 relates the number corresponding to the linear addition of the bit vectors output by the partial sum component to the product of the numbers corresponding to the two bit vectors input to this component.

```
L3: btonlist(psum-all(x, y)) == bton(x) * bton(y).
```

The subgoal [3] follows from this lemma by rewriting. As a result, the main theorem C3 is established by induction.

Lemmas L2 and L3 are proved in *RRL* again using the cover set induction method. In each of these proofs, the appropriate induction scheme leading to a successful proof was automatically chosen by *RRL*.

The proof of the lemma L3 is established by induction based on the definition of psum-all followed by case analysis on psum. Here is a part of the transcript generated in *RRL*.

```
Let P(x, y) be btonlist(psum-all(x, y)) ==
               (bton(x) * bton(y))
The induction will be done on X, Y in
psum-all(x, y), with the scheme:
[1] P(nil, y)
[2] P(cons(y1, y), y1) if {P(y, cons(0, y1))}
```

The first subgoal [1] trivially follows from the definitions of psum-all, bton and that of *. The second subgoal [2] simplifies to the formula:

```
bton(y2) * bton(cons(y1,y)) ==
bton(psum(y1,y2)) + bton(y) * bton(y2) +
bton(y) * bton(y2),
```

which is reduced to true by case analysis on psum and the definition of bton.

Lemma L2 was proved in *RRL* by induction based on the definition of 3-repeat. Here is the transcript generated in *RRL* for proving this lemma,

```
Let P(x) be btonlist(x) == bton(3-repeat(x))
Induction will be done on x in 3-repeat(x),
and will follow the scheme:
[1] P(lnil)
[2] P(consl(x1, lnil))
[3] P(consl(x1, consl(x2, lnil))) if
    {(len(app(x1, cons(0, nil))) = len(x2))}
[4] P(consl(x1,consl(x2,consl(x3, x)))) if
    P(3-once(consl(x1,consl(x2,consl(x3,x)))))
```

The subgoals [1], [2], [3] are easily established from the definitions of bton and btonlist using the lemma L1. In subgoal [4] the conclusion is

```
btonlist(consl(x1,consl(x2,consl(x3,x)))) ==
bton(3-repeat(consl(x1,consl(x2,consl(x3,x))))),
```

with the hypothesis being,

```
btonlist(3-once(consl(x1,consl(x2,consl(x3,x))))) =
bton(3-repeat(3-once(consl(x1,
consl(x2,consl(x3,x)))))).
```

The conclusion simplifies by the definition of 3-repeat and partial application of the hypothesis to

```
btonlist(consl(x1, consl(x2, consl(x3, x)))) ==
btonlist(3-once(consl(x1,consl(x2,consl(x3,x))))).
```

The above formula is automatically established by *RRL* by induction based on the scheme obtained from the cover set of the function 3-once. However, one of the induction subgoals in this proof requires another lemma L4 establishing the correctness of the carry-save adder with respect to addition over natural numbers. Lemma L4 and its proof are discussed in detail in Sect. 5.4.

A common top-level proof

The above proof for the Wallace multiplier can be made generic; the basic proof structure remains invariant when a proof of k-mult, a multiplier circuit that uses generic hardware components, is attempted. The correctness of a multiplier k-mult is stated in *RRL* as:

```
C4: bton(k-mult(x, y)) == bton(x) * bton(y)
    if (len(x) = len(y)).
```

The proof is similar to that of the Wallace multiplier discussed above using a similar set of lemmas about the behavior of components,

Linear addition of partial sums serves as a common denomination for any k, and the addition of k partial sums together can be reduced to linear addition. As in the case of the proof for the Wallace multiplier, this proof also involves characterizing the input-output behavior of the partial sum computation component and the partial sum addition component with respect to numbers, and then showing that cascading these two components leads to the desired overall behavior. It is shown that:

1. Multiplying the numbers corresponding to the input bit vectors of the partial sum computation component is the same as the number obtained by the linear addition of the list of partial sums output by this component
2. The number corresponding to the bit vector output by the partial sum addition component is the same as the number corresponding to the linear addition of the list of partial sums input.

The same strategy can be used to verify any multiplier in the family of multipliers (for any fixed `k`). Verification of other multipliers in the family of multipliers such as the linear array or the 7-3 multiplier can be performed in *RRL* using the same top-level proof. For instance, the verification of a linear array multiplier stated as

```
C5: bton(1-mult(x, y)) == bton(x) * bton(y)
    if (len(x) = len(y)),
```

is proved in *RRL* using three lemmas which are exactly the same as `L1` - `L3` with the lemma `L3` defined in terms of functions `1-repeat` instead of the function `3-repeat`. In order to establish lemma `L3`, it is necessary to verify the carry-save adder `L4` in this case also.

The correctness proof of the 7-3 multiplier in *RRL* also follows the same top-level proof using the lemmas `L1` - `L3` with the lemma `L3` defined in terms of the functions `7-repeat` instead of `3-repeat`. In a 7-3 multiplier, the main adder employed for adding partial sums has seven bit vectors as its input, and produces three bit vectors corresponding to the bitwise carry and the bitwise sum of the inputs. Just like the carry-save and the ripple-carry adders, this adder is verified separately in *RRL* to get a hierarchical verification proof.

5.4 Using generic hardware components

A multiplier circuit can be realized in a number of ways, based on different adder components used in it. Each of these realizations must be specified separately, and their correctness has to be established separately from the first principles. In this section we illustrate how such duplication of proof effort can be avoided by describing circuits in terms of *generic hardware components*, that enables hierarchically structured verification of circuits.

Abstracting adder circuits using behavioral constraints

In a verification proof of a multiplier circuit, it is sufficient to use the input-output behavior of adder circuits used in it. Other details of adder circuits are irrelevant. Adder circuits can be abstracted as generic hardware components satisfying behavioral constraints. The correctness proof of multiplier circuits can then be performed in terms of these generic hardware components. The generic hardware components are subsequently realized by specific adders that satisfy these behavioral constraints.

RRL has been extended to allow function instantiations and for handling theories (see also [6]). In addition to definitions and lemmas, *RRL* allows the user to introduce function symbols with behavioral constraints. For instance, a carry-save adder can be specified in *RRL* in terms of the generic component `g32-adder` (`g` in this name stands for generic, 32 is to signify that the component has 3 inputs and 2 outputs, and `adder` tells what the circuit does): as:

```
[g32-adder : list, list, list -> pairlst]
bton(fst(g32-adder(x, y, z))) +
bton(snd(g32-adder(x, y, z)))  ?=
bton(x) + bton(y) + bton(z) if
{(len(x) = len(y)), (len(y) = len(z))}.
```

The ripple carry adder and the adder employed in the 7-3 multiplier can be similarly abstracted in *RRL* in terms of the generic components `g31-adder` and `g73-adder`. Much like definitions which are distinguished from lemmas and properties by the use of the symbol `:=` instead of `==`, the symbol `?=` is used. Like all other (conditional) equations, constraints are oriented into terminating rewrite rules by *RRL* and are used for rewriting.

Different multipliers can be specified in *RRL* using generic hardware components. The use of such generic components in specifying the multipliers leads to shorter correctness proofs and reflects the common top-level structure of these proofs.

Realizing generic components: carry-save adder

A generic hardware component such as `g32-adder` above should be realized by a specific adder that satisfies the associated behavioral constraints. In this section, we use the correctness proof of a carry-save adder as an example to realize `g32-adder`. The other generic components used in the proofs of the multiplier circuits can be realized similarly using *RRL*.

The carry-save adder `csa` accepts three bit vectors of equal length as its inputs, and produces two bit vectors as its output, corresponding to the bitwise parity and the bitwise sum of its inputs. It is specified in *RRL* as

```
csa(x,y,z) :=  pairl(paritylst(x,y,z),
               cons(0,majoritylst(x,y,z))) if
               {(len(x) = len(y),
               len(y) = len(z))},
```

where `pairl` when given two bit vectors, constructs a pair of bit vectors. The function `paritylst`, given three input bit vectors of equal length, computes the bitwise parity of its inputs, and the function `majoritylst` computes their bitwise majority. The function `paritylst` is specified in *RRL* as follows:

```
paritylst(nil, nil, nil) := nil
paritylst(cons(x1,x),cons(y1,y),cons(z1,z)) :=
cons(mod2(x1,y1,z1), paritylst(x,y,z)) if
{len(x) = len(y), len(y) = len(z)}.
```

The function `mod2` is defined in the previous section. Recall that it produces a 0(1) based on the input being even or odd, respectively. The function `majoritylst` is defined in a similar fashion using the function `half` defined in the previous section.

