Verified Implementation of an Efficient Term-Rewriting Algorithm for Multiplier Verification on ACL2

Mertcan Temel
University of Texas at Austin
Austin, TX, USA
mert@utexas.edu

Automatic and efficient verification of multiplier designs, especially through a provably correct method, is a difficult problem. We show how to utilize a theorem prover, ACL2, to implement an efficient rewriting algorithm for multiplier design verification. Through a basic understanding of the features and data structures of ACL2, we created a verified program that can automatically verify various multiplier designs much faster than the other state-of-the-art tools. Additionally, users of our system have the flexibility to change the specification for the target design to verify variations of multipliers. We discuss the challenges we tackled during the development of this program as well as key implementation details for efficiency and verifiability. Those who plan to implement an efficient program on a theorem prover or those who wish to implement our multiplier verification methodology on a different system may benefit from the discussions in this paper.

1 Introduction

Integer multipliers are ubiquitous circuit components that are fundamental in general-purpose, cryptographic, image, and signal processors. They can be used for various arithmetic operations such as floating-point multiplication, integer multiplication, dot-product, division, and square root. Consequently, the correctness of multipliers is an important factor for the reliability of such systems.

Formal verification of integer multiplier designs is still an ongoing problem. Some earlier methods, such as BDDs and BMDs, can efficiently verify array multipliers [3, 4], which are regularly structured designs. However, these methods, as well as SAT Solvers [22], do not scale for automatic verification of more common multiplier architectures, i.e., Wallace-tree like multipliers and Booth Encoding. Their irregular and more advanced structure complicates the process for automatic tools; therefore, verification of industrial designs is carried out mostly manually [10, 11, 12, 21, 23]. Recent studies have focused on computer algebra based methods and they have shown significant improvements [5, 13, 17, 18, 31]. Some of these tools [13, 18] can verify large isolated integer multipliers in a shorter amount of time than previously reported tools (e.g., 256x256-bit multipliers can be verified in a few hours, and in some cases, minutes). However, very little effort was given to verifying the tools themselves. Only one tool [13] can generate certificates to check the verification result by external proof checkers. Additionally, specification for designs is hard-coded in these tools, and it may not be possible to verify modified or embedded multipliers with these tools [29].

We have created a new term-rewriting algorithm that can efficiently and automatically verify large arithmetic circuit designs with embedded multipliers [29, 30]. We have shown that this algorithm can verify designs with millions of gates in just a few minutes. For example, an isolated 1024x1024 Booth Encoded Wallace-tree multiplier can be verified in 5 minutes. Other multiplier-centric designs, such as dot-product, and various architectures, such as array multipliers, can also be verified quickly. Our

Rob Sumners, Cuong Chau (Eds.): 17th ACL2 Workshop (ACL2 2022)
EPTCS 359, 2022, pp. 116–133, doi:10.4204/EPTCS.359.11

algorithm scales much better than the best available tool [13]: it can verify 1024x1024-bit multipliers around 40 times faster on average.

Not only did we create a more efficient algorithm, but we also implemented and verified it using the interactive theorem prover, ACL2. This ensures the user of the result when a design is claimed to be correct. Moreover, the interactive system provides more flexibility to the user; e.g., the ability to decide on the specification for a design. In our previous work, we have shown that we can verify custom designs with control logic, multiply-accumulate, and dot-product with the same level of automation [29].

Managing a term-rewriting algorithm of this scale is not a trivial task. Implementing and verifying programs using an interactive theorem prover brings about unique challenges that one might not encounter while developing unverified programs in other high-level languages, such as Java and C++. Data structures in ACL2 and its rewriter have certain constraints, and the proof obligation bring about further challenges. While programming, we had to consider that terms are always represented as trees and are stored in memory as unmodifiable linked-lists. Additionally, terms might change so drastically during rewriting that it might become too difficult to apply some rewrite rules. In our previous work, we have described only the term-rewriting algorithm itself [29, 30], and some features of the supporting rewriter technology [27]. The goal of this paper is to elaborate on the multiplier-specific implementation details and notable challenges.

This paper is structured as follows. Sec. 2 provides the necessary background information to follow the paper, including the term-rewriting methods in ACL2 and our algorithm to verify multiplier designs. Sec. 3 describes the implementation framework. Sec. 4 details the main challenges we encountered during implementation and our solutions. [1]

Some portions of this work have previously appeared in the author's PhD thesis [28].

2 Preliminaries

In this section, we describe the basic and relevant methods to apply a term-rewriting algorithm in ACL2 as well as our algorithm to rewrite and simplify multiplier designs.

2.1 Methods of Rewrite

ACL2 is a programming language and an automated theorem prover for first-order logic [14]. Users can model systems and reason about them using either its built-in features, user-contributed libraries, or external tools such as a SAT solver [9, 15, 20, 24]. In our multiplier verification project, we are only interested in basic rewriting capabilities. A given conjecture to the system can be rewritten by rewrite rules, meta rules [8], or clause processors [16].

Rewrite Rules Previously proven lemmas can be stored as rewrite rules and later be used to rewrite terms to help prove conjectures. If a lemma is of the form $hyp \implies lhs = rhs$ and it is enabled in the ACL2 system as a rewrite rule, then the *lhs* will be compared to terms being rewritten. If the *lhs* pattern can be unified with a (sub)term and *hyp* is relieved, then the term will be replaced with the *rhs* pattern with appropriate term bindings.

Table 1 shows an example of how a rewrite rule alters a term. The `defthm` utility is used to submit conjectures to ACL2 to attempt proofs. We can prove the associativity of summation (see the first

[1] Various demo files showing how to input a Verilog design and verify it with our system, and the source code of the program discussed in this paper can be found in the ACL2 community books [1] under the books/projects/rp-rewriter/lib/mult3 directory.

Table 1: An example term being modified by an example rewrite rule

Rewrite Rule	Target Term	After RW1	After RW2
`(defthm sum-assoc (equal (+ (+ a b) c) (+ a (+ b c))))`	`(+ (+ x1 x2) (+ (+ x3 x4) x5))`	`(+ (+ x1 x2) (+ x3 (+ x4 x5)))`	`(+ x1 (+ x2 (+ x3 (+ x4 x5))))`

column) using the existing libraries and built-in axioms in ACL2. When the `defthm` event succeeds, it automatically saves the lemma as a rewrite rule with the given name. When this rewrite rule is in the system, we can apply it to terms whenever the pattern from the left-hand side finds a match. Assume that this is the only enabled rule and we would like to prove another conjecture which contains a term given in the "Target Term" column. The rewriter performs inside-out rewriting. Therefore, it will first dive into the innermost term to search for matching patterns. The first match occurs for the following bindings: `a` to `x3`, `b` to `x4`, and `c` to `x5`. With these term bindings, the term is replaced using the right-hand side of the rewrite rule and we obtain the term in the third column. The rule can find another match on this new term. After rewriting this term in a similar fashion, we obtain the term in the last column.

