Methodology for Formal Verification of Hardware Safety Strategies Using SMT

Anthony Faure-Gignoux, Kevin Delmas, Adrien Gauffriau, and Claire Pagetti

***Abstract*—Safety-critical embedded systems must maintain their functionality even in the presence of single permanent hardware failure. Naive redundancy of hardware is often unaffordable and impractical, therefore alternative strategies must be explored for minimal cost fault tolerance. The objective of this article is to propose a methodology to evaluate formally safety strategies using satisfiability modulo theory solvers. Practically, the approach consists in providing a bounded model checking demonstration applied to the formal model of hardware. We show the capabilities of the approach on an efficient hardware accelerator designed to perform parallel computations of matrix multiplications and convolutions.**

***Index Terms*—Bounded model checking (BMC), fault tolerance, SAT/satisfiability modulo theory (SMT).**

I. Introduction

With the need of more autonomy and more complex embedded functionalities (e.g., based on machine learning algorithms), safety-critical systems integrate more and more high-performance applications. This new trend has a direct impact on the hardware. Indeed, general purpose multicore processors are not adapted any more and additional accelerators, such as deep learning accelerators or GPU, are required.

Context: When designing safety-critical systems, hardware failures [1] that may occur during operations have to be addressed and their effect mitigated to forbid catastrophic consequences. The hardware failures are usually classified as either *transient* or *permanent*. In this article, we focus only on permanent failures. In the presence of such a failure, the system should be able to detect it and provide a way to mitigate effects that could lead to catastrophic consequences.

Contributions: We propose a methodology to ensure the correctness of hardware that implement arithmetical and logical operations and evaluate its mitigation strategy. The approach consists in formally modeling the hardware, the *intended functions*, the fault-detection and fault-tolerance strategies to be performed in the standard satisfiability modulo theory (SMT) description language SMTLib [2] with the bitvector theory. SMT [3] offers a powerful framework to formally model a problem as a set of constraints in first-order logic. Finally, we illustrate the proposed methodology on a state-of-the-art hardware.

Outline: This article is structured as follows. Section II details the proposed methodology. Section III applies the methodology on a case study. Section IV presents related works on formal verification and fault-tolerance strategies.

Manuscript received 26 July 2024; accepted 30 July 2024. This work was supported by PhD CIFRE Airbus. This manuscript was recommended for publication by A. Shrivastava. *(Corresponding author: Anthony Faure-Gignoux.)*

Anthony Faure-Gignoux and Adrien Gauffriau are with the 1YY, Airbus, 31300 Toulouse, France (e-mail: anthony.faure-gignoux@onera.fr).

Kevin Delmas and Claire Pagetti are with the DTIS, ONERA, 31000 Toulouse, France.

Digital Object Identifier 10.1109/LES.2024.3439859

II. Methodology

The objective is to provide a formal proof of the correctness of a hardware, even in the presence of a single permanent fault. This is achieved by verifying the correctness of fault-free execution and the effectiveness of the fault detection and fault tolerance strategies.

The hardware is correct if, for all possible inputs, it produces the same output as the *intended function*. The *intended function* expresses the mathematical operations expected to be performed by the hardware and acts as a reference (or a specification). We refer to the hardware to be verified as *HW* and the intended function to be matched as *IF*.

A. Formal Modeling

Hardware platforms perform operations on *binary words*. Thus, the formal modeling is performed using the bitvector theory of the standard SMT description language SMTLib [2]. This allows a fine-grained hardware model down to bit level. Definition 1 is a reminder about binary words.

Definition 1 (Binary Word): A binary word w is a finite sequence of bits. If the bitwidth of w is $n \in \mathbb{N}$, we refer to $w = w_{n-1} \dots w_0 \in \mathbb{B}^n$ as a n-bit word.

Modeling IF: IF is a function taking a succession of sets of input binary words and providing a set of output binary words (Definition 2).

Definition 2 (IF Model): Let *IF* be the model of the intended function. Let $n, k, \omega_\epsilon \in \mathbb{N}$. *IF* is defined as follows:

$$\begin{aligned} \text{IF} : \mathbb{B}^{n\times k\times \omega_\epsilon} &\rightarrow \mathbb{B}^{m\times l} \\ (X_i)_{i<\omega_\epsilon} &\rightarrow Y \end{aligned}$$

where ω_ϵ is the nominal (fault-free) number of execution cycles, $(X_i)_{i<\omega_\epsilon}$ is a sequence of k n-bit input words and Y is the vector of l m-bit output words.