The correctness of the carry-save adder with respect to addition over numbers can be stated in *RRL* as:

```
L4: bton(x) + bton(y) + bton(z) ==
    bton(paritytlst(x,y,z)) +
    bton(cons(0,majoritylst(x,y,z))) if
    {(len(x) = len(y)), (len(y) = len(z))}.
```

The above formula is proved directly in *RRL* by induction based on the definition of `paritylst`.

In order to realize the hardware component `g32-adder` by the carry-save adder `csa`, *RRL* provides an *instantiate* directive, which the user can invoke with a set of *function replacements* such as `((g32-adder csa),..)`. Based on these function replacements, the behavioral constraints are suitably instantiated by *RRL* and the instantiated formula is treated as a proof obligation which must be discharged from the properties of the realization. The instantiation directive implemented in *RRL* automatically discharges equational consequences using the combination of contextual-rewriting and built-in decision procedures. Proof obligations that need to be established by induction have to be explicitly presented by the user to *RRL*.

6 Heuristics for lemma generation

A key distinguishing feature of mechanizing proofs by induction from first-order theorem proving is that a proof by induction often involves the use of intermediate lemmas. These lemmas must CE[n] 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 CE[o] 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 CE[p] speculating intermediate conjectures which have been found effective in mechanizing verification of arithmetic circuits.

6.1 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 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 $*$ as needed for this proof. Also, one of the main reason *RRL* succeeds, is because of the aggressive use of the generalization heuristic, and the backtracking facility which enables *RRL* to attempt different generalizations, when an intermediate conjecture cannot be proved. We discuss these features below.

Most induction theorem provers support heuristics for generalizing conjectures. In *RRL*, at least two kinds of generalizations are performed:

1. Abstracting a nonvariable subterm appearing in a conjecture to be a variable, if the nonvariable subterm appears more than once in the conjecture (in the left side as well as the right side and/or the condition)
2. Dropping an assumption from a conditional conjecture.

Semantic analysis 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. In this section we illustrate the use of generalization based on syntactic analysis; for a discussion about a generalization heuristic based on semantic analysis, the reader may consult [27, 46].

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. Proof attempts on the original conjecture may not succeed if all induction schemes are flawed [3, 46]. Generalizing a subterm may result in new schemes some of which may be unflawed. A judicious application of the generalization heuristic based on whether a particular generalization is likely to result in unflawed induction schemes appears to be a good way to filter out many of the generalizations and avoid over-generalization.

Given a conjecture C of the form $l = r$ *if cond*, we look for a maximal nonvariable subterm s occurring in at least two of l, r and *cond*. Since there may be many such maximal subterms, they are abstracted to distinct variables in some order governed by heuristics. It is obvious that if a generalization C_g can be proved, then C follows from it (by rewriting). The backtracking mechanism comes in handy for this purpose. A generalization is attempted and if it turns out to be invalid, (i.e., a counter

CE[n] Could 'speculated' be replaced by 'assumed' or similar in line below?
CE[o] 'speculating'. Change to 'assuming'?
CE[p] 'speculate', 'speculating' seem to be being used in an unusual manner in the following secions. Is this a technical use, or can they all be replaced by 'assume', 'assuming'

example is generated) or a proof cannot be generated, the theorem prover backtracks and tries a different generalization. At the end, if no generalization can be proved, then the original conjecture C is attempted. It thus becomes very important that an invalid generalization is detected quickly, and here the role of examples to test a generalization becomes critical as a way to discard obviously invalid generalizations.

For example, consider the following intermediate subgoal generated in establishing the correctness of the ripple-carry adder `rcp`[9].

```
[3.16] bton(app(sum(rcp(x, y1, z1)),
       rcp(carry(rcp(x, y1, z1)), y2, z2))) ==
       x + bton(app(y1, y2)) + bton(app(z1, z2))
       if {(len(app(y1, y2)) = len(app(z1, z2)),
       len(y1) = len(y2), len(y2) = len(z1),
       len(z1) = len(z2),
       bton(rcp(x, y1, z1)) =
       x + bton(y1) + bton(z1),
       bton(rcp(carry(rcp(x,y1,z1)), y2,z2)) =
       carry(rcp(x, y1, z1)) +
       bton(y2) + bton(z2)).
```

The above subgoal is established in *RRL* by the cover set method using an induction scheme generated from the definition of `rcp`. The induction variables chosen are `y2`, `z2`. The induction proof leads to three subgoals. The intermediate lemma `I1` is generated by *RRL* to establish the first base case.

The first base case with `y2` = `z2` = `nil` simplifies by the definition of `rcp` to

```
[3.16.1] bton(app(sum(rcp(x, y1, z1)),
         cons(carry(rcp(x, y1, z1), nil)))) ==
         x + bton(y1) + bton(z1) if
         {(len(y1) = 0, len(z1) = 0,
         bton(rcp(x,y1,z1)) =
         x + bton(y1) + bton(z1))}.
```

The subterm `rcp(x, y1, z1)` occurs on the left-hand side of the subgoal `[3.16]` and in the conditions governing the subgoal. This subterm is replaced by a new variable in `[3.16]` to generate a new generalized conjecture.

```
[G1] bton(app(sum(Z1), cons(carry(Z1), nil))) ==
     bton(Z1) if {len(y1) = 0, len(z1) = 0}.
```

The conjecture `G1` is further generalized by *RRL* by dropping the conditions `len(y1) = 0` and `len(z1) = 0`. The generalization heuristics in *RRL* drop conditions that do not involve variables appearing in the body of the conjecture. The resulting conjecture is:

```
[I1] bton(app(sum(Z1), cons(carry(Z1), nil)))
     == bton(Z1).
```

This generalized conjecture is established in *RRL* by induction using the definition of `sum`. The induction scheme generated is the same as the structural induction over lists.

[9] A correctness proof of `rcp` is discussed in detail in Sect. 4.4

The other two subgoals in the induction proof of the subgoal `[3.16]` are established using this intermediate lemma and the definitions of the function symbols `rcp`, `len`, `app`, `bton`, `sum` and `carry`.

As stated above, if there are multiple subterms repeatedly occurring in a conjecture, then the generalization heuristics in *RRL* generate a new conjecture by abstracting maximal non-overlapping subterms simultaneously. If a generalized conjecture cannot be proved, *RRL* backtracks and generates progressively less general conjectures by abstracting fewer subterms. The process terminates once a generated conjecture is established or if there are no more subterms to abstract. In the latter case, the original conjecture is attempted without attempting any generalization.

In order to avoid reckless generalization, a subterm (or subterms) being generalized is matched against the data base of properties already proved. If the subterm appears in the left side of an unconditional rule that is some theorem already proved or a part of a definition, then the generalization mechanism uses this unconditional rule to generate a *constraint* on the new variable being introduced. For instance, if s is a subterm being generalized to a variable u and there is an unconditional rule $l \rightarrow r$ in which s appears in l at position p, then a constraint $l' = r$ is added to the generalized conjecture, where l' is obtained from l by replacing s with u at position p.

6.2 Speculating conjectures guided by use of induction hypothesis

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 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 [30]. Constraints are generated from the simplified conclusion and the induction hypothesis, and aided by the definitions and known properties of function symbols appearing in the conjecture. These constraints are also analyzed to speculate the missing side of an intermediate conjecture.

This approach has turned out to be quite effective in proving properties from tail-recursive definitions. We illustrate some of the key steps of this method on a simple property of a ripple-carry adder `rca`.

Consider proving the following conjecture about `rca` that states that adding a binary number whose least significant bit is 0 to another binary number using `rca` CE^q is equivalent to adding its half twice.

```
C6: bton(rca(x, y, lftshft(0, z))) ==
    bton(rca(0, rca(x, y, z), pad(1, z)))
    if {(len(y) = len(lftshft(0, z)),
    len(rca(x, y, z)) = len(pad(1, z)))},
```

The function `lftshft` takes a bit and a bit vector as its inputs, and produces a bit vector that is shifted left by one position using the input bit. The function `pad`, described earlier, pads a given bit vector with the given number of leading zeroes. The functions `lftshft` and `pad` are defined in *RRL* as

```
lftshft(x, y) := cons(x, y),
pad(0, z) := z,
pad(s(x), z) := pad(x, app(z, cons(0, nil))).
```

Conjecture `C6` simplifies by the definitions of `lftshft` and `pad` to

```
bton(rca(x, y, cons(0, z))) ==
bton(rca(0, rca(x, y, z),
app(z, cons(0, nil)))) if
{len(y) = add1(len(z)),
len(rca(x,y,z)) = len(app(z,cons(0,nil)))}.
```

The proof of this formula is done in *RRL* by the cover set method based on the induction scheme generated from the definition of `rca`. The induction variables chosen are `y, z`. Two base cases and one induction step case are generated.