Meta Rules Instead of applying a fixed pattern from a rewrite rule, users may define custom functions to match patterns and modify terms. We refer to these functions as *meta functions*, and their associated rules as *meta rules* [8]. Meta rules have associated trigger functions. Whenever the rewriter encounters an instance of one of those trigger functions, the associated meta function is called to rewrite the current term. Meta rules can often be used to achieve better efficiency in terms of both applicability and resources (i.e., CPU-time and memory). In order to define meta rules, users should verify the correctness of their meta functions.

There might be cases where a system with only rewrite rules is not sufficient to apply a term-rewriting algorithm. Consider the example rewrite rule given in Table 2. This rule shows that for the function `s`, we can remove duplicate elements in the summation argument. For this rewrite rule to apply, candidate terms have to match the exact pattern from this rewrite rule, requiring duplicate elements to appear at the beginning of the summation for all to be removed. As seen in the "Target Term" column, this might not always be the case; repeated terms `x2` and `x4` would not be removed by applying this rewrite rule. We could possibly define other rewrite rules to update these terms; however, there could be an indeterminate number of elements in summations and it is not feasible to define rewrite rules that would match all possible patterns. We can overcome this issue by defining a function that can go through the whole summation term, remove duplicates and return the desired term.

A simple program consisting of two meta functions that can remove such duplicate elements is given in Example 1. The second function `s-of-repeated-meta-fn` is to be called every time a new `s` instance is created. Then, the argument of such an instance is passed to the first function, `rm-dp-for-s`. This function recursively examines all the elements in a term representing summation. If there is a duplicate, the function removes it; and for all other terms, it reconstructs the summation. Finally, the updated summation term is returned to the second function and a functionally equivalent `s` instance is created. This program assumes that terms representing summations are lexicographically sorted (using commutativity and associativity of summation), ensuring that repeated elements appear next to each other.

Table 2: An example that shows a rewrite rule might not be sufficient to apply a term-rewriting strategy

Rewrite Rule	Target Term

`(defthm s-of-repeated`
`  (equal (s (i+ a (i+ a b)))`
`  (s b)))`
 	
where
`(s x) = (mod (ifix x) 2)`
`(i+ x y) = (+ (ifix x) (ifix y))`
`(ifix x) = (if (integerp x) x 0)`
	`(s (i+ x1`
	`  (i+ x2`
	`  (i+ x2`
	`  (i+ x3`
	`  (i+ x4`
	`  (i+ x4`
	`  x5)))))))`

Example 1. *Meta functions that can remove duplicates from the summation argument of an* `s` *instance.*

```
(define rm-dp-for-s (term)
  (case-match term
    (('i+ x ('i+ y z))
     (if (equal x y)
         (rm-dp-for-s z)
       `(i+ ,x ,(rm-dp-for-s `(i+ ,y ,z)))))
    (('i+ x y)
     (if (equal x y)
         ''0
       term))
    (& term)))

(define s-of-repeated-meta-fn (term)
  (case-match term
    (('s sum)
     `(s ,(rm-dp-for-s sum)))
    (& term)))
```

Verification of meta functions is often more difficult than rewrite rules. Not only do we need to prove properties about the interpreted functions, such as `s` and `i+`, we also need to show that the meta functions return equivalent terms. Example 2 shows the list of events to verify the functions in Example 1. We omit the lemmas about `s` and `i+` (e.g., `s-of-repeated` in Table 2) for brevity. To verify meta functions, ACL2 requires users to create an *evaluator* that can recognize functions in rewritten terms. Then, we prove a lemma for each meta function with this evaluator. The form `(my-eval term a)` represents evaluation of `term` for any binding alist `a`, which represents bindings for any free variable that might appear in `term`. The first lemma `rm-dp-for-s-is-correct` states that the updated term by our function `rm-dp-for-s` will evaluate to the same value as the original term when both appear as an argument of `s`. The second lemma states that the evaluation of `term` will be the same when it is updated by our second function `s-of-repeated-meta-fn`. This lemma is saved as a meta rule in ACL2 so that this function is triggered whenever a new instance of `s` is encountered during rewriting.

Example 2. *Verification of meta functions and the special* `defthm` *call to register the meta function as a meta rule*

```
(defevaluator my-eval my-eval-lst ((s a) (i+ a b)))

(defthm rm-dp-for-s-is-correct
  (equal (s (my-eval (rm-dp-for-s term) a))
         (s (my-eval term a))))

(defthm s-of-repeated-meta
  (equal (my-eval term a)
         (my-eval (s-of-repeated-meta-fn term) a))
  :rule-classes ((:meta :trigger-fns (s))))
```

Clause Processors Similar to meta rules, clause processors [16] can rewrite terms with user-defined functions. There are two main differences between meta rules and clause processors. First, clause processors do not require a trigger function, but they rewrite disjunctive clauses representing a given conjecture. Second, clause processors may return stronger/more general conjectures than the input. For example, a clause processor may drop some or all of the hypotheses of a given conjecture.

ACL2 has a very capable built-in rewriter but it is not developed and optimized for conjectures that can grow into very large terms. For this reason, we developed a custom rewriter, called RP-Rewriter [27], to more efficiently deal with large terms from conjectures for multiplier designs (e.g., Listing 2). RP-Rewriter is designed and used as a clause processor. This rewriter imitates some of the features of ACL2's built-in rewriter but also implements some optimizations for large terms (e.g., fast-alist support [27]) and some rewriting features that are needed by our term-rewriting algorithm (e.g., *side-conditions* feature as discussed in Sec. 4.2). RP-Rewriter is a verified clause processor, that is, all the rewriting performed on a conjecture is guaranteed to be sound. We let *only* RP-Rewriter manipulate target conjectures for multiplier designs and never the built-in rewriter by default. An existing set of rewrite/meta rules in ACL2 can be easily adapted to work with RP-Rewriter. Interested readers may find more information about RP-Rewriter in author's previous work [27]. That work details RP-Rewriter but does not deliver in-depth discussions about how our multiplication framework is implemented using this rewriter.

Clause processors can be used with other clause processors as well by concatenating clause-processor hints (see the related ACL2 documentation [6]). For example, in cases where RP-Rewriter cannot conclude a correctness proof, we can use the FGL system [24] as a clause processor to send the simplified term to an external SAT solver. This can help generate counterexamples or possibly conclude the proofs if the given conjecture is correct.