Modeling HW: We only consider hardware that perform arithmetical and logical operations on vectors of bits (so called *bitvectors*). We also assume the number of execution cycles required to perform a given operation is known. The

hardware is configurable thanks to a configuration vector, called conf, the value of which is set by an external system. The configuration forces the hardware to execute a certain operation. Let $F = \{\epsilon, f_1, \ldots, f_n\}$ be the set of potential permanent faults on the hardware where ϵ is the absence of fault. Definition 3 provides the description of the hardware model in the presence of fault or not.

Definition 3 (HW$_f$ Model): Let HW_f be the model of the hardware. Let $m, l, n, k, p \in \mathbb{N}$. HW_f is defined as follows:

$$\begin{aligned} \mathrm{HW}_f : \mathbb{B}^{m\times l} \times \mathbb{B}^{n\times k} \times \mathbb{B}^{p} &\rightarrow \mathbb{B}^{m\times l} \\ s, X, \mathrm{conf} &\rightarrow s' \end{aligned}$$

where s is the current memory state, $X = x^0 \ldots x^{k-1}$ is composed of the k n-bit words $x^i = x^i_{n-1} \ldots x^i_0 \in \mathbb{B}^n$, conf $\in \mathbb{B}^p$ is the configuration vector, and s' is the new memory state.

Thus, to perform IF, the hardware executes ω_ϵ times with input $(X_i)_{i<\omega_\epsilon}$ and updates the internal memory state. After the ω_ϵ steps, the output of HW is the result.

Modeling Safety Strategy: The safety strategy is represented by the function π of Definition 4. This function transforms the succession of input binary words of the fault-free execution into a new succession of input binary words for the mitigated hardware in the presence of a permanent fault. The mitigation (or reconfiguration) is also computed by π as a new configuration for the hardware.

Definition 4 (Safety Strategy Model): Let $F = \{\epsilon, f_1, \ldots, f_n\}$ be the set of potential permanent faults. Let $\omega_\epsilon \in \mathbb{N}$ be the nominal (fault-free) number of execution cycles and ω_f the number of execution cycles of the reconfigured hardware in the presence of fault $f \in F$. The safety strategy π is defined as follows:

$$\begin{aligned} \pi : F \times \mathbb{B}^{n\times k\times \omega_\epsilon} &\rightarrow \mathbb{B}^{n\times k\times \omega_f} \times \mathbb{B}^{p\times \omega_f} \\ f, (X_i)_{i<\omega_\epsilon} &\rightarrow \left(X'_i\right)_{i<\omega_f}, (\mathrm{conf}_i)_{i<\omega_f} \end{aligned}$$

where $(X_i)_{i<\omega_\epsilon}$ is the nominal sequence of inputs and $(X'_i)_{i<\omega_f}$ the associated sequence of inputs for the hardware reconfigured by $(\mathrm{conf}_i)_{i<\omega_f}$ in the presence of f.

B. Correctness Verification Using SMT

The correctness verification aims at formally proving that HW is compliant with IF for all fault models. Theorem 1 provides the correctness verification problem based on bounded model checking [4] (BMC). The verification is performed by a SMT solver, and Z3 solver [5] for the use case.

Theorem 1 (Correctness Verification): Let $F = \{\epsilon, f_1, \ldots, f_n\}$ be the permanent faults and π the safety strategy: 1) let $I(s_0) \triangleq \forall b \in s_0, b = 0$ be the formula encoding the initial memory state value to 0; 2) let $T_f(s_i, s_{i+1}, X'_i, \mathrm{conf}_i) \triangleq s_{i+1} = HW_f(s_i, X'_i, conf_i)$ be the formula encoding the memory state update performed by HW$_f$ model; and 3) let $P(s_{\omega_f}) \triangleq s_{\omega_f} \neq IF((X_i)_{i<\omega_\epsilon})$ be the formula encoding the property comparing the final memory state of HW$_f$ with IF.