The base cases are trivially established due to contradictory assumptions over `len`. In the first base case the first conjunct `len(nil) = add1(len(nil))` reduces to false. Similarly, in the second base case the first conjunct `len(cons(x1,nil)) = add1(len(cons(x1, nil)))` reduces to `add1(0) = add1(add1(0))` which reduces to false.
In the step case, the conclusion is

```
bton(rca(x,cons(y1,y),cons(0,cons(z1,z)))) ==
bton(rca(0,rca(x,cons(y1,y), cons(z1,z))),
app(cons(z1,z), cons(0, nil))) if
{(len(y) = add1(len(z)),
len(rca(x,cons(y1,y),cons(z1,z))) =
len(app(cons(z1,z), cons(0, nil)))).
```

The hypothesis is

```
bton(rca(half(x,y1,z1),y,cons(0,z))) ==
bton(rca(0,rca(half(x,y1,z1),y,z),
app(z,cons(0,nil)))) if
{len(y) = add1(len(z)),
len(rca(half(x,y1,z1), y, z)) =
len(app(z, cons(0, nil)))}.
```

The conclusion simplifies by the definitions of `rca, len, bton` and `app` to

```
mod2(x, y1, 0)  +
2 * bton(rca(half(x, y1, 0), y, cons(z1, z))) ==
mod2(0, mod2(x, y1, z1), z1) +
2 * bton(rca(half(0, mod2(x, y1, z1), z1),
rca(half(x,y1,z1),y,z),
app(z,cons(0,nil)))) if
{len(y) = add1(len(z)),
len(rca(half(x,y1,z1),y,z)) =
len(app(z,cons(0,nil)))}.
```

Further, simplification using the definitions of the function symbols `mod2, +`, and linear arithmetic leads to the subgoal[10]

```
I2: bton(rca(half(x,y1,0),y,cons(z1,z))) ==
    bton(rca(half(0, mod2(x, y1, z1), z1),
    rca(half(x, y1, z1), y, z),
    app(z, cons(0, nil))))
    if {len(y) = add1(len(z)),
    len(rca(half(x,y1,z1), y, z)) =
    len(app(z, cons(0, nil)))}.
```

The induction hypothesis cannot be applied to the conclusion. Proof attempts using other induction schemes result in similar failure. So the heuristic to speculate intermediate conjectures guided by induction hypotheses is invoked as described below.

First, positions in `I2`, where either side of the hypothesis can be potentially applied are determined, and the corresponding subterm of `I2` is equated with the appropriate side of the hypothesis.

The left and right sides of the hypothesis can be applied, respectively, to the left and right sides of `I2` at the root. This can be determined by analyzing the definitions of the function symbols `rca, bton` as discussed in [30, 46]. Two equations are generated corresponding to these potential applications of the hypothesis:

```
1: bton(rca(half(x, y1, 0), y, cons(z1, z))) ==
   bton(rca(half(x', y1, z1), y, cons(0, z))) if
   {len(y) = add1(len(z)),
   len(rca(half(x',y1,z1), y, z)) =
   len(app(z, cons(0, nil)))}.
2: bton(rca(half(0, mod2(x, y1, z1), z1),
   rca(half(x, y1, z1), y, z),
   app(z, cons(0, nil)))) ==
   bton(rca(0, rca(half(x', y1, z1), y, z),
   app(z, cons(0, nil)))) if
   {len(y) = add1(len(z)),
    len(rca(half(x',y1,z1), y, z)) =
    len(app(z, cons(0,nil))).}
```

Such equations are called *difference equations.* The primed variables in the difference equations are 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 the primed 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

[10] Recall that the functions `mod2` and `half` are Boolean functions. A case analysis on the values of `x, y1` and `z1` is invoked by `RRL` leading to 8 cases. All the cases reduce to the same subgoal `I2`.

a primed 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.

Decomposing equation 1 leads to failure resulting in the nontrivial equation `cons(z1, z) = cons(0, z)`. Decomposing equation 2 also leads to failure resulting in the nontrivial equation `0 = half(0, s(x, y1, z1), z1)`.

If the difference equations cannot be satisfied, the lemma generation heuristic identifies subterms in the hypothesis (which is a proxy for the conjecture) which if generalized, are likely to make solving difference equations feasible. In the example, a candidate subterm for generalization is `0`.

Abstracting `0` by a new variable `w` in `C6`[11] leads to the difference equations,

```
1: bton(rca(half(x,y1,w), y, cons(z1,z))) ==
   bton(rca(half(x', y1, z1), y, cons(w', z)))
   if {len(y) = add1(len(z)),
   len(rca(half(x',y1, z1) y, z)) =
   len(app(z, cons(0, nil)))}.
2: bton(rca(half(w, mod2(x, y1, z1), z1),
   rca(half(x, y1, z1), y, z),
   app(z, cons(0, nil)))) ==
   bton(rca(w', rca(half(x', y1, z1), y, z),
   app(z, cons(0, nil)))) if
   {len(y) = add1(len(z)),
   len(rca(half(x', y1, z1), y, z)) =
   len(app(z, cons(0, nil)))}.
```

Decomposing equation 2 leads to the substitutions `x' = x , w' = half(w, s(x, y1, z1), z1)`.
Applying these substitutions to the remaining difference equation 1 leads to the intermediate conjecture

```
I3: bton(rca(half(x,y1,w),y,cons(z1,z))) ==
    bton(rca(half(x,y1,z1), y,
    cons(half(w,mod2(x,y1,z1),z1),z))) if
    {len(y) = add1(len(z)),
    len(rca(half(x,y1,z1), y, z)) =
    len(app(z, cons(0, nil))))}.
```

Conjecture `I3` is automatically established by *RRL* by structural induction on the variable `y`, thereby, proving the conjecture `C6`.

Our work is similar in its motivation to that of [20, 22, 23] 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. The non-induction variables of a conjecture (called *sinks*) are exploited in that approach also. In rippling, it is assumed that a common term structure called the *skeleton* is shared by both the induction hypothesis and the conclusion. Meta level annotations called *wave fronts* are used to mark the differences between the hypothesis and the conclusion with regards to the skeleton. Similar annotations are associated with the rewrite rules (called *wave rules*), representing definitions and lemmas. Intermediate lemmas are generated as annotated equations by individually speculating each of the sides of the equation. Speculation typically involves starting with the skeleton embedded in second-order meta-variables denoting the missing wave fronts, and repeatedly simplifying using wave rules. The missing term structure is incrementally generated by unifying terms with second order meta variables after each simplification step. The approach uses higher-order unification, an expensive primitive operation, often leading to many useless paths.

Another related approach based on rippling is a critic [51] for handling diverging induction proof attempts. The critic is implemented in the theorem prover *SPIKE* and identifies accumulating term structures in successive induction subgoals by difference matching these subgoals [2], a technique for reconciling term annotations. Missing lemmas are heuristically speculated as wave rules that can aid in removal of this accumulating structure. The speculation of lemmas is based solely on the analysis of the proof attempts and does not exploit the structure of the rewrite rules.

The proposed approach, in contrast, is guided by heuristics to semantically match the hypothesis and the conclusion in a restricted fashion based on the structure of the available definitions and lemmas. We believe that it is simpler as additional annotations on rewrite rules are not needed and higher order unification for generating instantiations for term schemas as done in [20, 22, 23] is also avoided. Suitable instantiations are instead obtained by generating sufficiently many constraints on instantiations, and heuristically speculating ground instances using constraints. More constraints can be generated, if need be, depending upon how many resources a prover is interested in using in speculating conjectures.

6.3 Lemma speculation guided by circuit structure and homomorphisms

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 needed in the proof. A multiplier circuit in the family discussed above has four subcircuits. The first circuit `psum-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 `gsimple-adder` adds the bit vectors resulting from `k-repeat`. The fourth circuit, labeled `gklogk-adder`, is the adder used within `k-repeat` for adding `k` bit vectors used by the `k-repeat` component.

