


# VeSCMul: Verified Implementation of S-C-Rewriting for Multiplier Verification

Mertcan Temel 

Intel Corporation, Austin, TX, USA  
mert.temel@intel.com

**Abstract.** Formal verification of multipliers is difficult. This paper presents a custom tool, VeSCMul, designed to address this problem. VeSCMul can be effectively applied to a wide range of hardware verification challenges, including multipliers with saturation, flags, shifting, truncation, accumulation, dot product, and even floating-point multiplication. The tool is highly automated with a user-friendly interface, and it is very efficient; for instance, verification for designs with 64-bit operands can finish in seconds. Notably, VeSCMul has been successfully utilized for both commercial designs and publicly available benchmarks. Regarding the reliability of its results, VeSCMul itself is fully verified, instilling confidence in its users for soundness. It also has the option to be used with a SAT solver for completeness and counterexample generation. Readers of this paper will gain insights into the capabilities and limitations of VeSCMul, as well as how to employ it for the verification of their own designs.

**Keywords:** Multipliers · Hardware Verification · Formal Methods

## 1 Introduction

Integer multipliers are crucial components in processing units. Ensuring their correctness through formal verification is essential; however, historically, verifying them has proven to be challenging [4,6,15,18]. Automated methods like SAT solving, BDDs, and computer algebra systems have either failed to scale or demonstrated limited applicability in this context [2,8,12,16,25]. On the other hand, the S-C-Rewriting method has been shown to be very efficient in formally verifying a large variety of RTL designs [21,24,25,26].

S-C-Rewriting and auxiliary programs are packaged into the VeSCMul tool (pronounced “vesk-muhl”). VeSCMul is designed to be user-friendly and comprehensive for sound, fast, and automatic verification of multiplier-centric RTL designs. It has an improved user interface tailored for non-experts, simplifying tool usage. VeSCMul has also introduced the support for fully automatic verification with its new adder detection program. VeSCMul has undergone extensive testing on thousands of public benchmarks as well as proprietary industrial designs at Intel Corporation. Its open-source and free-license status enables others to use this tool for similar verification tasks.

This paper presents VeSCMul, and it is outlined as follows. Sec. 2 walks through a demo for VeSCMul, showing the user-interface. Sec. 3 gives an overview of the tool flow. Sec. 4 lists some of the noteworthy features. Sec. 5 delivers experimental results on both public and proprietary designs. Sec. 6 discusses related work and concludes the paper.

## 2 Installation and a Demo

VeSCMul is implemented in the ACL2 theorem prover and programming language [10], and it is fully verified. VeSCMul is open-source with the MIT license, included as a Community Book in the ACL2 distribution on Github, which can be found at <https://github.com/acl2/acl2> under `books/projects/vescmul`. Installing ACL2 and building the books will bring along VeSCMul.

A comprehensive and up-to-date documentation for VeSCMul is available as part of ACL2’s manual, accessible at <http://acl2.org/manual>. This documentation is extensive, covering thousands of topics from ACL2 sources and Community Books. Throughout this paper, various documentation topics are referenced using the notation “:doc <topic>”.

Once ACL2 is installed and books are built, users can run a VeSCMul demo by running the events from Listing 1.1 within an ACL2 interactive session.

Listing 1.1: Simple demo running VeSCMul on a signed 64x64-bit multiplier with Booth radix-4 encoding, Dadda tree, and Han-Carlson adder.

```
(include-book "projects/vescmul/top" :dir :system)

(vescmul-parse
 :name      my-multiplier-example
 :file      "DT_SB4_HC_64_64_multgen.sv"
 :topmodule "DT_SB4_HC_64_64")

(vescmul-verify
 :name      my-multiplier-example
 :concl     (equal RESULT
                  (loghead 128 (* (logext 64 IN1)
                                   (logext 64 IN2))))))
```

The first event (`include-book`) loads VeSCMul and required libraries, which takes about a minute. Alternatively, an executable can be created for instant loading (see :doc `save-exec`). The second event (`vescmul-parse`) parses the target design, taking a few seconds. The Verilog file is available in the ACL2 git repository under the `books/projects/vescmul/demo` directory. The third event (`vescmul-verify`) uses VeSCMul to verify the design. `:concl` specifies the conjecture, with `RESULT` as the output signal name, and `IN1` and `IN2` as input signal names. `logext` sign-extends bit-vectors (represented as integers), and `loghead` zero-extends or, in other words, truncates them. The inputs are 64-bit signed numbers, producing a 128-bit multiplication result. VeSCMul can fully verify this design in 1-2 seconds (as tested on a Macbook M1 pro).