2.2 The Term-Rewriting Algorithm for Multiplier Designs

The target designs for our verification algorithm are various integer multiplier designs coded in a hardware description language such as Verilog. Integer multiplier design algorithms, such as Wallace-tree, build circuits using unit adders, such as half/full-adders, and vector adders, such as a carry-lookahead adder. Verilog designs often implement these adders with a design hierarchy, that is, the instances of all the adder modules are distinguishable from the rest of the circuit. Our verification algorithm takes advantage of this property, and we simplify and rewrite adder modules before attempting to verify the target multiplier module.

Table 3: Targeted final forms of the verification algorithm for some modules/functions

Function	out_2	out_1 / c_{out}	out_0 / s_{out}
Half-adder	-	$c(a+b)$	$s(a+b)$
Full-adder	-	$c(a+b+c_{in})$	$s(a+b+c_{in})$
Bit-vector addition $a+b$	$s(a_2+b_2 \; +c(a_1+b_1 \; +c(a_0+b_0)))$	$s(a_1+b_1 \; +c(a_0+b_0))$	$s(a_0+b_0)$
Bit-vector multiplication $a*b$	$s(a_0b_2+a_1b_1+a_2b_0 \; +c(a_1b_0+a_0b_1 \; +c(a_0b_0)))$	$s(a_1b_0+a_0b_1 \; +c(a_0b_0))$	$s(a_0b_0)$

Definition 1. *Functions s and c are defined as follows.*

$$\forall x \in \mathbb{Z}\ s(x) = mod_2(x)$$
$$\forall x \in \mathbb{Z}\ c(x) = \left\lfloor \frac{x}{2} \right\rfloor$$

The first step of our verification algorithm is to rewrite the output of adder modules in terms of the `s` and `c` functions (see Def. 1). For each distinct adder module, we prove that their output can be represented in terms of `s`, `c` and `+` as given in Table 3. Since all the adder modules are hierarchical in the target designs, they are all replaced by these forms when later symbolically simulating the multiplier module.

After finding appropriate representations for adder modules, we start simplifying the main multiplier design. Multipliers generally consist of two parts: partial product generation and summation. We have two different set of lemmas to rewrite and simplify these segments.

Partial product generation algorithms, such as Booth encoding, may involve some optimizations to reduce gate-delay. These optimizations may result in very complex Boolean expressions. We rewrite such terms with a set of lemmas of the form *lhs* = *rhs*, where terms matching *lhs* are replaced with *rhs* with appropriate term bindings. Lemmas 1, 2, 3 perform algebraic rewriting to reduce the logical expressions into terms involving only $+$, $-$ and $\wedge$ operators.

Lemma 1. $\forall x \in \{0,1\}\ \neg x = 1-x$

Lemma 2. $\forall x,y \in \{0,1\}\ x \vee y = x+y-xy$

Lemma 3. $\forall x,y \in \{0,1\}\ x \oplus y = x+y-xy\ -xy$

The majority or all of partial products are passed to adder modules as inputs in multiplier designs. Since adder modules are rewritten in terms of `s` and `c`, we can see expressions from partial products as arguments of the `s` and `c` functions. We simplify certain occurrences of these functions using Lemmas 4, 5, 6, 7.

Lemma 4. $\forall x,y \in \mathbb{Z}\ s((-x)+y) = s(x+y)$

Lemma 5. $\forall x,y \in \mathbb{Z}\ c((-x)+y) = (-x)+c(x+y)$

Lemma 6. $\forall x,y \in \mathbb{Z}\ s(x+x+y) = s(y)$

Lemma 7. $\forall x,y \in \mathbb{Z}\ c(x+x+y) = x+c(y)$

The other major components of multiplier designs are partial product summation trees. Summation trees consist of a large web of adder modules. As we rewrite adder modules in terms of the `s` and `c` functions, we derive complex expressions in terms of these functions. We simplify such expressions using Lemmas 8 and 9.

Lemma 8. $\forall x, y \in \mathbb{Z}\ s(s(x)+y) = s(x+y)$

Lemma 9. $\forall x, y \in \mathbb{Z}\ c(s(x)+y) = c(x+y) - c(x)$

These lemmas help reduce expressions derived from symbolic simulation of multiplier designs to a fixed final form as seen in Table 3. We can convert the specification of a design into the same form and conclude our proofs with a syntactic comparison.

Additional details about this algorithm may be found in our previous work [29, 30], where readers may find a more thorough explanation and examples as to how this algorithm reduces complex multiplier designs to a fixed form. These studies also show that this algorithm is applicable to many different design strategies (e.g., signed/unsigned Booth Encoding; Wallace, Dadda, and other summation trees), scales almost linearly with circuit size and delivers consistent results across different architectures. It can verify even 1024x1024-bit multipliers or similarly sized dot-product designs within minutes. What these publications lack, however, are key implementation details that make this algorithm efficient and verifiable.

3 Implementation Overview

An input design needs to be represented with suitable semantics in ACL2 in order to state a conjecture and verify its functionality. Since our multiplier verification algorithm utilizes design hierarchy, we use the SVL system [25] that retains hierarchy while simulating a design, which makes it possible to rewrite instances of adder modules when they are used as submodules. The SVL system takes advantage of the industrially used VL and SV toolkits [9] to parse Verilog modules. This makes it possible for the SVL system to parse advanced modules but it may not support modules with large control logic that can be found in industrial designs. Thankfully, our setup makes it possible to use other semantics as well [29] such as the industrial-design-friendly SVTV system [9].

Before working on a multiplier module, we state rewrite rules about its adder modules so that their output can be rewritten in terms of the `s` and `c` functions. Listing 1 shows a simplified version of such a rewrite rule for the SVL semantics. This rule will be used when we later symbolically simulate the main multiplier module. Instead of `s` and `c`, we use their logically equivalent functions `s-spec` and `c-spec`, which will trigger meta functions to apply our term-rewriting algorithm.