HW_f is compliant with IF for all $f \in F$ if and only if the following statement is *UNSAT*: $\exists f \in F, (X'_i)_{i<\omega_f}, \pi(f, (X_i)_{i<\omega_\epsilon}) = ((X'_i)_{i<\omega_f}, (\mathrm{conf}_i)_{i<\omega_f}) \wedge I(s_0) \wedge \bigwedge_{i<\omega_f} T_f(s_i, s_{i+1}, X'_i, \mathrm{conf}_i) \wedge P(s_{\omega_f})$

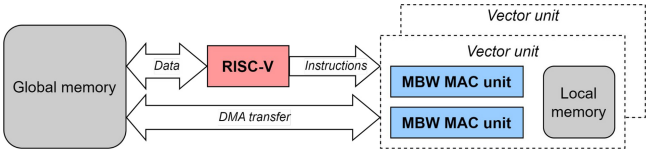

Fig. 1. ZuSE Ki-Avf accelerator overview. Excerpt from [7].

III. Case Study—ZuSE Ki-Avf Accelerator

We illustrate the proposed methodology on a state-of-the-art hardware: the *ZuSE Ki-Avf* [6] depicted in Fig. 1. The ZuSE Ki-Avf is composed of a *RISC-V* host processor and a cluster of vector processor units (two in the figure).

In this case study, we focus on the fault-tolerance strategy of a single multibit-width multiply-accumulate unit (MBW MAC) [7]. We have extended the design proposed in [7] to tolerate a single permanent fault on a multiplier. Fault-detection is performed by the host processor of the ZuSE Ki-Avf hardware and is therefore outside the scope of this article.

The MBW MAC unit takes two 16-bit words, a and b, as input and produces a 32-bit word, c, as output. The inputs remain unaltered throughout the execution process, as the fault is on the multiplier.

A. Preliminary Considerations

The *bit extension* operator is often implicit in high-level mathematical descriptions, but should be explicit when using binary words. Binary operations can result in larger binary words, and the bit extension operator allows to avoid overflows. Definition 5 gives the two bit extension operators.

Definition 5 (Bit Extension): Let $w = w_{n-1} \ldots w_0$ be a n-bit word and $k \in \mathbb{N}$. An extension translates w into a $(n+k)$-bit word. The unsigned extension is defined by $extU_k(w) = 0 \ldots 0 w_{n-1} \ldots w_0$ and the signed extension is defined by $extS_k(w) = w_{n-1} \ldots w_{n-1} \ldots w_0$.

Example 1 (Bit Extension): Let $w = 1001$ a 4-bit word. We have $extU_4(w) = 00001001$ and $extS_4(w) = 11111001$.

The MBW MAC extensively relies on subword manipulations. Property 1 gives the notation used for the subwords.

Property 1 (Subwords): If n is even, a n-bit word $w = w_{n-1} \ldots w_0$ can be represented as a pair of two $n/2$-bit words $w = [w_h; w_l]$ where $w_h = w_{n-1} \ldots w_{n/2}$ and $w_l = w_{n/2-1} \ldots w_0$. Indeed, $w = 2^{\lfloor n/2 \rfloor}(extS_{n/2}(w_h)) + extU_{n/2}(w_l)$ where $\lfloor n/2 \rfloor$ is the shift operator.

B. Formal Modeling

Modeling the Intended Function: First, we model the intended function. The accelerator can perform two mathematical operations [7]: 1) full-precision (FP) and 2) half-precision (HP). In this article, the FP computation becomes our intended function and is denoted by *IF* (Definition 6).

Definition 6 (IF Model): The model of the intended function. *IF* is formally defined as follows:

$$\begin{aligned} \mathrm{IF} : \left(\mathbb{B}^{16}\right)^2 &\rightarrow \mathbb{B}^{32} \\ a, b &\rightarrow \mathrm{extS}_{16}(a) \times \mathrm{extS}_{16}(b) \end{aligned}$$

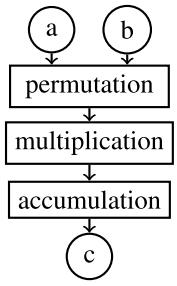

Fig. 2. Multibit-width MAC unit overview.

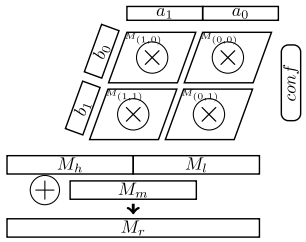

Fig. 3. Multiplication stage.

where $a = [a_h; a_l]$ and $b = [b_h; b_l]$ are the two 16-bit input words.

Modeling the Hardware: Second, we model the MBW MAC unit. We break down the MBW MAC unit into multiple stages (*permutation*, *multiplication* and *accumulation*) as illustrated in Fig. 2.