### 3 Tool Flow

The `vescmul-parse` and `vescmul-verify` utilities are two LISP macros that invoke various programs to parse and then verify target designs.

The `vescmul-parse` macro packs VL/SV/SVTV utilities to parse Verilog designs and create symbolic simulation vectors. These utilities are publicly available and come with the ACL2 installation. They have been developed and used in industry (i.e., Centaur Technology and Intel Corporation) (see `:doc sv`).

The `vescmul-verify` macro gathers the symbolic simulation objects, detects adder components, applies the S-C-Rewriting algorithm, and maybe utilizes SAT solving in the end. The program flow is shown in Fig. 1. These steps are explained as follows.

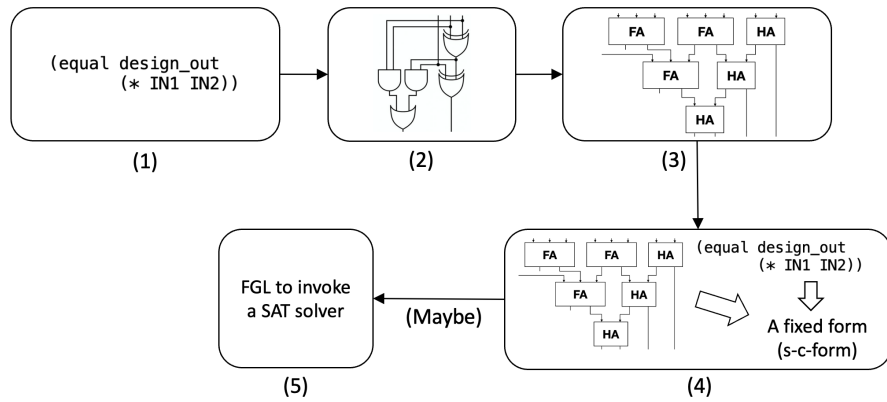


Fig. 1: Flow chart of `vescmul-verify`. (1) User states a conjecture with high-level specification. (2) VeSCMul receives a sea of gates from the design. (3) The tool identifies and rebuilds half/full-adders in this sea of gates. (4) The design and the spec are rewritten with the S-C-Rewriting methodology. (5) If rewriting is not conclusive, rewritten conjecture can be passed to FGL for SAT solving.

(1) Specification is provided by the user, stating a relation between input and output signals. This is typically a combination of multiplication (`*`), addition (`+`), subtraction (`-`), truncation/zero-extension (`loghead`), sign-extension (`logext`), part selection (`part-select`), and possibly user-defined functions.

(2)(3) S-C-Rewriting algorithm needs to differentiate and specially rewrite adder components (e.g., full/half-adders) in a design. In previous work [25,26], S-C-Rewriting algorithm was used only for designs whose design hierarchy information around adders was readily available. VeSCMul has been improved to now support flattened designs. This is achieved by an internal program that goes through a sea of gates to identify and mark the adder components before applying the S-C-Rewriting algorithm. Tests have shown that this program works very well for successful verification of various architectures (see Sec. 5). Should

the program not identify some adders and the verification attempt fails because of that, users may also pass hierarchical verification hints (see `:doc vescmul`).

(4) When VeSCMul applies S-C-Rewriting, the rewriter tries to rewrite both the specification and the design to the same form (i.e., *s-c-form* [25]), and the two sides are compared syntactically. For correct multiplier designs, this is usually enough to prove the conjecture.

(5) If S-C-Rewriting cannot show the conjecture to be correct, it returns its rewritten form. Users have the option to automatically use the FGL utility [19] (see `:doc fgl`) that can bit-blast the rewritten conjecture, perform AIG transformations, and invoke an external SAT solver like CaDiCaL [1]. FGL is also a verified program. This can either generate counterexamples for false conjectures, or help finalize the proofs in some fringe cases. For example, in x86 multiplier designs, extra circuitry is used to calculate flags based on multiplication results, such as the overflow flag that is set when a certain portion of the result are not homogeneously 0s or 1s. VeSCMul by itself may not be able to process the extra flag logic; however, it can rewrite and simplify the multiplication component, send the rewritten expression to an external SAT solver through FGL, and finalize such proofs in a matter of seconds or minutes. Note that if the multiplication component is not rewritten as intended by S-C-Rewriting, it is unlikely for a SAT solver to scale and finish the proofs for operand sizes greater than 16-bits.