From the circuit structure, we have:

```
k-mult(x, y) == gsimple_adder(k-repeat
```

[11] If the subterm that is identified for generalization occurs on only one side of the conjecture, then a new conjecture is generated. The side in which the subterm does not appear is replaced by a term schema in the new conjecture. An instantiation for the schema is speculated while attempting to prove the new conjecture. The reader may refer to [30] for details.

```
(k, psum-all(x, y))) if
len(x) = len(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)
if len(x) = len(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, expressed using `bton` as discussed above. The specification for `gsimple-adder` is

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

The specification of the `gklogk-adder` component is

```
btonlist(gklogk-adder(x1, k)) = btonlist(x1)
if (lenlst(x1) = k).
```

The equation expressing the correctness of the multiplier unfolds to

```
bton(gsimple_adder(k-repeat(psum-all(x, y)))) ==
bton(x) * bton(y) if len(x) = len(y).
```

The above formula simplifies, using the specification of the adder component `gsimple-adder`, to

```
C7: btonlist(k-repeat(k, psum-all(x, y)))) ==
    bton(x) * bton(y) if len(x) = len(y).
```

This formula cannot be simplified any further. An attempt to prove this formula by induction fails, however, no matter what induction scheme suggested by any of the definitions of `psum-all, bton, len` is used. This can be found by analyzing the definitions of the function symbols `psum-all` and `k-repeat`.

Intermediate conjecture generation heuristic is invoked. Since the functions `bton` and `btonlist` are specified as homomorphisms, 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'(btonlist(Z)),
3. btonlist(k-once(k, Z))   == I'(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 the discussion in the previous section on a constraint-based approach for lemma speculation, constraints on the term schemas are generated using the definitions of the function symbols (using induction). ¿From these constraints, instantiations for the term schemas are speculated.

Let us consider conjecture 1. Two constraints are generated for the term schema `G'` from the definition of `applst`.

```
1.1 G'(0, btonlist(Z)) = btonlist(Z),
1.2 G'(bton(Z11) + btonlist(Z1), btonlist(Z)) =
    bton(Z11) + G'(btonlist(Z1), btonlist(Z)).
```

Substituting `Z1 = lnil` in constraint `1.2` as suggested by the definition of `btonlist` gives

```
1.3. G'(bton(Z11), btonlist(Z)) =
     bton(Z11) + G'(0, btonlist(Z)).
```

The above constraint simplifies, using constraint `1.1`, to

```
1.4. G'(bton(Z11), btonlist(Z)) =
     bton(Z11) + btonlist(Z)).
```

By generalizing the constraint `1.4`, the instantiation for the term schema `G'` is speculated to be

```
G'(x, y) = x + y.
```

We get the first intermediate conjecture to be

```
I4: btonlist(applst(x, y)) ==
    btonlist(x) + btonlist(y).
```

The conjecture `I4` can be easily established in *RRL* by coverset induction using the definition of `applst`.

Consider intermediate conjecture 2 now. From the definition of `k-repeat`, a constraint is generated for the term schema `H'`:

```
2.1. H'(btonlist(applst(Z1, Z))) =
     H'(btonlist(applst(gklogk-adder(Z1, k),
        k-once(k, Z)))).
```

Constraint `2.1` can be simplified using the lemma `I4` and the specification of the adder component `gklogk-adder` to

```
2.2 H'(btonlist(Z1) + btonlist(Z)))) =
    H'(btonlist(Z1) + btonlist(k-once(k, Z))).
```

Again, as suggested by the definition of `btonlist`, substituting `Z1 = lnil` in the above constraint gives

```
2.3 H'(btonlist(Z)) = H'(btonlist(k-once(k, Z))).
```

The constraint `2.3` cannot be further simplified. Using the definition of `k-once` with `Z = applist(Z11, Z12)` leads to a formula which is the same as the constraint `2.2`. However, since the root function symbols of the two sides of the constraint 2.3 are the same, the decomposition heuristic in *RRL* is used to equate the arguments, and generate additional constraints. The constraint generated by decomposition in this case is

```
I5:  btonlist(Z) == btonlist(k-once(k, Z)).
```

The conjecture `I5` is easily established in *RRL* by induction using the coverset of `k-once`.

This implies that the initial conjecture

```
2. btonlist(k-repeat(k, Z)) == H'(btonlist(Z)),
```

is independent of the instantiation chosen for the schema `H'`. The intermediate conjecture is speculated to be

```
I6: btonlist(k-repeat(k, Z)) == btonlist(Z).
```

The conjecture `I6` is proved in *RRL* using the intermediate lemmas `I4, I5` by coverset induction on the definition of the function symbol `k-repeat`.

The lemmas `I4, I5, I6` are all the intermediate lemmas that are needed to establish the correctness of the multiplier as shown below.

Going back to

```
C7: btonlist(k-repeat(k, psum-all(x, y)))) ==
    bton(x) * bton(y) if len(x) = len(y),
```

`C7` simplifies by lemma `I6` to,