Definition 7 (MAC Model): Let *MAC* be the model of the MBW MAC unit. Let $\text{acc} \in \mathbb{B}^{32}$ be the current memory states and c be the new memory states. Let $X = a, b \in \mathbb{B}^{16\times 2}$ be the operand vector composed of two 16-bit words. Let $\text{conf} \in \mathbb{B}^{10}$ be the configuration vector composed of 10 Boolean signals. *MAC* is composed of three stages: 1) permutation *PERM*; 2) multiplication *MUL*; and 3) accumulation *ACC*. The *MAC* is the following:

$$\begin{aligned} \text{MAC} : \mathbb{B}^{32} \times \mathbb{B}^{16\times 2} \times \mathbb{B}^{10} &\to \mathbb{B}^{32} \\ \text{acc}, \quad a, b, \quad \text{conf} &\to \quad c \end{aligned}$$

$$c = \text{ACC}(\text{acc}, \text{MUL}(\text{PERM}(a, b, \text{conf}), \text{conf}), \text{conf}).$$

1) Permutation Stage:

Definition 8 (PERM Model): Let *PERM* be the model of the permutation stage. Let $a = [a_h; a_l], b = [b_h; b_l] \in \mathbb{B}^{16}$ be two 16-bit input words. Let $a_1, a_0, b_1, b_0 \in \mathbb{B}^8$ be the four 8-bit output subwords. Let $\text{SP} \in \text{conf}$ be the configuration signal that permutes the subwords of a. We have: $PERM(a, b, conf) = [a_1; a_0; b_1; b_0]$ with

$$a_1 = \begin{cases} a_h, \text{ if } \neg SP \\ a_l, \text{ else} \end{cases} \quad a_0 = \begin{cases} a_l, \text{ if } \neg\text{SP} \\ a_h, \text{ else} \end{cases} \quad \begin{aligned} b_1 &= b_h \\ b_0 &= b_l. \end{aligned}$$

2) Multiplication Stage: Fig. 3 illustrates that the multiplication stage comprises four multipliers, denoted as $M_{(i,j)}$. Definition 9 provides the formal definition of the multipliers, while Definition 10 formally describes the multiplication stage.

Definition 9 (Multipliers): Let $(M_{(i,j)})_{i<2,j<2}$ be a multiplier. Let $a_i, b_j \in \mathbb{B}^8$ be the 8-bit input subwords. Let $e_{i,j} \in \text{conf}$ be the configuration signal that enables $M_{(i,j)}$ and $s_{a_i} \in \text{conf}$ (resp. s_{b_j}) be the configuration signal that indicates whether a_i (resp. b_j) is signed or not. The 16-bit product is defined as follows:

$$M_{(i,j)} = \begin{cases} a'_i \times b'_j, \text{ if } e_{i,j} \\ 0, \qquad \text{else.} \end{cases}$$

With

$$a'_i = \begin{cases} \text{extS}_8(a_i), \text{ if } s_{a_i} \\ \text{extU}_8(a_i), \text{ else} \end{cases} \qquad b'_j = \begin{cases} \text{extS}_8(b_j), \text{ if } s_{b_j} \\ \text{extU}_8(b_j), \text{ else.} \end{cases}$$

Definition 10 (MUL Model): Let *MUL* be the model of the multiplication stage. Let $a_1, a_0, b_1, b_0 \in \mathbb{B}^8$ be the four 8-bit input subwords. Let $(M_{(i,j)})_{i<2,j<2}$ be a 16-bit partial product produced by the multiplier i, j. Let $M_r \in \mathbb{B}^{32}$ be the 32-bit multiplication result. Let $M_h, M_l \in \mathbb{B}^{16}$ and $M_m \in \mathbb{B}^{17}$ be intermediate results. Let $INV \in \text{conf}$ be the configuration signal that configure the wiring between the partial products and the intermediate results. The formal model is

$$\begin{aligned} &\text{MUL}([a_1; a_0; b_1; b_0], conf) = M_r \\ &= 2^{16}(\text{extS}_{16}(M_h)) + 2^8(\text{extS}_{16}(M_m)) + \text{extU}_{16}(M_l). \end{aligned}$$

If $\neg INV$: $M_h = M_{(1,1)}$, $M_m = M_{(0,1)} + M_{(1,0)}$, $M_l = M_{(0,0)}$. Otherwise: $M_h = M_{(0,1)}$, $M_m = M_{(1,1)} + M_{(0,0)}$, $M_l = M_{(1,0)}$.

3) Accumulation Stage: The operations of the *accumulation* stage are formally defined in Definition 11.