## 4 Notable Features and Compatible Tools

This section highlights some of the useful and noteworthy features of VeSCMul as well as compatible tools.

**Customizable specification:** Users can state their own specifications to verify various multiplier configurations such as multiply-add, dot product, and multipliers with shifted, truncated and/or saturated outputs.

**Automatic adder detection:** VeSCMul includes an adder-detection program that identifies and marks adders before employing the S-C-Rewriting algorithm. This makes the overall verification procedure fully automatic for a large variety of multiplier designs (see Sec. 5 for experiments).

**End-to-end verified:** The author has rigorously verified, using ACL2, that VeSCMul’s all rewriting operations on given conjectures are sound. Users can place high confidence in the results when a design is claimed to be correct. Verifying such a substantial program is a complex process, demanding ACL2 expertise [20,21,22].

**Exporting a clean multiplier with design hierarchy:** The included adder-detection program can be used as a stand-alone feature. Given a flattened multiplier design, VeSCMul can export a functionally equivalent Verilog module with adder components separated as half/full-adder submodules. This feature may be particularly useful for researchers addressing the multiplier verification problem, where adder detection can be a common challenge [7,11,12,14]. For soundness, VeSCMul includes a mechanism for formal equivalence checking between the original and exported designs.

**Integration into other verification flows:** Proofs generated by VeSCMul can be integrated into other ACL2-based verification workflows. For instance, when verifying floating-point fused-multiply-add (FMA) operations, which often involves decomposing the design into integer multiplication and post-multiplication parts, VeSCMul can be used for the multiplication part while SAT solving can be employed for the rest. Existing and actively used decomposition tool flows in ACL2 (see `:doc decomposition-proofs`) and VeSCMul are compatible.

**Verification of sequential circuits:** VeSCMul can handle sequential circuits, including pipelined designs. Additional key arguments can be provided to `vescmul-parse` to verify such designs (see `:doc vescmul-parse`). Modules with control logic reusing the same circuitry for various arithmetic operations (e.g., see `:doc multiplier-verification-demo-2`) are also supported.

**Waveform generation:** VeSCMul is compatible with another tool (see `:doc svtv-debug`) for generating waveforms in the VCD format. This capability can be valuable for pinpointing the cause of bugs in case of counterexamples.

## 5 Experiments

VeSCMul has undergone extensive testing and utilization across various architectures in both public benchmarks and proprietary x86 processor design projects at Centaur Technology and Intel Corporation.

Various benchmarks are gathered for experiments using publicly available generators [3,13,23]<sup>1</sup>. Summation trees include Dadda (dt), Wallace (wt), 4-to-2 compressor (4:2), array (ar), redundant binary addition (rbat), balanced delay (bdt), overturned stairs (os) trees. Partial products include signed/unsigned (s/u) simple (sp), Booth radix-2 (b2), radix-4 (r4), radix-8 (r8), radix-16 (r16) encodings. Final stage adders include block carry lookahead (bcla), carry lookahead (cla), carry-select (csel), Ladner-Fischer (lf), carry-skip (csk), conditional sum (cond), Brent-Kung (bk), ripple-carry (rp), Kogge-Stone (ks), Han-Carlson (hc), J. Sklansky conditional (jsk) adders.

Table 1 contains a large number of benchmarks to compare the performance of VeSCMul to other prominent verification tools: AMulet [8] and RevSCA2 [12] that target  $n \times n$ -bit multipliers with  $2n$ -bit results. The newest version of AMulet (AMulet2) timed out for the majority of the benchmarks; the owner is notified, and AMulet1 is used in the experiments instead. The results for AMulet1 includes the time to check for proof certificates. RevSCA2 is neither a verified program nor does it produce certificates to check its results. These experimental results show that VeSCMul scales much better for large multipliers.

In addition to standard input/output sizes ( $n \times n$ -bit multipliers with  $2n$ -bit results), Table 2 includes VeSCMul’s verification results for variations such as multiply-add (e.g.,  $64 \times 64 + 64$ ), multipliers with asymmetric operand sizes (e.g.,  $10 \times 1024$ ), shifted/truncated outputs (e.g.,  $64 \times 64[95:32]$  returns the output bit positions from 32 to 95), and dot product (e.g.,  $8(16 \times 16) + 32$  is an

<sup>1</sup> All tests are available at <https://temelmertcan.github.io/mult-experiments.html>, or the peer-reviewed artifact is available at <https://zenodo.org/records/10048797>

Table 1: Proof-time results with success rates for a large set of  $n \times n$ -bit multipliers