```
btonlist(psum-all(x, y)))) ==
bton(x) * bton(y) if len(x) = len(y).
```

This subgoal is established in *RRL* by induction using the coverset of `psum-all`.

7 Finite tables and case analysis

Many hardware circuits, for example, SRT division, use a large number of preprogrammed constants, which are typically specified as finite tables. Implementations of arithmetic functions such as the square root and trigonometric functions are based on even much larger lookup tables [18, 44, 54]. Many of these implementations use multiple lookup tables [18, 54]. These tables are realized in hardware using PLAs (programmable logic arrays) [17, 39].

We discuss extensions to *RRL* for performing extensive case analysis typically arising in the verification of such circuits as well as for handling large finite tables. A hardware implementation of the radix 4 SRT division algorithm, the source of the now notorious Intel Pentium bug, is used for illustration.

A finite table can be specified in *RRL* as a predicate 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 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. To use the cover-set induction method to generate cases in proofs [53], it is necessary to relativize the original formula using a predicate specifying the subdomain of index values for which the table has the entries.

With either of these approaches, *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 [47]. In order for the verification to scale up to these applications, it is necessary that the underlying structure in tables be better exploited by the theorem prover *RRL*.

We introduce a special data structure for modeling finite tables. Partial tables are modeled using special *Dontcare* entries. In order to better exploit the structure of the tables we introduce *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.

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 [31]. 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.

Below, we first give an overview of the SRT division algorithm, and then briefly discuss its hardware implementation using a finite table for selecting quotient digit. Later we review a special data structure for finite tables, and show how it can be used to specify the quotient digit table of the radix 4 SRT division, and illustrate special reasoning mechanisms for the data structure in a proof of the main invariant property of the SRT division circuit.

7.1 SRT division algorithm

The SRT division algorithm [41, 47, 48] is an iterative algorithm for dividing a normalized dividend by a normalized divisor in which the quotient is computed digit by digit by repeatedly subtracting the multiples of the divisor from the dividend. The algorithm can be formalized in terms of the following recurrences about division in base (radix) r.

$$\begin{aligned}
P_0 &:= \mathit{dividend}/r, \quad Q_0 := 0,\\
P_{j+1} &:= r * P_j - q_{j+1} * D, \quad \mathit{for}\ j = 0, \cdots, n-1,\\
Q_{j+1} &:= r * Q_j + q_{j+1}, \quad \mathit{for}\ j = 0, \cdots, n-1,
\end{aligned}$$

where D is the positive divisor, P_j is the partial remainder at the beginning of the j-th iteration, and $0 \le P_j < D$, for all j, Q_j, is the quotient at the beginning of the iteration j, q_j is the quotient digit at iteration j, and n is the number of digits in the quotient. The bounds on the successive partial remainder $0 \le P_j < D$ guarantee the convergence of the algorithm.

SRT dividers used in practice incorporate several performance enhancing techniques while realizing the above recurrences. In particular, it is necessary to minimize the number of iterations and efficiently compute the quotient digit in each iteration. In radix 4 SRT division algorithm, the quotient digits are represented by a *redundant signed-digit* representation with digits in the range $[-2, 2]$. Tradeoffs between speed, radix choice and redundancy of quotient digits are discussed in [47]. Because of the redundancy, the bounds on the successive partial remainders for the convergence of the algorithm can be looser:

$$-D * 2/3 \leq P_j \leq D * 2/3.$$

By substituting the recurrence for the successive partial remainders, the range of shifted partial remainders, $4 * P_j$, that allow a quotient digit k to be chosen is:

$$[(k - 2/3) * D, (k + 2/3) * D].$$

The above relation between the shifted partial remainder range P and divisor D is diagrammatically plotted as a *P-D plot* given in Fig. 2. The plot gives the shifted partial remainder ranges in which a quotient digit can be selected, without violating the bounds on the next partial remainder. For example, when the partial remainder is in the range $[5/3D, 8/3D]$, the quotient digit 2 is selected. The shaded regions represent quotient digits overlaps where more than one quotient digits selection is feasible.

Redundancy in quotient digits allows the quotient digit to be selected based on only a few significant bits of the partial remainder and the divisor. As explained in [47], for a radix 4 SRT divider with the partial remainders and divisor of arbitrary width n, $n > 8$, it suffices to consider partial remainders up to 7 bits of accuracy and a divisor up to 4 bits of accuracy. This reduces the complexity of the quotient selection process and it can be implemented as a finite table. The partial remainder computation can be overlapped with the quotient digit selection computation.

The quotient digit selection table implementing the P-D plot for radix 4 is reproduced above from [47]. Rows are indexed by the shifted truncated partial remainder $g7g6g5g4.g3g2g1$ (in 2's complement); columns are indexed by the truncated divisor $f1.f2f3f4$; table entries are the quotient digits. The table is compressed by considering only row indices up to 5 bits since only a few entries in the table depend upon the two least significant bits $g2g1$ of the shifted partial remainder. For those cases, the table entries are symbolic values A, B, C, D, E defined in term of $g2g1$ as:

$A = -(2 - g2 * g1)$, $B = -(2 - g2)$, $C = 1 + g2$, $D = -1 + g2$, $E = g2$.

Every entry in the table is thus for four remainder estimates. The - entries in the table are the *dontcare* entries.

In the above recurrence relations, q_{j+1} is replaced by `qtable(up, ud)`, where `qtable` is the quotient selection table, `up, ud` are, respectively, the truncated partial remainder and divisor.

7.2 SRT divider circuit

A radix 4 SRT divider circuit based on the above quotient digit selection table is described in Fig. 3. The reg-

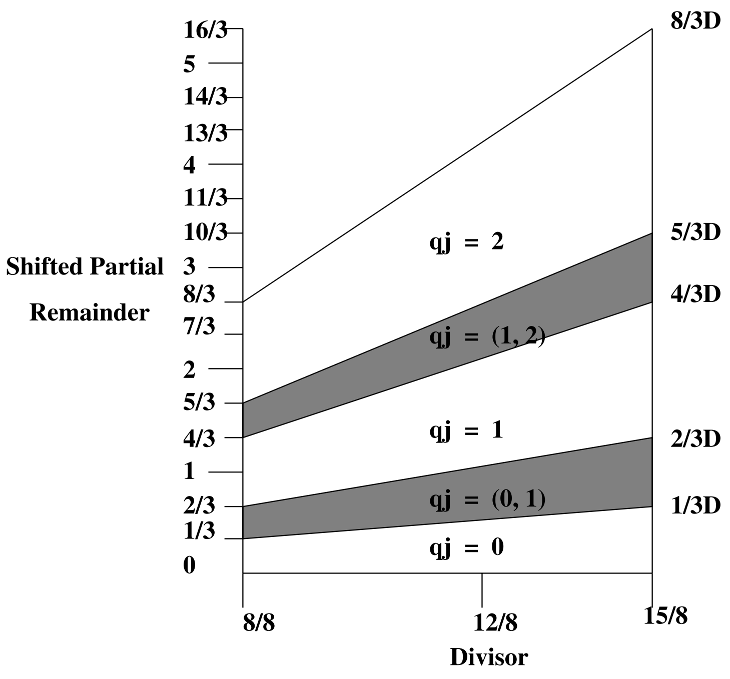

Fig. 2. P-D plot for radix 4

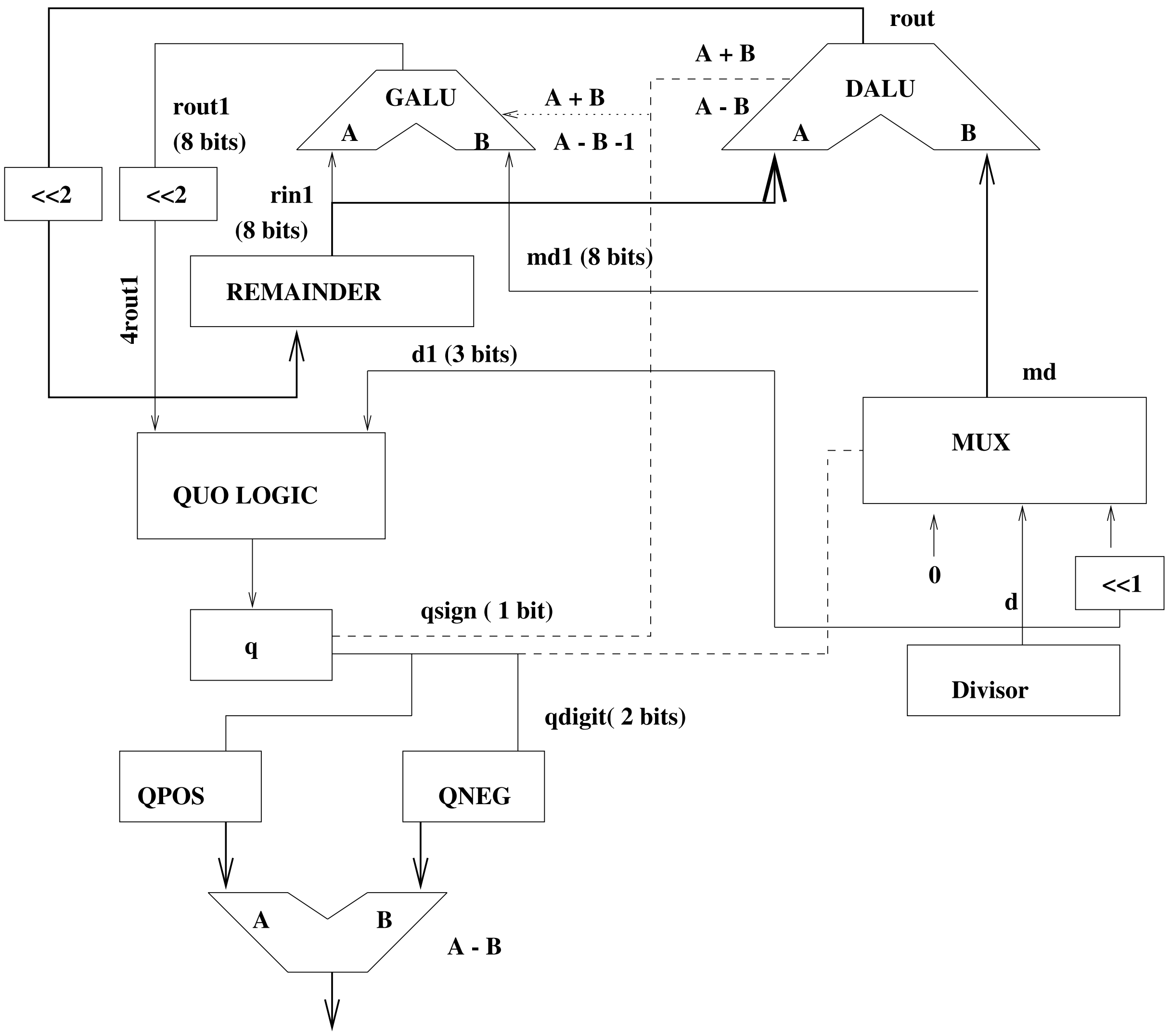

Fig. 3. SRT division circuit using radix 4

isters *divisor*, *remainder* in the circuit hold the value of the divisor and the successive partial remainders, respectively. The register q holds the selected quotient digit along with its sign; the registers *QPOS* and *QNEG* hold the positive and negative quotient digits of the quotient. A multiplexor *MUX* is used to generate the correct multiple of the divisor based on the selected quotient digit by appropriately shifting the divisor. The hardware component *DALU* is a full width ALU that computes the partial remainder at each iteration. The component *GALU* (the guess ALU [47]) is an 8-bit ALU that computes the approximate 8-bit partial remainder to be used for quotient selection. The components $<<2$ perform left shift by 4. The hardware component *QUO LOGIC* stands for the quotient selection table, and it is typically implemented using an array of preprogrammed read-only-memory. *GALU* computes an 8-bit estimate of the next partial remainder which is left shifted by 4 and then used with the truncated divisor ($d1$) to index into *QUO LOGIC* to select the quotient digit for the next iteration.

Note that *GALU* and the quotient digit selection are done in parallel with the full width *DALU* so that the correct quotient digit value is already available in the register q at the beginning of each iteration.

7.3 Modeling table as a special data type

We define a table as a special data type 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* as given in Table 1.TS^r In the absence of that, we propose a simple mechanism using finite enumerated types for indices.

A finite enumerated data type is a finite set of distinct values, typically denoted by a finite set of distinct free constructor symbols, i.e., every two distinct constructors are not equal. Such a data type `en` can be specified in *RRL* by listing CE^s its constructors as nullary constants of types `en`, and declaring them to be *free*. Since finite

TS^r Table 1 or 2?
CE^s Should 'nullary' be 'null' in line below?

Editor's or typesetter's annotations (will be removed before the final TEX run)

MS ID: STTTN032
31 May 2000 14:59 CET

Table 2. Quotient digit selection table

parrem	Divisor							
	f1.f2f3f4							
g7g6g5g4.g3g2g1	1.000	1.001	1.010	1.011	1.100	1.101	1.110	1.111
1010.0	–	–	–	–	–	–	–	–
1010.1	–	–	–	–	–	–	-2	-2
1011.0	–	–	–	–	–	-2	-2	-2
1011.1	–	–	–	-2	-2	-2	-2	-2
1100.0	–	–	-2	-2	-2	-2	-2	-2
1100.1	-2	-2	-2	-2	-2	-2	-2	-2
1101.0	-2	-2	-2	-2	-2	-2	B	-1
1101.1	-2	-2	-2	B	-1	-1	-1	-1
1110.0	A	B	-1	-1	-1	-1	-1	-1
1110.1	-1	-1	-1	-1	-1	-1	-1	-1
1111.0	-1	-1	D	D	0	0	0	0
1111.1	0	0	0	0	0	0	0	0
0000.0	0	0	0	0	0	0	0	0
0000.1	1	1	1	1	E	0	0	0
0001.0	1	1	1	1	1	1	1	1
0001.1	2	C	1	1	1	1	1	1
0010.0	2	2	2	2	C	1	1	1
0010.1	2	2	2	2	2	2	2	1
0011.0	–	2	2	2	2	2	2	2
0011.1	–	–	2	2	2	2	2	2
0100.0	–	–	–	–	2	2	2	2
0100.1	–	–	–	–	–	2	2	2
0101.0	–	–	–	–	–	–	–	2
0101.1	–	–	–	–	–	–	–	–

subranges of natural numbers are often used for indices, a finite enumerated data type can also be specified as a subrange: `enum en: [lo ... hi: nat]`, where `lo` and `hi` are natural numbers with `lo <= hi`. Subranges over integers can also be used as shown below for the quotient digit selection table for SRT division.

If the constructors of an enumerated type are given using numbers, then an implicit conversion from the values of the enumerated type to numbers is done so that the usual operations on numbers supported by *RRL* as a part of the quantifier-free theory of Presburger arithmetic can be used. As will be evident below, for SRT division, such an implicit conversion is quite useful.

A parameterized (generic) function `lookup` is associated with the data type `table` to access the entries of a specific table given the index values. We slightly abuse the notation and write `lookup(t, i1, j1)` to mean the entry associated with the index values `i1, j1` in the table `t`. For convenience, we introduce the syntactic sugar for `lookup(t, i1, j1)` and write it as `t(i1, j1)`. A table can be specified by enumerating its entries as: `t(i1, j1) := v1, t(i2, j2) := v2, ...` . Entries not explicitly listed are assumed to be not specified.

Tables with don't-care entries

Many lookup tables, in practice, have don't-care entries, i.e., for certain index values, it does not really matter what the table entry is. This may be so either because the table is not meant to be used for such index values, or the properties of interest involving the table do not depend upon the entry value for such index values. A table with don't-care entries is supported in a similar way to a table without don't-care entries, with the difference that a special constant value `dontcare` of type `Dtcare` is used as an entry value.

Specifying quotient digit selection table in RRL

The quotient digit selection table is specified in *RRL* by `qtable` as an instance of the parameterized table type discussed above. The table indices are given by the integer subranges `column` and `row`. The entry type of `qtable` is the union of integers and `Dtcare`, but only the subrange `[m(2)...2]` is used. The unary function `m` is the minus operation on integers[12]. As the table is too big to be included here, we reproduce below a portion of the specification – a part of the eighth row.

The eighth row corresponds to four shifted truncated remainder estimates: $\{-17/8, -9/4, -19/8, -5/2\}$ depending upon the values of $g2g1$; they are scaled up by

[12] Instead of using fractional numbers for indices, it is more convenient and faster for *RRL* to use their scaled integer versions as indices to the table. So all row and column indices are scaled up by 8. Scaling up effectively leads to using number representations of bit vectors of the shifted truncated partial remainder estimate and the truncated divisor estimate by dropping the decimal point.

multiplying by 8, to $\{-17, -18, -19, -20\}$ (2's complement is used in Table 1 TS t for row indices). Below, the table entries for row index 20 are given.