Listing 1: A simplified rewrite rule for a full-adder module. This rule will be applied when verifying a multiplier module.

```
(defthm full_adder_is_correct
  (implies (and (bitp a)
                (bitp b)
                (bitp cin))
           (equal (svl-run (list a b cin) <full_adder>)
                  (list (s-spec (+ a b cin))
                        (c-spec (+ a b cin))))))
```

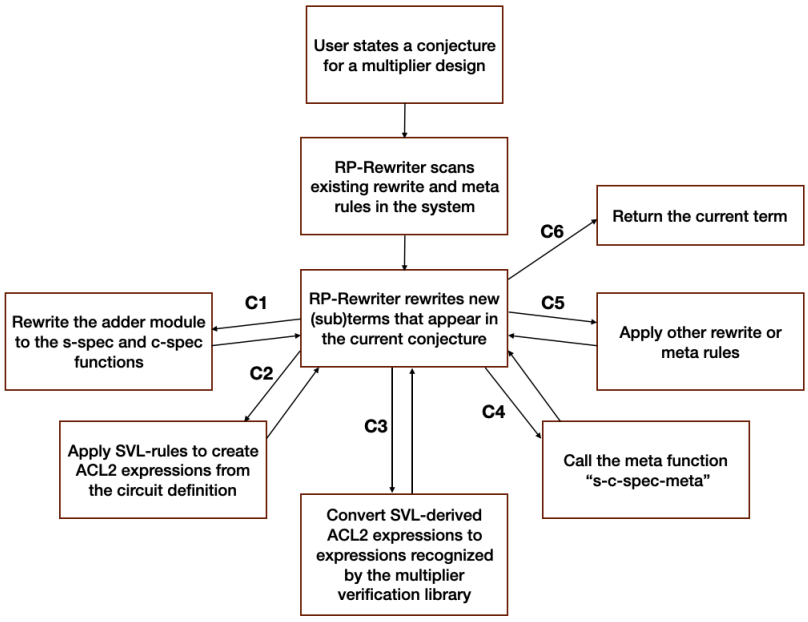


Figure 1: Term rewriting flow when verifying a multiplier design. Transition conditions: (C1) the current term is an instance of svl-run-phase of an adder module; (C2) An instance of an SVL simulation function; (C3) An instance of an ACL2 expression appeared as a result of SVL-rules; (C4) An instance of the s-spec or c-spec functions; (C5) Some other instance which may be rewritten by some rewrite/meta rules; (C6) There is no applicable rule.

After creating rewrite rules for all the adder modules, we state the desired conjecture for the main multiplier design. Listing 2 shows an example of such a conjecture and the event to submit to ACL2 in order to have our system simplify and prove it. This conjecture contains two free variables `a` and `b` that represent the two bit-vectors that the design takes as input. The left-hand side is the symbolic simulation of a design in SVL format with these free variables. In the actual program, `svl-run` returns an *alist* of output signals; however, for brevity, assume that it returns the value of the only output signal here. The right-hand side is the specification. We provide a clause processor hint to `defthm`, which will call RP-Rewriter [27] to simplify and verify such conjectures. As discussed in Sec. 2.1, RP-Rewriter is a verified clause processor with certain optimizations for large terms and it will be used to apply our term-rewriting algorithm.

Listing 2: A simplified correctness conjecture for a signed 64x64-bit multiplier with SVL semantics

