CoSA: Integrated Verification for Agile Hardware Design

Cristian Mattarei, Makai Mann, Clark Barrett, Ross G. Daly, Dillon Huff, and Pat Hanrahan
Stanford University
Stanford, California (USA)
{mattarei, makaim, clarkbarrett, ross.daly, dhuff, pmh}@stanford.edu

***Abstract*—Symbolic model-checking is a well-established technique used in hardware design to assess, and formally verify, functional correctness. However, most modern model-checkers encode the problem into propositional satisfiability (SAT) and do not leverage any additional information beyond the input design, which is typically provided in a hardware description language such as Verilog.**

In this paper, we present CoSA (CoreIR Symbolic Analyzer), a model-checking tool for CoreIR designs. CoreIR is a new intermediate representation for hardware. CoSA encodes model-checking queries into first-order formulas that can be solved by Satisfiability Modulo Theories (SMT) solvers. In particular, it natively supports encodings using the theories of bitvectors and arrays. CoSA is closely integrated with CoreIR and can thus leverage CoreIR-generated metadata in addition to user-provided lemmas to assist with formal verification. CoSA supports multiple input formats and provides a broad set of analyses including equivalence checking and safety and liveness verification. CoSA is open-source and written in Python, making it easily extendable.

I. Introduction

Formal verification has become an important part of the design process, particularly in the hardware domain. As hardware and software systems become increasingly complex, more time than ever before is spent on verification to avoid costly and potentially dangerous bugs.

For many years, hardware model-checking experts focused on general techniques applicable to any design provided in a standard format such as a hardware description language (HDL) or AIGER [6], without any extra information from the designers. While there has been impressive progress, these techniques still often fail to scale on industrial-sized systems. This requires verification engineers to either shrink the parameter sizes if possible, or manually add additional lemmas. Frequently, these additional lemmas are simple invariants which are known by the designer or design tool, but are not easily inferred by the formal system.

This paper introduces the CoreIR Symbolic Analyzer (CoSA), a model-checking tool for the hardware intermediate representation CoreIR [11]. CoSA can leverage additional knowledge provided by CoreIR to improve performance on many classes of proofs.

This research was supported in part by the Defense Advanced Research Projects Agency (contract FA8650-18-2-7854) and by gifts from Intel Corporation (through the Stanford Agile Hardware Project) and Cisco Systems.

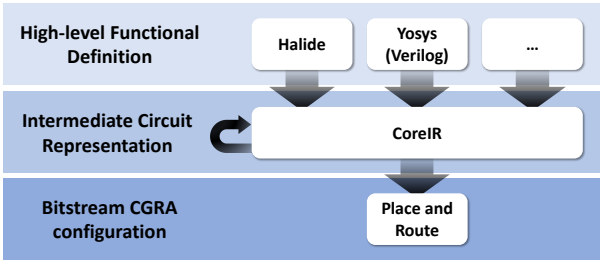

Fig. 1. AHA Flow

CoSA was developed as a tool for verifying correctness at various stages of the toolflow in the Agile Hardware (AHA) Project at Stanford University [18]. This project aims to improve performance and design productivity by incorporating ideas from agile software development to speed up the development cycle.

Compared to the software community, there are very few open-source tools for hardware design and verification. As seen in the software domain, open-source tools can help encourage innovation and distribute effort, the latter of which is particularly lacking in the hardware community. Furthermore, in the last decade, open-source SMT solvers have become powerful tools for verification, and the community no longer needs to rely exclusively on commercial tools. In support of these goals, the Agile Hardware Project is developing an end-to-end open-source toolchain.

The rest of the paper is organized as follows: Section II provides background on CoreIR and the Agile Hardware Project; Section III describes CoSA's supported formal analyses, architecture, and integration with design; Section IV describes a set of applications of the tool; Section V covers related work on hardware verification tools; and Section VI provides concluding remarks.

II. CoreIR

CoreIR is an intermediate representation and compilation framework for digital designs [11]. It is front-end agnostic and thus can be a compiler target for any language representing hardware designs. Primitives in the IR have the same semantics as the SMT theory of bitvectors [3], allowing for easy formal verification integration. CoreIR can be transformed into custom back-ends using a flexible pass framework, and serialized into different hardware and SMT-based formats.