Definition 11 (ACC Model): Let *ACC* be the model of the accumulation stage. Let $M_r \in \mathbb{B}^{32}$ be the 32-bit input word. Let $\text{acc} = [\text{acc}_h; \text{acc}_l] \in \mathbb{B}^{32}$ be the current memory state. Let $c \in \mathbb{B}^{32}$ be the 32-bit output word which is also the new memory state of MBW MAC. Let $c_{acc} \in \text{conf}$ be the configuration signal to consider c as a 32-bit word or as $[c_1; c_0]$ two independent 16-bit subwords. We have $ACC(\text{acc}, M_r, \text{conf}) = c$, with

$$c = \begin{cases} 2^{16}(\text{acc}_h) + \text{acc}_l + M_r, \text{ if } \neg c_{acc} \\ [\text{acc}_h + M_h; \text{acc}_l + M_l], \text{ else .} \end{cases}$$

Modeling of the Safety Strategy: Third, we model the fault. In this article, we consider only a single permanent fault on a multiplier of the MBW MAC unit. The fault $f_{k,l}$ impacting the multiplier $M_{(k,l)}$ is captured by forcing $\neg e_{k,l} \in \text{conf}$. This capture method allows for a unique MAC model, i.e., $\forall f \in F, \text{MAC}_f = \text{MAC}$. The fault-detection is performed by the host processor which is out-of-scope. Thus, only the fault-tolerance strategy is modeled (Definition 12).

Definition 12 (Fault-Tolerance Strategy Model): Let π be the fault-tolerance strategy function. Let $F = \{\epsilon, (f_{k,l})_{k<2,l<2}\}$ be the set of potential permanent faults on the hardware, $f_{k,l}$ is a permanent fault on the multiplier k, l. Let $f \in F$ be the hardware fault, $(X_i)_{i<\omega_\epsilon}$ be the nominal input sequence. Let ω_f, $(\text{conf}_i)_{i<\omega_f}$, $(X'_i)_{i<\omega_f}$ be defined by π as follows:

$\forall i < \omega_\epsilon, X_i = X$ and $\forall i < \omega_f, X'_i = X$

If $f = \epsilon$, then $\omega_f = 2$ and $\text{conf}_0 = \text{conf}_1 = \{\neg SP, e_{1,1}, e_{1,0}, e_{0,1}, e_{0,0}, s_{a_1}, \neg s_{a_0}, s_{b_1}, \neg s_{b_0}, \neg c_{\text{acc}}, \neg INV\}$.

Otherwise, $\omega_f = 3$, $\text{conf}_0 = \{\neg SP, \neg e_{k,l}, (e_{i,j})_{i\neq k \vee j\neq l}, s_{a_1}, \neg s_{a_0}, s_{b_1}, \neg s_{b_0}, \neg c_{acc}, \neg INV\}$ and $\text{conf}_1 = \text{conf}_2 = \{SP, \neg e_{k,l}, (e_{i,l})_{i\neq k}, \neg(e_{i,j})_{j\neq l}, \neg s_{a_1}, s_{a_0}, s_{b_1}, \neg s_{b_0}, \neg c_{\text{acc}}, INV\}$.

C. Correctness Verification Using SMT

Once we have formally modeled the MBW MAC unit, the next step is to evaluate its correctness. Theorem 2 is the correctness verification problem applied to the MBW MAC model. The fault is applied by forcing the inhibition of a multiplier.

Theorem 2 (MAC Correctness): The inputs of *MAC* do not change during the execution, thus: $\forall i < \omega_\epsilon, X_i = X$. Let π be the fault-tolerance strategy defined by Definition 12, and providing ω_f and $(conf_i)_{i<\omega_f}$. Let $(\text{acc}_i)_{i<\omega_f}$ be the memory states: 1) let $I(\text{acc}_0) \triangleq \text{acc}_0 = 0$ be the initialization of the accumulator to 0; 2) let $T_f(\text{acc}_i, \text{acc}_{i+1}, X, \text{conf}_i) \triangleq \text{acc}_{i+1} = \text{MAC}(\text{acc}_i, X, conf_i)$ be the accumulation of the new multiplication result; and 3) let $P(\text{acc}_{\omega_f}) \triangleq \text{acc}_{\omega_f} \neq IF(X)$.