Op. Size	PP	Benchmarks	RevSCA2 [12]	AMulet1 [7]	VeSCMul
32x32	sp	48	0.5s (77%)	0.4s (100%)	0.5s (100%)
	r2	48	0.8s (62%)	1.4s (100%)	0.7s (100%)
	r4	48	1.4s (87%)	1.3s (91%)	0.6s (100%)
	r8	48	241s (44%)	TO (0%)	0.7s (100%)
	r16	48	TO (0%)	TO (0%)	1.9s (100%)
64x64	sp	54	11s (77%)	1.9s (100%)	1.7s (100%)
	r2	48	17s (62%)	32s (100%)	2.6s (100%)
	r4	240	19s (75%)	4.9s (88%)	2.8s (90%)
	r8	48	1630s (19%)	TO (0%)	2.7s (100%)
	r16	48	TO (0%)	TO (0%)	8s (100%)
128x128	sp	54	83s (65%)	11s (100%)	6.6s (100%)
	r2	48	356s (52%)	928s (100%)	10.1s (100%)
	r4	48	642s (50%)	274s (91%)	8.4s (100%)
	r8	48	TO (0%)	TO (0%)	11s (100%)
	r16	48	TO (0%)	TO (0%)	37s (100%)
256x256	sp	48	2501s (65%)	82s (100%)	27s (100%)
	r4	48	TO (0%)	9529s (91%)	33s (100%)
512x512	r4	6	TO (0%)	TO (0%)	138s (100%)
1024x1024	r4	6	TO (0%)	TO (0%)	776s (100%)

Multiplier sizes range from 32x32 to 1024x1024, grouped wrt. partial product algorithm. Total of 1032 different benchmarks are used and the timing results of successful proof attempts are averaged. The tools could not verify all the benchmarks and the success ratios are given in parentheses. VeSCMul is used only for fully automatic verification (without a SAT solver), but it can verify the missing cases with user-provided hints. Time-out (TO) is set to 3600 seconds (1 hour) for up to 128x128; 16200 seconds (4.5 hours) for the rest. Collected on Intel<sup>®</sup> E-2378G CPU, 32GB memory.

Table 2: Proof-time and memory allocation for various designs

Arch.	Function	Time, Mem		Arch.	Function	Time, Mem
dt-ub4-bcla	64x64	2.1s, 0.3GB		4:2-ub4-cla	64x64	9.7s, 0.7GB
ar-sb4-csel	64x64	2.0s, 0.3GB		rbat-sb4-lf	64x64	2.4s, 0.3GB
bdt-sb4-csk	64x64	2.4s, 0.3GB		os-sb4-cond	64x64	1.8s, 0.3GB
dt-ssp-bk	64x64	1.7s, 0.3GB		ar-usp-rp	64x64	1.0s, 0.2GB
4:2-ub4-ks	64x64	3.7s, 0.5GB		4:2-ub8-lf	64x64	3.4s, 0.5GB
dt-sb16-hc	64x64	8.0s, 1.9GB		wt-ub16-bk	64x64	8.3s, 1.9GB
dt-ssp-bk	128x128	5.9s, 1.0GB		4:2-ub4-hc	128x128	13.3s, 1.8GB
wt-usp-lf	256x256	28s, 4.4GB		dt-sb4-jsk	256x256	27s, 4.4GB
dt-sb4-jsk	512x512	130s, 19GB		dt-sb4-jsk	1024x1024	725s, 83GB
dt-sb4-ks	10x1024	32s, 5.1GB		dt-sb4-ks	1024x10	32s, 5.7GB
dt-sb2-bk	64x64+64	2.5s, 0.4GB		wt-sb4-lf	64x64[63:0]	0.9s, 0.2GB
wt-sb4-lf	64x64[95:32]	1.8s, 0.3GB		wt-sb4-lf	64x64[127:64]	2.2s, 0.4GB
wt-sb8-bk	8(16x16)+32	2.0s, 0.3GB		dt-sb4-ks	4(32x64)+128	5.2s, 1.1GB

8-point dot product with 16-bit operands accumulating onto a 32-bit number). Comparable verification tools do not support these configurations. VeSCMul can fully automatically verify these designs without user hints or SAT solvers.

Moreover, around 7500 different multiplier designs with diverse architectures, operand sizes, operations, truncation, and shifting were randomly generated [23]. Overall, VeSCMul achieved a 98% success rate for fully automatic verification without hints or SAT solvers. The remaining 2% is mostly made up of multipliers with a special 7-to-3 compressor tree, and shifted multipliers, but they could still be verified by VeSCMul with a user-provided design hierarchy hint.