In the AHA toolflow [18], depicted in Figure 1, a user first writes an application in a high-level language, such as the image processing domain-specific language, Halide [19]. This compiles to CoreIR and then goes through several optimization passes before being mapped to a back-end. One of the main targets of the AHA tool flow is a custom Course-Grained Reconfigurable Array (CGRA). The CGRA is designed to have the flexibility of an FPGA while improving performance on certain kinds of applications (e.g. image processing) [23]. This performance is gained by configuring at the word level and by composing specialized heterogeneous tiles containing memories and dedicated processing elements (essentially ALUs). A set of place and route tools produce a bitstream which configures the CGRA to implement the application.

As shown in Figure 1, other high-level hardware description languages can integrate with CoreIR in addition to Halide. In fact, the CGRA is written in Verilog, which is compiled into CoreIR using the VerilogToCoreIR [13] Yosys [25] pass. Another example is the hardware design language Magma [21].

The verification goals in the AHA project include assessing functional correctness of the CGRA, as well as verifying that the firmware produces the correct configuration for the high-level, behavioral definition from Halide. Given these requirements, we integrated the formal verification at the CoreIR level, thus allowing us to support the required analyses.

III. CoSA: CoreIR Symbolic Analyzer

CoSA integrates with CoreIR to provide formal analyses. In this section we explain the analyses supported by the tool and describe its architecture.

A. Formal Analyses

CoSA reduces all analyses to symbolic model-checking problems [10]. The underlying theoretic model is a Symbolic Transition System (STS), as expressed in Def. 1.

Def. 1 (Symbolic Transition System)**.** A *Symbolic Transition System* is a tuple $S = \langle V, I, T \rangle$ where V is a set of (input V_I, state V_S, and output V_O) variables, $I(V)$ is a formula representing the initial states, and $T(V, V')$ is a formula representing the transitions. A *state* of S is an assignment to the variables V_S.

The core analyses of CoSA are primarily based on *safety* and *liveness* checking. A *safety* property is a formula φ which should hold in every state of an STS M (denoted in Linear Temporal Logic [22] as $M \models G\varphi$). This is essentially invariant verification, meaning that if the property holds then φ is an invariant of the system. If the property does not hold, an execution of the system that leads to $\neg\varphi$ is typically provided as a counterexample.

Alternatively, a *liveness* property is a formula φ which should hold infinitely often in every execution of an STS M (denoted $M \models GF\varphi$), A practical example of this analysis is to verify that a processor is always going to be ready to receive a new command. In liveness verification, a counterexample is an execution where, at some point, φ no longer holds along an infinite execution path. A typical representation of such a trace is a "lasso-shaped" execution, in which the last state of the trace is equal to one of the previous states.

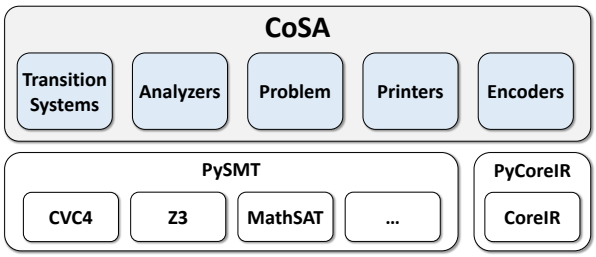

Fig. 2. CoSA Architecture

When analyzing circuit designs, it is often necessary to perform equivalence checking between two systems. The checking is usually based on standard safety verification on a synchronous combination of the systems under analysis, as expressed in Definition 2.

Def. 2 (Synchronous Product of STS)**.** Given two Symbolic Transition Systems $S_1 := \langle V_1, I_1, T_1 \rangle$ and $S_2 := \langle V_2, I_2, T_2 \rangle$ where $V_1 \cap V_2 = \emptyset$, the synchronous product S of S_1 and S_2, namely $S_1 \times S_2$, is defined as $S := \langle V_1 \cup V_2, I_1 \wedge I_2, T_1 \wedge T_2 \rangle$.

B. Verification Engines

CoSA analyzes model-checking problems with Bounded Model-Checking (BMC) [5] techniques, and encodes them using SMT formulas. For each analysis, CoSA provides techniques able to prove or disprove the property. More specifically, for the counterexample generation of safety and liveness verifications the tool relies on BMC [5], while K-Induction [20]/Interpolation [15] and K-Liveness [9] are used to prove safety and liveness properties, respectively.