```
[8...15 : column]           [m(48)...47: row]

table [qtable : row, column -> integer U Dtcare]

qtable(m(20),8)  := m(2)  qtable(m(20),9)  := m(2)
qtable(m(20),10) := m(2)  qtable(m(20),11) := m(2)
qtable(m(20),12) := m(1)  qtable(m(20),13) := m(1)
qtable(m(20),14) := m(1)  qtable(m(20),15) := m(1)
```

The table entry for the eighth row and the column index 1.011 (11) is B, where $B = -(2 - g2)$. For all other column indices, the entries do not depend upon $g2g1$. So for all column indices other than 11, the table value is the same irrespective of whether the row index is $-20, -19, -18$ or -17.

For the column index 11, the table entry is -2 if the row index is -20 or -19, since in that case $g2$ is 0; if the row index is -18 or -17, then the table entry is -1.

```
qtable(m(19),11) := m(2)
qtable(m(18),11) := m(1)
qtable(m(17),11) := m(1)
```

Other rows are similarly specified, with each row defining 32 table entries.

7.4 Verifying SRT division by exploiting sparsity in tables

The main invariant of SRT division is specified in *RRL* using `qtable` as:

```
(C3): m(2) * divsr <=
12 * parrem - 3 * qtable(up, ud) * divsr
<= 2 * divsr if
{m(2) * divsr  <= 3 * parrem  <= 2 * divsr,
ud  <= 8 * divsr < ud  + 1,
up <=  32 * parrem < up + 1}.
```

The above formula states that if the partial remainder `parrem` in the previous iteration is within bounds of the divisor `divsr` (the absolute value of the partial remainder is within two-thirds of the divisor) and if the table indices `up, ud` correctly approximate the divisor and the partial remainder within certain bounds, then the partial remainder computed in the next iteration `4 * parrem - qtable(up, ud) * divsr` would continue being appropriately bounded by the divisor.

In [24, 31], we reported two different methods for proving this invariant. In [31], $(C3)$ was automatically proved in *RRL* by modeling quotient selection table as a function over integers, and by performing case analysis on the table indices *up* and *ud*. This leads to 1536 cases, 768 cases each for proving the upper bound and lower bound, respectively. Don't-care entries are modeled by out-of-bound integers, and the intermediate cases generated are extremely cumbersome. In [24], the proof was done using an intensional formulation of the quotient table by abstracting table entries in terms of boundary value predicates proposed in [13]. This approach requires user guidance in terms of additional lemmas besides the manual abstraction of the table. Establishing the correctness of this manual abstraction is nontrivial.

The proof can, however, be vastly simplified by noting that the number of distinct entries in the table is only six, `Dtcare, -2, -1, 0, 1, 2`. Hence only 12 top-level cases need be generated if case analysis is performed on entries instead of indices.

Reasoning About sparse tables

RRL automatically invokes case analysis on entries for sparse tables such as the quotient digit selection table. The notion of sparsity used by *RRL* is reviewed below. More details can be found in [34].

A table `t`, `table [ t:e1, e2` $\rightarrow$ `er]`, is sparse iff $|entries(t)| \leq minimum(|e1|, |e2|)$, where $entries(t)$ is the set of all table entries including the don't-care value, if used.

The rationale behind this definition is that performing case analysis on entry values for a sparse table does not result in more cases than would arise if the case analysis is done on any of the index variables. 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.

Sometimes it is also possible to use properties of index types as well for testing constraints on indices deduced from an instantiation of a given conjecture based on a particular table entry value. For a particular table entry, say v, a proof of the simplified conjecture can be attempted for the index values for which the table has v as the entry. 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 v 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.

Consider a universally quantified conjecture $\phi(x_1, \cdots, x_n, y_1, \cdots, y_m)$ where $x_1, \cdots, x_n$ are the index variables and $y_1, \cdots, y_m$ are the nonindex variables. Without any loss of generality and for simplicity, we assume a single table term $t(x_1, \cdots, x_n)$ occurring in ϕ. Consider a particular entry value, say v of $t(x_1, \cdots, x_n)$; let I be the finite set of index tuples (values of $x_1, \cdots, x_n$) for which the table t has entry value v.

When ϕ is attempted for the case $t(x_1, \cdots, x_n) = v$, a formula ϕ_1 with $t(x_1, \cdots, x_n)$ replaced by v is obtained

CE u End-stop added after 'mynot' and ∀ eliminated. OK?

Editor's or typesetter's annotations (will be removed before the final TEX run)

MS ID: STTTN032
31 May 2000 14:59 CET

from ϕ. Typically ϕ_1 is simpler than ϕ. If it can be proved, we are done. In general, ϕ_1 need not be valid, in which case, the goal is to find an equivalent quantifier-free formula $\psi(x_1, \cdots, x_n)$ without any nonindex variables such that for each index tuple satisfying ψ, ϕ_1 is true for every value of the nonindex variables. CE^{u}

Let $\theta(x_1, \cdots, x_n)$ be $\neg$. The formula θ characterizes the index tuples for which there is at least a tuple of values for nonindex variables that falsifies ϕ_1. Let $\psi_1(x_1, \cdots, x_n)$ be a quantifier-free formula equivalent to $\neg\ \forall\ y_1, \cdots y_m\ \ \phi_1(x_1, \cdots,\ x_n, y_1, \cdots,\ y_m)$. Then $\psi = \neg\psi_1$ characterizes the set of index tuples such that for all values of nonindex variables, ϕ_1 is true. The formula $\psi(x_1, \cdots, x_n)$ is the constraint on index variables for ϕ to be valid if $t(x_1, \cdots, x_n) = v$. If for every tuple in I, ψ is true, then ϕ is valid for the case when $t(x_1, \cdots, x_n) = v$.

For a don't-care entry value, it must be ensured that the conjecture is valid independent of the table entry. This particular case is handled separately without replacing the table term by the don't-care value.

We illustrate this approach in a proof of the above invariant of the SRT division algorithm.

SRT division correctness

The correctness proof of $(C3)$ is done by case analysis on table entry values rather on the indices. For the lower bound, this leads to $five$ top-level cases – five corresponding to the entry values in the subrange `[m(2)...2]`, and one case is generated for the don't-care entry value. Six cases are generated for the upper bound as well.

For *qtable(up, ud) = 0*, $(C3)$ simplifies to