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl-run (list a b) <signed_64x64_mult>)
                  (loghead 128 (* (logext 64 a)
                                  (logext 64 b))))))
  :hints (("Goal" :clause-processor (rp-cl))))
```

Fig.1 shows the rewriting flow when simplifying a multiplier design conjecture. Transition conditions C1-6 are ordered with respect to their application priority. Whenever one of these conditions is met on

the currently rewritten term, the associated rule(s) apply. These conditions and the rewriting scheme are developed in a way that allows a single pass rewriting without any expensive global search/lookup on terms when proving multipliers correct. We describe each of these transition conditions below, first describing C2 and C3 before C1 in an effort to help the readers understand our system more easily.

C2 When the rewriter starts working on the given conjecture, such as the one in Listing 2, it works from inside-out. It starts with the `svl-run` instance. This function and its subroutines have a logical definition in ACL2. With these definitions, we use rewrite and meta rules to quickly expand such `svl-run` instances into regular ACL2 terms that represent the symbolic simulation of a given design. For example, if a given design implements bitwise logical NAND of two vectors, `a` and `b`, then these rules would rewrite an instance of `(svl-run (list a b) <dummy-NAND>)` to `(4vec-bitnot (4vec-bitand a b))`. Here, the `4vec-bitand` and `4vec-bitnot` functions perform bitwise operations on four-valued ('0', '1', 'X', and 'Z') bit-vectors. We will refer to these rules as *SVL-rules*. SVL-rules are defined independently from our multiplier verification library and can work with any design to convert them to regular ACL2 terms defined with `4vec` functions.

C3 The resulting terms from SVL-rules are specific to the SVL and its parent libraries. We define a small set of bridge rules to convert such terms into expressions that our multiplier verification library can recognize. For example, `binary-and` is a function that performs logical AND on two-valued ('0' or '1') numbers and it is defined in our multiplier verification library. These bridge rules convert `(4vec-bitand a b)` to `(binary-and a b)` if we know through the current context that `a` and `b` are two-valued numbers.

C1 In ACL2, a more recently defined rewrite rule has priority over previous rules (i.e., it will be tried first), which is how we rewrite instances of adder modules in terms of the `s-spec` and `c-spec` functions. When the SVL functions simulate a module, it makes recursive calls for the simulation of sub-modules. For example, when `(svl-run (list a b) <signed_64x64_mult>)` is expanded, the rewriter might generate many instances of this form: `(svl-run (list (f1 a) (f2 b)) <half_-adder>)`. When users define a rewrite rule (e.g., Listing 1), the rewriter can replace this instance with `(list (s-spec (f1 a) (f2 b)) (c-spec (f1 a) (f2 b)))` instead of expanding the half adder module with SVL-rules to generate `4vec` functions. In other words, such rewrite rules for adder modules override the transition condition C2. After such rewriting, our multiplier-specific rules can start applying our term-rewriting algorithm.

C4 As new instances of `s-spec` and `c-spec` are created, a meta rule implementing our term-rewriting algorithm (see Sec. 2.2) is triggered. This meta rule calls a large set of meta functions that applies the described term-rewriting algorithm. Our meta functions efficiently search for the patterns from Lemmas 1-9 and rewrite terms accordingly. Some of those meta functions are given in Example 1. A more detailed discussion of these functions can be found in author's dissertation [28].

When we started developing this term-rewriting algorithm, we used rewrite rules to quickly test what rewriting strategy works for our goal and what does not. After we came up with the lemmas that we observed to work best, we slowly implemented the lemmas as meta functions to achieve better proof-time performance. Implementing our algorithm as customized meta functions has helped us identify bottlenecks in proof-time performance. We improved the time and memory performance by optimizing the functions as guided by diligent debugging and profiling.

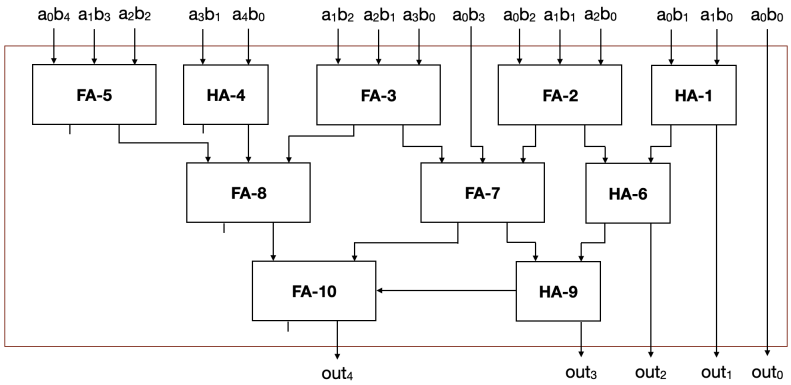


Figure 2: A partial circuit diagram of a 5x5-bit Wallace-tree multiplier demonstrating its directed acyclic graph structure

C5 When rewriting the left-hand side of the conjecture in Listing 2 finishes, we end up with a simplified term consisting of primarily `s` and `c` instances. An example for the resulting term is given in the last row of Table 3. After the left-hand side, the rewriter starts working on the right-hand side (specification of the design). We include rewrite rules in our library to rewrite the built-in multiplication function (shown with `*`) to the same form in Table 3. Finally, the rewriter compares the two sides with a syntactic check. If the two sides are equivalent, then the conjecture is proved. Otherwise, further processing (e.g., debugging, utilizing other verification methods on the simplified term) may be needed.

Our system provides flexibility to the user as to how the conjecture can be defined and how our multiplier verification system can be used for other designs, e.g., multipliers with saturation. This means that the conjecture may have user-defined functions and they may require their own rewrite/meta rules, or users may need to improve the system with additional rules. To accommodate for such cases, our rewriter checks for other applicable rules in the system, and applies them when they are present.

C6 If there are no other applicable rules for the current term, or if the current term is a constant (a quoted value), then the rewriter returns the term.

4 Challenges

Developing a verifiable and efficient software presents certain challenges, such as limited flexibility when using various data structures and subtle problems resulting from proof obligation. In this section, we discuss some of these challenges and our solutions.

4.1 Data Structure

Tree Representation of Terms ACL2's built-in rewriting system, as well as RP-Rewriter, represent and parse terms as a tree. For example, when parsing and rewriting the term `(f (g x) (g x) y)`, the duplicated subterms `(g x)` would be processed separately assuming no caching mechanism is enabled. This may not cause any concern for small terms; however, a design that may be represented with a directed acyclic graph (DAG) may yield an exponentially large term. Without caution, this can cause resource allocation to grow exponentially when verifying such designs.

Fig. 2 shows a partial circuit diagram for a Wallace-tree multiplier. We see that starting from out_3, data from the inputs flow through a directed acyclic graph structure with multiple parents. When this structure is reduced to the available tree structure to represent the functionality of each output bit, some subterms are repeated. This DAG structure persists for larger multipliers and tree representation of these designs can grow exponentially with design size.

Memoization or a rewrite-cache system may be used to rewrite flattened DAG structures to trees; however, we have found that it was not a beneficial option for multiplier verification for numerous reasons. First, efficiently managing a cache table is a difficult problem. A large number of steps are taken when rewriting multiplier designs, and caching all the rewriting may come at a substantial cost for memory allocation and run time. Developing a smart method to choose what to cache may or may not be feasible. Second, the memoization system in ACL2 looks for a *hit* only by checking the addresses of objects but not their actual value because comparing large terms while looking for a cache hit can be very expensive. This can decrease the number of matches and the system may be unable to prevent repeated rewriting. ACL2 has a system to normalize the addresses of objects, called `honsing` [2]. Every time a new object is created, this utility registers it in a table and prevents copies from being allocated. This can circumvent this memoization problem but `honsing` itself comes at a cost as well: through our multiplier verification experiments, we found that it slows down the program significantly while increasing memory usage. This was an expected outcome as terms change very frequently as rewriting progresses.