C. Framework

CoSA [14] is written in Python and its usage is regulated by the modified BSD license. As represented in Figure 2, CoSA builds on top of PySMT [12], which provides a solver-agnostic Python library to interface with SMT solvers. The internal architecture of CoSA is divided into the following parts:

- Transition Systems: defines the internal representation of the model, which is based on a hierarchical set of Transition Systems;
- Analyzers: implements the logic responsible for solving a verification problem. This includes BMC engines and liveness checking;
- Problems: used to define and manage the status of a verification problem;
- Printers: provides support for trace printing (i.e., textual or VCD format), and model translation such as the generation of an SMV file [8];
- Encoders: responsible for encoding different model descriptions into the internal representation. This includes interpreting CoreIR models, and extracting additional information used to optimize the verification process.

Case Study	# State Vars	Total # Bits
A	44	14,771
B	110	27,307
C	1,029 (5 Arrays)	414,847

TABLE I
SIZES OF THE CASE STUDIES - REPORTED FOR COMPOSED SYSTEMS.

For added flexibility, CoSA supports multiple input formats, all of which get translated internally into STS's. In fact, the model under analysis is defined using a list of files whose STS's are synchronously combined (see Def. 2) to produce a single STS. The supported input formats are CoreIR, Explicit-state Transition System (ETS), Symbolic Transition System (STS), and BTOR2 [16]. More information on the input formats is provided in [14]. This approach allows the user to describe complex analyses without modifying the original CoreIR model. For instance, the analysis of programmable hardware often requires a configuration sequence before checking its behavior. This sequence typically includes a reset procedure, for both pos-edge and neg-edge registers, as well as a configuration phase which sequentially loads a bitstream through the configuration port. CoSA facilitates a clear separation between hardware definition, e.g., CoreIR design, and configuration sequence, e.g., ETS. CoSA can generate SMT-LIB files for each of the analyses. Moreover, the ability to translate to SMV format makes it possible to use additional model-checkers such as nuXmv [8].

IV. CASE STUDIES

Below we include several case studies illustrating the utility of CoSA. All of these examples come from the Agile Hardware Project, and cover various stages in the Agile Hardware flow including hardware design, optimization passes, and mapping image processing applications to reconfigurable hardware. All models were translated to CoreIR from (System) Verilog or Halide in order to be analyzed with CoSA. Table I reports the number of variables in the models, including the total size in Bits. All experiments were run on a 2.6GHz Intel Core i7 with 16GB of RAM, and we compared with Yosys, as a reference for open-source word-level model checking.

A. Hardware: Global Controller

The global controller is responsible for configuring the CGRA, managing clock domains, and reading register values for debugging. This module interfaces the JTAG controller, which handles serial communications to and from the chip, with the main CGRA fabric. In this case study, we focused on verifying the global controller in isolation.

The global controller has a register named **state** which records the current state. Certain operations might take multiple cycles to complete, so it uses a counter to keep track of the number of cycles. At the beginning of an operation, the counter is set to the expected delay, and the controller returns to the **ready** state when the counter reaches zero.

Table II lists a selection of properties we attempted to verify using CoSA and the result of each. For the third property, CoSA exposed a bug in the design that could cause the global controller to be stuck in the current state for 2^{32} cycles. The global controller allows the user to configure the operation delay, and because of subtle timing issues, the counter is assigned to the user-specified delay minus one. Thus, if the user asks for a delay of zero, the counter underflows. In this case, the counter would count down starting at the maximum value of a 32-bit unsigned integer and the only way to recover would be to reset the controller. This issue was fixed by special-casing zero-delay requests.

Property	Result
Always return to **ready** state, assuming counter delay < 10	**T**
When not in **ready** state, the counter always decreases	**T**
No underflow in counter	**F**
Read signal is high implies the controller is in the **read** state	**T**
Write signal is high implies the controller is in the **write** state	**F**

TABLE II
PROPERTIES FOR THE GLOBAL CONTROLLER

CoSA also found a counterexample trace in which the write signal could be corrupted. This is accomplished by asking the global controller to switch clock domains, then immediately requesting a write operation. The clock domain switch disables all other operations until the switch is completed, but there is a delay of one clock cycle. Thus, if the write signal is enabled within that delay, it is kept high throughout the clock domain switch, but the controller is not in the **write** state. While interesting, this could not happen in the full system, because it always takes multiple cycles to produce each operation through the JTAG controller.