VeSCMul has also proven successful in industrial designs, particularly for Intel x86 instructions with various functional configurations, including multiply-accumulate, dot product, output shifting/truncation, flag calculations based on multiplication results, and saturation. In some cases, the assistance of a SAT solver becomes necessary (for flags and saturation). These designs can be fully verified rapidly and automatically, with results similar to those in the public designs. To the best of the author’s knowledge, VeSCMul is the first tool to achieve comparable verification tasks scalably and automatically.

Additionally, VeSCMul has played a vital role in the verification flow of floating-point multiply and fused-multiply-add operations. Verifying these designs is notably challenging, with no known fully automated verification method. We employ decomposition techniques [5,17], where VeSCMul is used for the multiplication part, significantly reducing manual effort. Complete verification of single and double precision operations can be completed in under an hour.

## 6 Related Work and Conclusion

AMulet [8], RevSCA2 [12], and DyPoSUB [14] are other state-of-the-art tools for multiplier verification. Like VeSCMul, AMulet prioritizes soundness and can produce proof certificates. In contrast, RevSCA2 and DyPoSUB lack such proofs or mechanisms, and DyPoSUB has been identified as unsound [9]. Additionally, these tools primarily focus on verifying  $n \times n$ -bit multipliers with  $2n$ -bit results. On the other hand, VeSCMul stands out by offering scalable and automatic verification for a broader range of multiplier-centric arithmetic circuits, and it allows users to specify their conjectures. These target designs can encompass regular multipliers, multiply-add operations, dot products, and operations involving shifting, truncation, accumulation, and saturation.

This paper has showcased VeSCMul for multiplier verification, which has demonstrated favorable results in experiments involving both public and proprietary RTL designs. This tool is open-source and compatible with other hardware verification tools. It has an improved user-interface tailored for ACL2 novices. The tool itself is fully verified, so users can have a high level of confidence in its soundness. Future work includes adding support for more input formats (currently limited to System Verilog) such as AIGER and DIMACS CNF, and further enhancements in automation to handle corner-case designs that currently require user hints for verification.

## References

1. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froylyks, N., Heule, M., Iser, M., Jarvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
2. Ciesielski, M., Su, T., Yasin, A., Yu, C.: Understanding Algebraic Rewriting for Arithmetic Circuit Verification: a Bit-Flow Model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019). <https://doi.org/10.1109/tcad.2019.2912944>
3. Homma, N., Watanabe, Y., Aoki, T., Higuchi, T.: Arithmetic module generator (AMG) (2006), <https://www.ecsis.riec.tohoku.ac.jp/views/amg-e>
4. Hunt, W.A., Swords, S., Davis, J., Slobodova, A.: Use of Formal Verification at Centaur Technology. In: Hardin, D. (ed.) *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pp. 65–88. Springer (2010). [https://doi.org/10.1007/978-1-4419-1539-9\\_3](https://doi.org/10.1007/978-1-4419-1539-9_3)
5. Jacobi, C., Weber, K., Paruthi, V., Baumgartner, J.: Automatic formal verification of fused-multiply-add FPUs. In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 2*. p. 1298–1303. DATE '05, IEEE Computer Society, USA (2005). <https://doi.org/10.1109/DATE.2005.75>
6. Kaivola, R., O’Leary, J.: Verification of Arithmetic and Datapath Circuits with Symbolic Simulation, pp. 1–52. Springer Nature Singapore, Singapore (2022). [https://doi.org/10.1007/978-981-15-6401-7\\_37-1](https://doi.org/10.1007/978-981-15-6401-7_37-1)
7. Kaufmann, D., Biere, A., Kauers, M.: Verifying Large Multipliers by Combining SAT and Computer Algebra. In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. pp. 28–36 (Oct 2019). <https://doi.org/10.23919/FMCAD.2019.8894250>
8. Kaufmann, D., Biere, A.: AMulet 2.0 for verifying multiplier circuits. In: Groote, J.F., Larsen, K.G. (eds.) *International Conference on Tools and Algorithms for the Construction and Analysis of Systems , TACAS 2021. Lecture Notes in Computer Science*, vol. 12652, pp. 357–364. Springer (2021). [https://doi.org/10.1007/978-3-030-72013-1\\_19](https://doi.org/10.1007/978-3-030-72013-1_19)
9. Kaufmann, D., Biere, A.: Fuzzing and delta debugging and-inverter graph verification tools. In: Kovács, L., Meinke, K. (eds.) *Tests and Proofs*. pp. 69–88. Springer International Publishing, Cham (2022). [https://doi.org/10.1007/978-3-031-09827-7\\_5](https://doi.org/10.1007/978-3-031-09827-7_5)
10. Kaufmann, M., Moore, J.S.: ACL2 and its applications to digital system verification. In: Hardin, D.S. (ed.) *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pp. 1–21. Springer (2010), [https://doi.org/10.1007/978-1-4419-1539-9\\_1](https://doi.org/10.1007/978-1-4419-1539-9_1)
11. Liew, V., Beame, P., Devriendt, J., Elffers, J., Nordström, J.: Verifying properties of bit-vector multiplication using cutting planes reasoning. In: *2020 Formal Methods in Computer Aided Design (FMCAD)*. pp. 194–204 (2020). [https://doi.org/10.34727/2020/isbn.978-3-85448-042-6\\_27](https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_27)
12. Mahzoon, A., Große, D., Drechsler, R.: 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*. pp. 185:1–185:6. DAC ’19, ACM, New York, NY, USA (2019). <https://doi.org/10.1145/3316781.3317898>
13. Mahzoon, A., Große, D., Drechsler, R.: SCA multiplier generator GenMul (2019), <https://github.com/amahzoon/genmul>