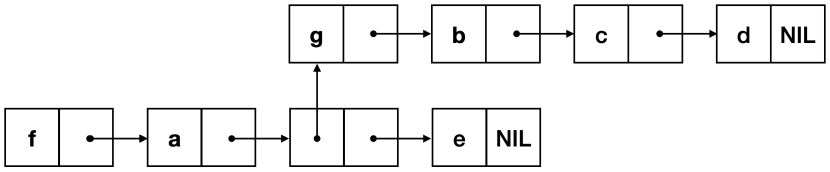
Another workaround could be using a rewriter that retains the graph structure (DAG-aware rewriting). However, such a system may bring about its own challenges, and memory and time performance may decline. The current tree representation of terms in our system makes it easier to develop and verify meta functions and a rewriter with a more complicated data structure might make this process more tedious.

In order to prevent repeated rewriting of the same terms resulting from a DAG, we start rewriting designs with their original structure and we perform a single pass of rewriting as discussed in Sec. 3. The SVL format preserves the original DAG structure of input designs. We unwind designs one node at a time when rewriting the `svl-run` instance in our target conjectures (see Listing 2). This system gradually builds the final term representing the functionality of each output signal. During this process, we apply our term-rewriting algorithm when an applicable term appears. When the design has been processed completely using the SVL semantics, we produce a term fully rewritten in accordance with our term-rewriting strategy.

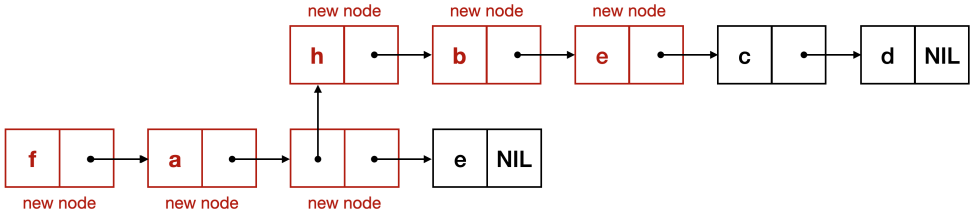
When using other semantics, the user needs to be aware of this issue and may need to implement a similar mechanism. For example, we implemented a work-around for the SVTV system [9] to overcome this problem. The SVTV system represents the functionality of designs with honsed expressions that are called *SVEX*es. SVEXes are intended to be interpreted using some memoization and honsing techniques. For our rewriter, these expressions can seem exponentially large because of all the repeated nodes in the tree form. We implemented a rewrite rule mechanism, that converts these SVEXes into a different form of representation (*SVEXL*, whose source code is located under books/centaur/svl/svexl in ACL2 distribution [1]). SVEXL system marks the repeated nodes in a fashion that is clearly visible to the rewriter. That way, we obtain a similar mechanism to the SVL when using the SVTV system.

.

Unmodifiable Linked-lists ACL2 terms are represented with a pointer-based linked-list structure. Fig. 3a shows how an example term `(f a (g b c d) e)` is represented in memory. Each node is a Common LISP `cons` pair, and the first and the second element/pointer in each node can be accessed with `car` and `cdr`, respectively.



(a) Linked-list representation of an example term `(f a (g b c d) e)`



(b) Nodes created after rewriting the example term in (a) to `(f a (h b e c d) e)`

Figure 3: Representation of ACL2 terms in memory

Despite this linked list structure, ACL2 forbids many common linked-list programming strategies, such as updating a node's value without deallocating/allocating new nodes. Instead, whenever a node in a linked list is to be updated, every node up to the target node is discarded and new nodes are created. For example, if we would like to replace the term `(f a (g b c d) e)` with `(f a (h b e c d) e)`, then we would have to reallocate at least 6 new nodes even though the only changes are updating the value of an existing one and adding a new one. Fig. 3b marks these new nodes.

The reason behind this structure is basically to keep the ACL2 world sound while the programming and verification procedures remain relatively simple. Assume that a term is copied into two places. If an ACL2 program updates nodes in one of the copies without discarding and creating new nodes, then the other copy would be updated as well even when it is not intended. It is not an easy task to keep track of such situations and prove that it is done correctly. For this reason, ACL2 does not allow mutable linked-lists. This programming limitation can be difficult to adapt for programmers who make extensive uses of data structures in other high-level languages, such as C++. During simple rewriting, it is not trivial to create O(1) queues, insert new elements inside a linked-list without discarding nodes, and use arrays. Using `stobj` or `fast-array` structures to achieve the desired behavior might be an option but they can complicate the development and verification process of user-defined meta functions.

Profiling of the rewriter and meta functions shows that the majority of time is spent in functions that allocate memory and only a negligible amount of time is spent in computation intensive functions. In order to achieve optimal performance, we consider this data structure throughout our system. For example, we have worked on several optimizations when merging summation lists. When rewriting multiplier designs, our program creates many terms that represent summation of partial products as well as `c` and `s` terms. Some of the lemmas, such as Lemma 8, may cause two summation terms to be merged. We try to keep expressions in a canonical form, where we keep all the summation lists lexicographically sorted. For example, if we are summing the terms `(i+ a (i+ b c))` and `(i+ d e)`, then the final term should be `(i+ a (i+ b (i+ c (i+ d e))))`. Simple commutativity rules would use bubble-sort to create the final form. At every step of this sorting, many nodes would be discarded and a new linked-list would be created. We can, however, use a meta function to *merge* two lists by iterating the two lists only once because we know that the input summation lists are always sorted.

Additionally, we divide the summations into three different groups for each `s`, `c`, and partial product terms. For instance, instead of defining the function `c` as `(c args)` where `args` may represent the summation term of the arguments; we define `c` as `(c s-args pp-args c-args)`, where `s-args` may represent summation of only `s` arguments, `pp-args` only partial product arguments, and `c-args` only `c` arguments (note that in the actual implementation the `s` and `c` functions also have another argument *hash* as discussed in Sec. 4.3 but it is omitted in this section). Separating elements to such different lists enables the program to discard and allocate fewer nodes when inserting an element to summation arguments. Such optimizations that focus on the linked-list structure and its limitations can reduce memory allocation substantially. In our experiments, we have seen that our optimizations for summation lists could improve the performance (both memory allocation and time) by almost 100-fold for some Booth-Encoded 64x64-bit multipliers.

4.2 Side-conditions

Throughout our rewriting, it is often the case that terms may change so extensively that it may become very difficult to prove some properties about them. In our previous work [27], we introduced RP-Rewriter's side-conditions feature that enables users to attach properties to terms and retain them throughout rewriting. This is one of the key features of our term-rewriting system. In this section, we discuss in detail how we use the side-condition feature during multiplier design correctness proofs.

The full/half-adder modules from Fig. 2 can be rewritten in terms of the `s` and `c` functions through a rewrite rule of the form given in Listing 1 (they are actually rewritten to their logical equivalent `s-spec` and `c-spec` functions to trigger meta rules, but we will refer to only `s` and `c` functions in this section). This rule has `bitp` hypotheses for each input signal, which indicates that we can rewrite unit adders in terms of `s` and `c` only when the inputs satisfy `bitp`. Rewriting starts from the top of the DAG (Fig. 2) and finishes towards the output. Therefore, we first apply this rewrite rule for `FA-2` and `FA-3` modules. Then, we move to `FA-7`. When attempting to apply the rewrite rule for `FA-7`, we have the following term bindings:

`a` to `(c (+ a0b2 a1b1 a2b0))` (carry output from `FA-2`),

`b` to `a0b3`,

`cin` to `(s (+ a1b2 a2b1 a3b0))` (sum output of `FA-3`).

We can easily relieve the `bitp` hypotheses of this rewrite rule for these bindings, and for the carry output of `FA-7` we obtain:

```
(c (+ (c (+ a0b2 a1b1 a2b0))
      a0b3
      (s (+ a1b2 a2b1 a3b0))))
```

This term will be automatically simplified by our term-rewriting algorithm as described in Sec. 2.2. In this case, only Lemma 9 will apply. The resulting term is:

```
(+ (c (+ (c (+ a0b2 a1b1 a2b0))
         a0b3
         a1b2 a2b1 a3b0))
   (- (c (+  a1b2 a2b1 a3b0))))
```

This term will be one of the inputs when our program tries to rewrite `FA-10`. This time, it will not be as easy to relieve the `bitp` hypothesis. Even if we came up with a smart rule/set of rules to prove that this term satisfies `bitp`, terms can change so significantly that the cost of backchaining can be very high.

On the other hand, we know that the resulting forms for each output signal of unit adders, such as the carry output `(c (+ a b cin))`, always satisfy `bitp`. We use RP-Rewriter's side-condition feature [27] to remember this property while we rewrite adder modules in terms of `s` and `c` (see Listing 1). To attach the side-condition, we rewrite the carry output to `(rp 'bitp (c (+ a b cin)))` instead, where `rp` is logically defined as an identity function always returning the second argument. RP-Rewriter has an invariant for such `rp` terms and it *r*etains the *p*roperty that the term in the second argument always satisfies `bitp` no matter how much it might change later. Therefore, when we rewrite the `FA-7` instance, we obtain:

```
(rp 'bitp (c (+ (c (+ a0b2 a1b1 a2b0))
                a0b3
                (s (+ a1b2 a2b1 a3b0)))))
```

for the carry output. Our program will rewrite this term the same way, and but this time we obtain:

```
(rp 'bitp (+ (c (+ (c (+ a0b2 a1b1 a2b0))
                   a0b3
                   a1b2 a2b1 a3b0))
             (- (c (+  a1b2 a2b1 a3b0)))))
```

This term will be one of the inputs to `FA-10`, and the rewriter can quickly relieve the `bitp` hypothesis with the attached side-condition. With this system, we can relieve the hypotheses for all adder modules without having to backchain or worry about using any extra rules about `bitp`. The side-conditions feature presents an O(1) and easily implementable solution.

4.3 Comparison of Large Terms

As multiplier designs are simplified, large terms might be frequently compared against each other either for lexicographical sorting, to apply lemmas such as Lemma 6, or to cancel terms in summations (i.e., $a + (-a) = 0$). Comparing two large terms might consume a considerable amount of proof-time. We have implemented a *term-hashing* mechanism to overcome this issue and quickly compare two large terms.

For the `s`, `c` functions, we add a logically extraneous argument, a hash value calculated with respect to immediate subterms. The signature of these functions are `(s hash args)` and `(c hash args)`, where `hash` is ignored in the logical definitions. The other argument `args` consists of summation of other `s`, `c` and `binary-and` (from partial products) instances. One of our meta functions traverses the terms in `args` and using their hash values (all terms are expected to have a hash value), it calculates a value for the parent `s` and `c` instances. We follow a similar procedure to calculate hash values for `binary-and` instances as well.

These hash values help our program compare two terms much more rapidly. If the two hash-codes are different, the program never dives into large terms and the terms are quickly known to be not syntactically equivalent. If the task is to lexicographically sort terms, then it only compares the hash values. If the hash values are the same, it may mean that the two terms are syntactically equivalent, then the program dives into the subterms and compares the whole terms to each other to see if that is indeed the case.

Table 4 shows our experiment results when the term-hashing feature is enabled and disabled. We have tested the feature for 110 different benchmarks [7, 19, 26] for various sizes and different architectures: Wallace-tree like multipliers, array multipliers, simple partial products (SP) and Booth Encoded partial products (BP). Tests were performed on a iMac Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz with 32GB system memory. When disabled, we force hash values to be 0 for all terms. We see that this

Table 4: Average proof-time results for a total of 110 different benchmarks for various multiplier sizes when the term-hashing feature is enabled and disabled

Term-hashing	64x64		128x128		256x256	
	SP	BP	SP	BP	SP	BP
Disabled (secs)	1.02	2	8.12	21.15	101.9	256.6
Enabled (secs)	0.53	1.04	2.22	4.02	11.8	18.7
Speed-up	1.9x	1.9x	3.6x	5x	8.6x	13.7x

feature helps the program scale better with growing design size and it has resulted in 92% (13x) speed-up on average for 256x256-bit multipliers.

5 Final Words and Conclusions

We have implemented a term-rewriting algorithm for multiplier design verification on an interactive theorem prover. Our implementation outperforms other state-of-the-art tools for formal verification of integer multipliers. In our previous works [29, 30], we introduced this term-rewriting algorithm and showed that it scales very well with growing circuit sizes (i.e., proof-time grows 4–6 times when the circuit grows 4 times); its performance is very consistent across different benchmarks as tested for over 20 different architectures including various Wallace-tree like and Booth Encoded signed/unsigned multipliers; it is so efficient that verifying various 64x64-bit multipliers takes less than 2 seconds on average, and verifying 1024x1024-bit multipliers takes around 5 minutes. Additionally, since our method is applied using a generic rewriter, it delivers a familiar interface and flexibility for modifications; therefore it allows designs with different specifications to be verified, and different semantics to be used. This tool has been used to verify various designs including multiply-accumulate, dot-product, multipliers with saturation, round-and-scale, truncated or right-shifted multipliers, as well as integer multipliers used in floating-point designs. Even though we have only described its use with the SVL system in this paper, our tool can work with the industrially capable SVTV system, and has been able to verify real-world designs at Intel Corporation and Centaur Technology [29].

Other systems with the closest performance [13, 18] do not scale as well as our program, extending the proof-time to hours for large multipliers. They may deliver inconsistent results for different architectures, and they are developed for isolated multipliers only. Even though it might be possible to include these tools in other frameworks, embedded multipliers often are not constructed with isolated multipliers [29]. For example, a multiply-accumulate design (MAC) might be built as follows. Partial products are created for multiplicand and multipliers. Then, instead of summing the partial products and obtaining the multiplication result first and adding the addend later, the addend and the partial products are summed together in the same summation tree (e.g., a Dadda tree) to calculate the final result of the MAC. It may not be possible to easily extract an isolated multiplier instance from such a design to use the competing tools. This integrated design procedure can be followed for complex chip designs to reuse a large multiplier component for different operations, such as dot-product and parallel multiplication [29]. Similarly, users may not easily extract an isolated multiplier from such designs and therefore may not be able to use tools developed only for isolated multipliers in their framework.

Our implementation in ACL2 provides substantially better results and a soundness guarantee. However, developing a competitive and verified program on a theorem prover has presented its own challenges. Our experience teaches that understanding the basic methods of term-rewriting (e.g., rewrite

rules, meta rules, and clause processors), data structures (e.g., tree representation of terms, unmodifiable linked-lists), and some rewriter features (e.g., side-conditions) are essential for creating and verifying such a system. Our implementation of a multiplier verification algorithm serves as an example of how an interactive theorem prover can be used to create large-scale efficient and sound programs that can successfully compete with other state-of-the-art tools programmed in high-level languages.

References

[1] *ACL2 System and Community Books*. Available at `https://github.com/acl2/acl2`.

[2] Robert S. Boyer & Warren A. Hunt (2006): *Function Memoization and Unique Object Representation for ACL2 Functions*. In: *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and Its Applications*, ACL2 '06, Association for Computing Machinery, New York, NY, USA, p. 81–89, doi:10.1145/1217975.1217992.

[3] Randal E. Bryant & Yirng-An Chen (1994): *"Verification of Arithmetic Functions with Binary Moment Diagrams"*. In: *DAC 1994*, doi:10.21236/ada281028.

[4] Jerry R. Burch (1991): *Using BDDs to Verify Multipliers*. In: *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, Association for Computing Machinery, New York, NY, USA, p. 408–412, doi:10.1145/127601.127703.

[5] Maciej Ciesielski, Tiankai Su, Atif Yasin & Cunxi Yu (2019): *Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, doi:10.1109/tcad.2019.2912944.

[6] ACL2 Community (Accessed Jan 1, 2021): *ACL2+Books Documentation*. Available at `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html`.

[7] Naofumi Homma, Yuki Watanabe, Takafumi Aoki & Tatsuo Higuchi (2006): *Formal Design of Arithmetic Circuits Based on Arithmetic Description Language*. *IEICE Transactions* 89-A, pp. 3500–3509, doi:10.1109/ispacs.2006.364918. Available at `https://www.ecsis.riec.tohoku.ac.jp/topics/amg/`.

[8] Warren A. Hunt, Matt Kaufmann, Robert Bellarmine Krug, J. Strother Moore & Eric Whitman Smith (2005): *Meta Reasoning in ACL2*. In Joe Hurd & Tom Melham, editors: *Theorem Proving in Higher Order Logics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 163–178, doi:10.1007/11541868_11.

[9] Warren A. Hunt, Matt Kaufmann, Moore, J S. & Anna Slobodova (2017): *Industrial Hardware and Software Verification with ACL2*. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375(2104), p. 20150399, doi:10.1098/rsta.2015.0399.

[10] Warren A. Hunt, Sol Swords, Jared Davis & Anna Slobodova (2010): *Use of Formal Verification at Centaur Technology*. In David Hardin, editor: *Design and Verification of Microprocessor Systems for High-Assurance Applications*, Springer, pp. 65–88, doi:10.1007/978-1-4419-1539-9_3.

[11] Christian Jacobi, Kai Weber, Viresh Paruthi & Jason Baumgartner (2005): *Automatic Formal Verification of Fused-Multiply-Add FPUs*. In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, IEEE Computer Society, USA, p. 1298–1303, doi:10.1109/DATE.2005.75.

[12] Roope Kaivola & Naren Narasimhan (2002): *Formal Verification of the Pentium ® 4 Floating-Point Multiplier*. In: *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*, pp. 20–27, doi:10.1109/DATE.2002.998245.

[13] D. Kaufmann, A. Biere & M. Kauers (2019): *Verifying Large Multipliers by Combining SAT and Computer Algebra*. In: *2019 Formal Methods in Computer Aided Design (FMCAD)*, pp. 28–36, doi:10.23919/FMCAD.2019.8894250.

[14] Matt Kaufmann & J. Strother Moore (2010): *ACL2 and Its Applications to Digital System Verification*. In David S. Hardin, editor: *Design and Verification of Microprocessor Systems for High-Assurance Applications*, Springer, pp. 1–21, doi:10.1007/978-1-4419-1539-9_1.

[15] Matt Kaufmann & J. Strother Moore (2010): *ACL2 and Its Applications to Digital System Verification*. In David S. Hardin, editor: *Design and Verification of Microprocessor Systems for High-Assurance Applications*, Springer, pp. 1–21, doi:10.1007/978-1-4419-1539-9_1.

[16] Matt Kaufmann, Jstrother Moore, Sandip Ray & Erik Reeber (2009): *Integrating External Deduction Tools with ACL2*. *J. Applied Logic* 7, pp. 3–25, doi:10.1016/j.jal.2007.07.002.

[17] Alireza Mahzoon, Daniel Große & Rolf Drechsler (2018): *PolyCleaner: Clean your Polynomials before Backward Rewriting to verify Million-gate Multipliers*. *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, doi:10.1145/3240765.3240837.

[18] Alireza Mahzoon, Daniel Große & Rolf Drechsler (2019): *RevSCA: Using Reverse Engineering to Bring Light into Backward Rewriting for Big and Dirty Multipliers*. In: *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, ACM, New York, NY, USA, pp. 185:1–185:6, doi:10.1145/3316781.3317898.

[19] Alireza Mahzoon, Daniel Große & Rolf Drechsler (2019): *SCA Multiplier Generator GenMul*. Available at `http://www.sca-verification.org`.

[20] J. Strother Moore & Marijn J. H. Heule (2017): *Industrial Use of ACL2: Applications, Achievements, Challenges, and Directions*. In Giles Reger & Dmitriy Traytel, editors: *ARCADE 2017, 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements, Gothenburg, Sweden, 6th August 2017*, *EPiC Series in Computing* 51, EasyChair, pp. 42–45, doi:10.29007/dh3f.

[21] David M. Russinoff (2019): *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, doi:10.1007/978-3-319-95513-1.

[22] Amr Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken & Rolf Drechsler (2016): *Formal Verification of Integer Multipliers by Combining Gröbner Basis with Logic Reduction*. In: *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Research Publishing Services, pp. 1048–1053, doi:10.3850/9783981537079_0248.

[23] Anna Slobodova, Jared Davis, Sol Swords & Warren A. Hunt (2011): *A Flexible Formal Verification Framework for Industrial Scale Validation*. In: *Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, IEEE/ACM, Cambridge, UK, pp. 89–97, doi:10.1109/memcod.2011.5970515.

[24] Sol Swords (2020): *New Rewriter Features in FGL*. In Grant O. Passmore & Ruben Gamboa, editors: *Proceedings of the Sixteenth International Workshop on the ACL2 Theorem Prover and its Applications, Worldwide, Planet Earth, May 28-29, 2020*, *EPTCS* 327, pp. 32–46, doi:10.4204/EPTCS.327.3.

[25] Mertcan Temel (2019): *ACL2 SVL Documentation*. Available at `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____SVL`.

[26] Mertcan Temel (2019): *Fast Multiplier Generator*. Available at `https://github.com/temelmertcan/multgen`.

[27] Mertcan Temel (2020): *RP-Rewriter: An Optimized Rewriter for Large Terms in ACL2* 327. doi:10.4204/eptcs.327.5.

[28] Mertcan Temel (2021): *Automated, Efficient, and Sound Verification of Integer Multipliers*. Ph.D. thesis, The University of Texas at Austin.

[29] Mertcan Temel & Warren A. Hunt (2021): *Sound and Automated Verification of Real-World RTL Multipliers*. In: *2021 Formal Methods in Computer Aided Design (FMCAD)*, pp. 53–62, doi:10.34727/2021/isbn.978-3-85448-046-4_13.

[30] Mertcan Temel, Anna Slobodova & Warren A. Hunt (2020): *Automated and Scalable Verification of Integer Multipliers*. In Shuvendu K. Lahiri & Chao Wang, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 485–507, doi:10.1007/978-3-030-53288-8_23.

[31] Cunxi Yu, Maciej Ciesielski & Alan Mishchenko (2018): *Fast Algebraic Rewriting Based on And-Inverter Graphs*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37(9), pp. 1907–1911, doi:10.1109/tcad.2017.2772854.