MAC is compliant with *IF* for all $f \in F$ if and only if the following statement is UNSAT: $\exists f \in F, X \in (\mathbb{B}^{16})^2$

$$I(\text{acc}_0) \wedge \bigwedge_{i<\omega_f} T_f(\text{acc}_i, X, \text{conf}_i) \wedge P\big(\text{acc}_{\omega_f}\big).$$

Z3 solver has performed the verification in 50 min on a PC with a processor i5-1245U 1.60 GHz and 16 Go RAM. The fault-tolerance strategy is effective. The complete model is given in an open source GitHub [8].

IV. Related Work

Hardware verification is a well-known and challenging problem that has received considerable attention in the literature. As highlighted by [9], ensuring the compliance of a hardware implementation with its specifications requires substantial effort. To tackle this problem, a first family of methods is based on intensive testing. Yet the complexity of hardware often makes exhaustive testing impractical due to the combinatorial explosion of possible scenarios. Another family of approaches is based on formal methods, which offer formal guarantees regarding the correctness of the hardware. A subset of these methods, discussed in a survey [10], encodes the verification problem as a SMT problem. Clarke et al. [11] provided numerous successful applications of formal methods in verifying hardware designs. A significant advancement in SMT solvers, compared to SAT solvers, lies in the integration of dedicated theory solvers capable of native reasoning on non-Boolean terms. The bitvector theory [12] has been extensively used to reason about hardware implementation of arithmetic operations. Therefore, we rely on a SMT-based encoding of the verification problem to check the correctness of the proposed fault-tolerant hardware accelerator.

V. Conclusion

We propose a twofold methodology to, first, formally model a hardware and ensure its correctness, and second, evaluate its safety strategies, including fault-detection and fault-tolerance strategies. The methodology is the following: 1) formally model the intended function; 2) formally model the hardware; 3) formally model the safety strategy; and 4) perform a BMC on a set of hardware model in presence of a permanent fault. We apply it on a simple architecture.

This work has two well-identified limitations. First, we have only applied the methodology to one simple hardware. Second, the application of the methodology is not scalable. Future work would be to apply the methodology to various more complex hardware, such as the versatile tensor accelerator [13] (VTA). Additionally, the construction of the formal model will be automated. The automated construction could be based on a *CHISEL* [14] hardware description language.

References

[1] S. Werner, J. Navaridas, and M. Luján, "A survey on design approaches to circumvent permanent faults in networks-on-chip," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 1–36, 2016.

[2] C. W. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard version 2.0," in *Proc. 8th Int. Workshop Satisf. Modulo Theories*, vol. 13, Edinburgh, U.K., 2010, p. 14.

[3] C. W. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Cham, Switzerland: Springer, 2018, pp. 305–343.

[4] O. Strichman, "Tuning SAT checkers for bounded model checking," in *Proc. Int. Conf. Comput. Aided Verif.*, 2000, pp. 480–494.

[5] L. M. de Moura and N. S. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Constr. Anal. Syst.*, 2008, pp. 337–340.

[6] G. B. Thieu et al., "ZuSE Ki-Avf: Application-specific AI processor for intelligent sensor signal processing in autonomous driving," in *Proc. Design, Autom. Test Eur. Conf. Exhib. DATE*, 2023, pp. 1–6.

[7] M. Beyer, S. Gesper, A. Guntoro, G. P. Vayá, and H. Blume, "Exploiting subword permutations to maximize CNN compute performance and efficiency," in *Proc. 34th IEEE Int. Conf. Appl.-Specif. Syst., Archit. Process. ASAP*, 2023, pp. 61–68.

[8] A. Faure-Gignoux, K. Delmas, A. Gauffriau, and C. Pagetti. "Formal verification MBW MAC." Accessed: Mar. 8, 2024. [Online]. Available: https://github.com/AnthonyFaureGignoux/formal_verification_MBW_MAC

[9] A. Wright, "Modular SMT-based verification of rule-based hardware designs," Ph.D. dissertation, Dept. Electr. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2021.

[10] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Trans. Design Autom. Electr. Syst.*, vol. 4, no. 2, pp. 123–193, 1999.

[11] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., *Handbook of Model Checking*. Cham, Switzerland: Springer, 2018.

[12] C. W. Barrett, D. L. Dill, and J. R. Levitt, "A decision procedure for bit-vector arithmetic," in *Proc. 35th Conf. Design Autom. Conf.*, 1998, pp. 522–527.

[13] T. Moreau et al., "A hardware-software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, Sep./Oct. 2019.

[14] J. Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," in *Proc. 49th Annu. Design Autom. Conf.*, 2012, pp. 1216–1225.