14. Mahzoon, A., Große, D., Scholl, C., Drechsler, R.: Towards formal verification of optimized and industrial multipliers. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 544–549 (2020). <https://doi.org/10.23919/DATE48585.2020.9116485>
15. Russinoff, D.M.: Formal Verification of Floating-Point Hardware Design: A Mathematical Approach. Springer (2019). <https://doi.org/10.1007/978-3-319-95513-1>
16. Sayed-Ahmed, A., Große, D., Kühne, U., Soeken, M., Drechsler, R.: 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). pp. 1048–1053. Research Publishing Services (2016). [https://doi.org/10.3850/9783981537079\\_0248](https://doi.org/10.3850/9783981537079_0248)
17. Slobodová, A.: Challenges for formal verification in industrial setting. In: Brim, L., Haverkort, B., Leucker, M., van de Pol, J. (eds.) Formal Methods: Applications and Technology. pp. 1–22. Springer Berlin Heidelberg, Berlin, Heidelberg (2007), [https://doi.org/10.1007/978-3-540-70952-7\\_1](https://doi.org/10.1007/978-3-540-70952-7_1)
18. Slobodova, A., Davis, J., Swords, S., Hunt, W.A.: 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). pp. 89–97. IEEE/ACM, Cambridge, UK (2011). <https://doi.org/10.1109/memcod.2011.5970515>
19. Swords, S.: New rewriter features in FGL. Electronic Proceedings in Theoretical Computer Science **327**, 32–46 (Sep 2020). <https://doi.org/10.4204/eptcs.327.3>
20. Temel, M.: RP-Rewriter: An optimized rewriter for large terms in ACL2. vol. 327, p. 61–74. Open Publishing Association (Sep 2020). <https://doi.org/10.4204/eptcs.327.5>
21. Temel, M.: Automated, Efficient, and Sound Verification of Integer Multipliers. Ph.D. thesis, The University of Texas at Austin (2021), <https://repositories.lib.utexas.edu/handle/2152/88056>
22. Temel, M.: Verified implementation of an efficient term-rewriting algorithm for multiplier verification on ACL2. International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2022) **359**, 116–133 (may 2022). <https://doi.org/10.4204/eptcs.359.11>
23. Temel, M.: Multgen: a fast multiplier generator (2023), <https://github.com/temelmertcan/multgen>
24. Temel, M.: Formal Verification of Booth Radix-8 and Radix-16 Multipliers. In: Design, Automation & Test in Europe Conference & Exhibition (DATE) (to appear) (2024)
25. Temel, M., Hunt, W.A.: Sound and automated verification of real-world RTL multipliers. In: Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19–22, 2021. pp. 53–62. IEEE (2021). [https://doi.org/10.34727/2021/isbn.978-3-85448-046-4\\_13](https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_13)
26. Temel, M., Slobodova, A., Hunt, W.A.: Automated and scalable verification of integer multipliers. In: Computer Aided Verification. pp. 485–507. Springer International Publishing, Cham (2020). [https://doi.org/10.1007/978-3-030-53288-8\\_23](https://doi.org/10.1007/978-3-030-53288-8_23)