We also compared the performance of CoSA against the Yosys verification engine, only considering safety properties since Yosys does not natively support liveness checking. We ran the SMT solver CVC4 [1] on the SMT-LIB generated by CoSA and by Yosys (configured with Verific [24] bindings for parsing temporal SystemVerilog Assertions). It takes 4.684s to check all the properties generated by CoSA and 5.395s to check the properties generated by Yosys. The runtimes are comparable, with CoSA running slightly faster.

B. Software: Fold-Constants Pass

CoreIR has an extensible infrastructure for optimization and analysis passes on hardware designs. In the context of the Agile Hardware Project, the design goes through multiple passes before being placed and routed on the fabric. To catch bugs as close to the source as possible, it is desirable to check that these passes produce functionally equivalent designs.

CoSA supports equivalence checking on CoreIR design files and, when necessary, incorporates extra information provided by the CoreIR pass to assist in the proof.

The fold-constants pass is interesting because it can change the number of state variables in the system, which traditionally makes equivalence checking far more difficult. The pass takes any subgraph of the design which is always constant and replaces it with a constant module. The replaced subgraph could be combinational logic operating on constants, or it could be a register which never changes value.

1) Equivalence Checking: Although this pass modifies the design, the functional behavior of the system should not

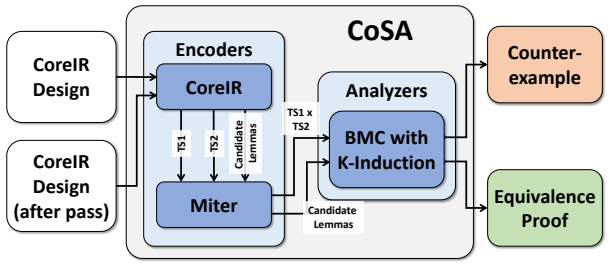

Fig. 3. CoSA automatic proof decomposition strategy for CoreIR passes

change. Given two STS's S_1 and S_2, we need to check that $S_1 \times S_2 \models G(V_{I_1} = V_{I_2}) \implies G(V_{O_1} = V_{O_2})$.

A pure SMT-based K-Induction technique could solve this problem; however, it does not scale well even for moderately sized systems. Alternatively, a verification expert could manually add additional lemmas, but this is time-consuming and procedural. Instead, our approach is to generate lemmas from CoreIR, as depicted in Figure 3. In this specific case, these lemmas express the part of the circuit that has been replaced with a constant by CoreIR, and CoSA adds them as assumptions for the equivalence proof only if they are invariants in the model.

With this proof decomposition, CoSA can check 52 lemmas and prove equivalence between pre-pass and post-pass CoreIR of a CGRA processing element tile configured to do a multiplication in 50 seconds, whereas K-Induction without the additional lemmas does not complete in 2 hours. To compare with Yosys, we produced Verilog from CoreIR for the pre-pass and post-pass designs. These were instantiated together in a top module, similar to the synchronous product encoding in CoSA. K-Induction in Yosys was also unable to prove equivalence in 2 hours.

C. Firmware: Sequential Equivalence of Design and Configured Hardware

We have shown above that CoSA can prove properties of Verilog designs, as well as functional equivalence between CoreIR designs transformed by optimization passes. It is also useful to verify that the configured CGRA faithfully implements the application described by a CoreIR file.

As a simple example, we generated CoreIR that implements a 2x1 convolution, henceforth referred to as the application. This was mapped to CGRA primitives, and then the place and route tools were used to produce a bitstream for a 4x4 CGRA. From the bitstream, we generated an ETS, S_{ETS}, which toggles configuration signals and passes the bitstream to the CGRA inputs. We simulated the CGRA synchronized with S_{ETS} in CoSA to configure the CGRA.

For performance reasons, it helps to simulate without unrolling. In this case, the transition relation was only unrolled one step. The SMT solver was called repeatedly to generate the next step, and the initial state was reassigned each time. A separate check can verify that the configuration phase is deterministic and correct. For space reasons this is not covered here. Once the CGRA was configured, the reset and configuration signals were disabled, and the initial state was assigned to the configured state.

A 2x1 convolution slides a 2-dimensional kernel over an input image. In hardware, this is implemented serially using a linebuffer to delay input pixels. In this case, it was configured for 10x10 input images, and thus the linebuffer has depth 10.