```
(C3.0): (-divsr) <= (6 *  parrem) <= divsr if
{(-2 * divsr) <= (3 * parrem) <= (2 * divsr),
(ud <=  (8 * divsr) < (ud + 1)),
(up <=  (32 * parrem) < (up + 1))}.
```

Consider the subcase of this simplified formula to show `- divsr <= 6 * parrem`. The negated formula is:

```
(Exists. parrem, divsr)
[(6 * parrem < -divsr) and
(-2 * divsr <= 3 * parrem) and
(3 * parrem <= 2 * divsr) and
(ud <=  (8 * divsr) and (8 * divsr < ud + 1) and
(up <=  32 * parrem) and
(32 * parrem < up + 1)].
```

The non-index variable `parrem` can be eliminated by cross-multiplying the coefficients of `parrem` [26]. The resulting formula is `6 * up <= -32 * divsr`. Using `ud <= 8 * divsr` eliminates the remaining nonindex variable `divsr` to give `48 * up < -32 * ud`. The constraint on indices is generated by negating and simplifying this formula:

```
(0, Lowerbound): up >= -2/3 ud.  (I)
```

For the second subcase corresponding to the upper bound `6 * parrem <= divsr`, the nonindex variables `divsr, parrem` are eliminated from the negated formula leading to the constraint

```
(0, Upperbound): up + 1 <= 2/3 ud.  (II)
```

The constraints for other table entries can be similarly derived.

The proof of the invariant $(C3)$ is reduced to showing that nine constraints similar to I and II on indices `up, ud` are satisfied for different quotient digit values. One method to check them is to explicitly plug in various values of `up, ud` which give rise to each of quotient digits.

Since these constraints are simple inequalities, and indices are subranges over numbers, this information can be used to simplify checking these constraints. For instance, consider constraint `(I): up >= - 2/3 ud`. For `qtable(up, ud) = 0`, `ud` ranges over [8...15], meaning `-2 * ud` is in the range `[m(30), m(16)]`. The constraint is satisfied for all values of `up` greater than or equal to `m(5)`. The remaining cases to be considered are:

```
[(m(6), 10),..., (m(6), 15),
 (m(7), 12),..., (m(7), 15),...,
 (m(8), 12),..., (m(8), 15).]
```

When `up = m(6)`, `9 <= ud`. When `up = m(7)`, `11 <= ud`. So all cases are considered.

A similar analysis works for `(II)` and other quotient digits, including the don't-care entry.

To summarize, for circuits using sparse tables, verification of their properties can involve extensive case analysis, which can be simplified and structured by performing case analysis based on the values appearing in the table instead of its indices. Projection and elimination techniques can be exploited to generate simpler constraints that are easier to check. Index type structure can be used to the advantage in simplification and analysis.

8 Conclusion: future enhancements and 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* that 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. Such experience gives hope that with additional enhancements and extensions to theorem proving tools, it is possible to automatically verify a larger class of circuits. In the rest of this section, we propose enhancements needed for theorem provers and challenges faced by theorem proving research to make theorem provers more effective and useful for this application of hardware reliability.

We are quite aware of the limitation of the approach advocated in this paper. The main thing we have demonstrated is that a theorem proving tool exists with capabilities and features using which it is possible to automatically verify (without any human guidance or intervention) properties of simple arithmetic circuits such as adders, multipliers and division 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.

First and foremost the input language of *RRL* is a functional programming language using recursion, and not a hardware description language. There is clearly 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*. This does not appear to be an easy task, as a number of researchers have attempted to do so with limited success due to the problematic semantics of some of the features in these languages.

The second enhancement needed is that 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 CE^{V} circuit. If the portion of the circuit which is giving rise to the conflict between the circuit description and the property can be localized, then this can be helpful in understanding the problem and possibly lead to a fix. Some believe that this can perhaps be the most useful role for theorem proving tools.

Third, 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 hardware design environment. The use of a theorem prover should not be perceived as a burden or something to be avoided unless it becomes unavoidable.

The above discussion about extensions and enhancements to theorem proving tools has been motivated by the objective of making them accessible to the practitioners of the application domain. Now, we discuss extensions to a theorem prover such as *RRL* to enhance its reasoning power to broaden its scope and make it widely applicable to a larger family of circuit designs. First, we address enhancements to address some peculiarities of *RRL*.

As of now, *RRL* does not use an efficient representation of numbers including naturals, integers and rationals. 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 procedure 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 many control-theoretic properties, it is more effective to use a state machine model. It would be useful to 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. CE^{W}

References

1. Angelo, C.M., Claesen, L., De Man, H.: A Methodology for proving correctness of parameterized hardware modules in HOL. In: Borrione, D., Waxman, R. (eds.): CHDL '91, Amsterdam: Elsevier Science (North-Holland), 1991
2. Basin, D., Walsh, T.: Difference Matching. In: Kapur, D. (ed.): Proc. CADE 11. LNAI 607. Berlin, Heidelberg, New York: Springer-Verlag, 1992
3. Boyer, R.S., Moore, J.S.: A Computational Logic. ACM Monogr Comput Sci, 1979
4. Boyer, S.B., Moore, J.S.: Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. Mach Intell 11: 83–157, 1988
5. Boyer, R.S., Moore, J.: A Computational Logic Handbook. New York: Academic Press, 1988
6. Boyer, R.S., Moore, J., Kaufmann, M.: Functional Instantiation in Nqthm. CLI Tech Rep
7. Brock, B.C., Hunt, W.A., Kaufmann, M.: The FM9001 Microprocessor Proof. CLI Tech Rep 86, Dec 1994
8. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput C-35(8), 1986
9. Bryant, R.E., Chen, Y.-A.: Verification of arithmetic functions with binary moment diagrams. Tech Rep CMU-CS-94-160, June 1994
10. Bryant, R.E.: Bit-level analysis of an SRT divider circuit. Tech Rep CMU-CS-95-140, Carnegie Mellon University, April 1995
11. Burch, J.R., Clarke, E.M., Mcmillan, K.L., Dill, D.L.: Sequential circuit verification using symbolic model checking. 27th ACM/IEEE Design Autom Conf, 1990
12. Camilleri, A.J., Gordon, M.J.C., Melham, T.F.: Hardware verification using higher-order logic. In: Borrione, D. (ed.): HDL Descriptions to Guaranteed Correct Circuit Designs. Amsterdam: North Holland, pp. 43–67, 1987
13. Clarke, E.M., German, S.M., Zhao, X.: Verifying the SRT division algorithm using theorem proving techniques. In: Alur, R., Henzinger, T. (eds.): Proc. Comput Aided Verification, 8th Int Conf CAV '96, New Brunswick, July/August 1996. LNCS 1102. Berlin, Heidelberg, New York: Springer-Verlag, pp. 111–122, 1996
14. Cyrluk, D., Rajan, S., Shankar, N., Srivas, M.K.: Effective theorem proving for hardware verification. In: Kumar, Kropf

CE^{V} The following sentence changed and re-ordered. OK?
CE^{W} Please supply author's/s' (ed/s) initials whenever they are missing in the refs, e.g., [14]

Editor's or typesetter's annotations (will be removed before the final TEX run)

(eds.): Proc. 2nd Conf Theorem Provers in Circuit Design, Sept 1994
15. Dadda, L.: Some schemes for parallel multipliers. In: Swartzlander, E.E., Jr. (ed.): Computer Arithmetic I, IEEE Comput Soc Press, 1990
16. Dershowitz, N.: Termination of rewriting. J Symb Comput 3: 69–116
17. Ercegovac, M.D., Lang, T.: Division and Square Root: Digit Recurrence Algorithms and Implementations. Boston, MA: Kluwer, 1994
18. Ercegovac, M.D., Lang, T.: Radix-4 square root without initial PLA. IEEE Trans Comput 39(8), 1990 CE^{x}
19. German, S.: Towards Automatic Verification of Arithmetic Hardware. Lecture Notes, 1995
20. Hesketh, J.T.: Using middle out reasoning to guide inductive theorem proving. Ph.D thesis. University of Edinburgh, UK, 1991
21. Hunt, W.A., Brock, B.C.: The verification of a bit-slice ALU. Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects. LNCS 408. Berlin, Heidelberg, New York: Springer-Verlag, 1989
22. Ireland, A., Bundy, A.: Productive Use of Failure in Inductive Proof. Edinburgh DAI Research Report No: 716
23. Ireland, A.: The use of planning critics in mechanizing inductive proofs. In: Voronkov, A. (ed.): Proc LPAR ´92. LNAI 624. Berlin, Heidelberg, New York: Springer-Verlag, 1992
24. Kapur, D.: Rewriting, decision procedures and lemma speculation for automated hardware verification. Proc 10th Int Conf Theorem Proving in Higher Order Logics. LNCS 1275. Berlin, Heidelberg, New York: Springer-Verlag, 1997
25. Kapur, D.: Shostak's congruence closure as completion. Proc Intl Conf on Rewriting Techniques and Applications, (RTA-97), Barcelona, Spain, June 1997
26. Kapur, D., Nie, X.: Reasoning about numbers in Tecton. Proc 8th Int Symp Methodol for Intelligent Systems, (ISMIS '94), Charlotte, NC, October 1994, pp. 57–70
27. Kapur, D., Subramaniam, M.: New uses of linear arithmetic in automated theorem proving for induction. J Autom Reasoning 16(1-2): 39–78, 1996
28. Kapur, D., Subramaniam, M.: Mechanically verifying a family of multiplier circuits. In: Alur, R., Henzinger, T. (eds.): Proc Comput Aided Verification (CAV '96), New Jersey. LNCS 1102. Berlin Heidelberg New York: Springer-Verlag, pp. 135–146, 1996
29. Kapur, D., Subramaniam, M.: Mechanical verification of adder circuits using powerlists. Dept Comput Sci Tech Rep, SUNY Albany, November 199. Accepted for publication in J Formal Methods in System Design
30. Kapur, D., Subramaniam, M.: Lemma discovery in automating induction. In: McRobbie, Slaney (eds.) Proc Int Conf on Autom Deduction, CADE-13. LNAI 1104. New Jersey, July 1996
31. Kapur, D., Subramaniam, M.: Mechanizing reasoning about arithmetic circuits: SRT division. In: Sivakumar, Ramesh (eds.): Proc 17th FSTTCS. CE^{y} LNCS. Berlin, Heidelberg, New York: Springer-Verlag, 1997
32. Kapur, D., Subramaniam, M.: Intermediate lemma generation from circuit descriptions. (in preparation) State University of New York, Albany, NY, May 1997
33. Kapur, D., Zhang, H.: An overview of Rewrite Rule Laboratory (RRL). J Comput Math Appl 29(2): 91–114, 1995 CE^{z}
34. Kapur, D., Subramaniam, M.: Mechanizing reasoning about large finite tables in a rewrite-based theorem prover. In: Proc. of ASEAN-8. LNCS, 1998
35. Moore, J., Lynch, T., Kaufmann, M.: A Mechanically Checked Proof of the Correctness of the AMD5K86 Floating Point Division Algorithm. CE$^{\text{TOOMUCH}}$ CL Tech Rep, March 1996
36. Miner, P.S., Leathrum, J.F.: Jr. Verification of IEEE compliant subtractive division algorithm. Proc FMCAD '96, Palo Alto, CA. LNCS 1166. Berlin, Heidelberg, New York: Springer-Verlag, 1996
37. Montoye, R.K., Hokenek, E., Runyon, S.L.: Design of the IBM RISC System/6000 floating-point execution unit. IBM J 34(1), 1990
38. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans Program Lang Syst 1(2): 245–257, 1979
39. Omondi, A.R.: Computer Arithmetic Systems: Algorithms, Architecture and Implementations. Englewood Cliffs, NJ: Prentice Hall, 1994
40. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. Cambridge, MA: MIT Press, 1991
41. Robertson, J.E.: A new class of digital division methods. IRE Trans Electron Comput, pp. 218–222, 1958
42. Ruess, H.: Hierarchical verification of two-dimensional high-speed multiplication in PVS: a case study. Formal Methods in CAD. LNCS 1166. Berlin, Heidelberg, New York: Springer-Verlag, 1996
43. Ruess, H., Shankar, N., Srivas, M.K.: Modular verification of SRT division. In: Alur, R., Henzinger, T. (eds.): Proc Comput Aided Verif, 8th Int Conf - CAV '96, New Brunswick, July/August 1996. LNCS 1102. Berlin, Heidelberg, New York: Springer-Verlag, pp. 123–134, 1996
44. Sarma, D.D., Matula, D.: Measuring the accuracy of ROM reciprocal tables. IEEE Int Symp on Comput Arith, IEEE Computer Society, 1993
45. Shostak, R.E.: Deciding combination of theories. J ACM 31(1): 1–12, 1984
46. Subramaniam, M.: Failure Analyses of Inductive Theorem Provers. Doctoral Dissertation, State University of New York, Albany 1996
47. Taylor, G.S.: Compatible hardware for division and square root. Proc 5th IEEE Symp Comput Archit, May 1981
48. Tocher, K.D.: Techniques of multiplication and division for automatic binary computers. Q J Mech Appl Math 11(3): 1958
49. Verkest, D., Claesen, L., De Man, H.: On the use of the Boyer-Moore theorem prover for correctness proofs of parameterized hardware modules. In: Claesen, L. (ed.): Formal VLSI Specification and Synthesis: VLSI Design Methods I, Elsevier Science (North-Holland), 1990
50. Wallace, C.S.: A suggestion for a fast multiplier. In: IEEE Trans Electron Comput EC-13: 14–17, 1964
51. Walsh, T.: A divergence critic. In: Bundy, A. (ed.): Proc CADE 12. LNAI 814. Berlin, Heidelberg, New York: Springer-Verlag, 1994
52. Zhang, H.: Implementing contextual rewriting. In: Remy, Rusinowitch (eds.): Proc 3rd Int Workshop on Cond Term Rewriting Syst. LNCS 656. Berlin, Heidelberg, New York: Springer-Verlag, pp. 363–377, 1992
53. Zhang, H., Kapur, D., Krishnamoorthy, M.S.: A mechanizable induction principle for equational specifications. In: Lusk, Overbeek (eds.): Proc 9th Int Conf Automat Deduction (CADE), Chicago. LNCS 310. Berlin, Heidelberg, New York: Springer-Verlag, pp. 250–265, 1988
54. Zurawski, J.H., Gosling, J.B.: Design of a high speed square root multiply and divide unit. IEEE Trans Comput C-36, 1987

CE^{x} Please supply missing information in ref below.(Which lecture notes, number...)
CE^{y} Please supply missing LNCS number in ref below.
CE^{z} Please supply missing LNCS number in ref below.
CE$^{\text{TOOMUCH}}$ Should 'CL' in ref below be 'CLI'?