The application implements the linebuffer using a memory with a 5-bit address and a counter. The CGRA implements the linebuffer with nontrivial use of two memories with 9-bit addresses. Convolution depends on the correct linebuffer behavior; thus, these memories could not be soundly blackboxed in a SAT-based model checker. CoSA encodes memories from both the application file and the translated CGRA using the SMT theory of arrays.

We were unable to prove full equivalence because, due to the linebuffers, the equivalence property is not inductive. Unfortunately, we also cannot strengthen the property with array extensionality because of the different use and address widths of memories in the two linebuffer implementations: the memory abstractions are incomparable via standard array equivalence. However, in 2 minutes CoSA was able to prove that if reset is held low, the configuration of the CGRA does not change. Furthermore, CoSA showed in just over 80 minutes that, under basic assumptions of correct usage, the configured CGRA matches the behavior of the CoreIR 2x1 convolution for all executions up to 20 cycles (10 cycles of valid pixel output). For the first ten cycles, inputs are invalid. Thus, CoSA begins sequential equivalence checking once the linebuffer is full and output pixels are valid. Full verification with larger designs is the aim of ongoing work.

V. Related Work

BtorMC [17] is a word-level model checker that relies on the SMT-solver Boolector 3.0 [17] to solve (invariant) model checking problems using bounded techniques [4]. Differently from CoSA, BtorMC is tightly integrated with Boolector, and it does not allow for a simple integration with different solvers.

Yosys [25] is an open source Verilog synthesis suite that provides SMT-based invariant model checking. It interfaces with SMT solvers via SMT-LIB [2] files. Yosys can also rely on ABC [7] for other analyses such as liveness checking. However, ABC engines are based on an encoding into SAT.

VI. Conclusion

In this paper we introduced the CoreIR Symbolic Analyzer (CoSA), an open-source formal verification tool for CoreIR. CoSA provides a broad set of SMT-based formal analyses including model checking and equivalence checking. Moreover, CoSA is able to automatically extract additional information, such as lemmas, from CoreIR to speed up verification tasks.

A series of case studies from the Agile Hardware (AHA) Project at Stanford University [18] were described in order to show that CoSA is capable of handling real hardware verification problems.

For future work, we intend to extend the functionality of CoSA to include full support of Linear Temporal Logic (LTL) and additional input formats such as SMV.

REFERENCES

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. Cvc4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, jul 2011. Snowbird, Utah.

[2] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[3] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, et al. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[5] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.

[6] A. Biere, K. Heljanko, and S. Wieringa. Aiger 1.9 and beyond. *Available at fmv. jku. at/hwmcc11/beyond1. pdf*, 2011.

[7] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[8] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuxmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.

[9] K. Claessen and N. Sörensson. A liveness checking algorithm that counts. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 52–59. IEEE, 2012.

[10] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *International Conference on Computer Aided Verification*, pages 419–422. Springer, 1996.

[11] R. Daly. CoreIR: A simple LLVM-style hardware compiler. https://github.com/rdaly525/coreir, 2017.

[12] M. Gario and A. Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*, pages 373–384, 2015.

[13] D. Huff. Verilog to CoreIR translator. https://github.com/dillonhuff/VerilogToCoreIR, 2018.

[14] C. Mattarei. CoSA: CoreIR Symbolic Analyzer. https://github.com/cristian-mattarei/CoSA, 2018.

[15] K. L. McMillan. Interpolation and sat-based model checking. In *International Conference on Computer Aided Verification*, pages 1–13. Springer, 2003.

[16] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. Btor2 , BtorMC and Boolector 3.0. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

[17] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. BTOR2, BtorMC and Boolector 3.0. In *Computer Aided Verification - 30th International Conference, CAV 2018, Oxford, UK, July 14-17*, Lecture Notes in Computer Science. Springer, 2018.

[18] J. Parkhurst, M. Horowitz, P. Hanrahan, and C. Barrett. AHA Agile Hardware Project. https://aha.stanford.edu/, 2018.

[19] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[20] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.

[21] S. University. Magma: a Hardware Design Language Embedded in Python. https://github.com/phanrahan/magma, 2017.

[22] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for concurrency*, pages 238–266. Springer, 1996.

[23] A. Vasilyev, N. Bhagdikar, A. Pedram, S. Richardson, S. Kvatinsky, and M. Horowitz. Evaluating programmable architectures for imaging and vision applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[24] Verific Design Automation. Verific. http://www.verific.com/.

[25] C. Wolf, J. Glaser, and J. Kepler. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.