Pono: A Flexible and Extensible SMT-Based Model Checker

Makai Mann[1(✉)], Ahmed Irfan[1], Florian Lonsing[1], Yahan Yang[1,3], Hongce Zhang[2], Kristopher Brown[1], Aarti Gupta[2], and Clark Barrett[1]

[1] Stanford University, Stanford, USA
{makaim,irfan,lonsing,barrett}@cs.stanford.edu, ksb@stanford.edu
[2] Princeton University, Princeton, USA
hongcez@princeton.edu, aartig@cs.princeton.edu
[3] University of Pennsylvania, Philadelphia, USA
yangy96@seas.upenn.edu

Abstract. Symbolic model checking is an important tool for finding bugs (or proving the absence of bugs) in modern system designs. Because of this, improving the ease of use, scalability, and performance of model checking tools and algorithms continues to be an important research direction. In service of this goal, we present Pono, an open-source SMT-based model checker. Pono is designed to be both a research platform for developing and improving model checking algorithms, as well as a performance-competitive tool that can be used for academic and industry verification applications. In addition to performance, Pono prioritizes transparency (developed as an open-source project on GitHub), flexibility (Pono can be adapted to a variety of tasks by exploiting its general SMT-based interface), and extensibility (it is easy to add new algorithms and new back-end solvers). In this paper, we describe the design of the tool with a focus on the flexible and extensible architecture, cover its current capabilities, and demonstrate that Pono is competitive with state-of-the-art tools.

1 Introduction

Model checking [39,61] is an influential verification capability in modern system design. Its greatest success has been with finite-state systems, where propositional methods such as binary decision diagrams (BDDs) [28] and Boolean satisfiability (SAT) solvers [69] are used as verification engines. At the same time, significant efforts have been made to lift model checking techniques from finite-state to infinite-state systems [24,30,31,35,46,63]. This requires more expressive verification engines, such as solvers for satisfiability modulo theories (SMT) [19]. Proponents of SMT-based techniques argue that such techniques can also benefit

"Pono" is the Hawaiian word for proper, correct, or goodness. Our goal is that Pono can be a useful tool for people to verify the correctness of systems.

A. Silva and K. R. M. Leino (Eds.): CAV 2021, LNCS 12760, pp. 461–474, 2021.
https://doi.org/10.1007/978-3-030-81688-9_22

finite-state systems, due to their ability to leverage word-level reasoning. Indeed, a word-level model checker won the most recent hardware model checking competition [22], giving credence to this claim. Despite these successes, there remain many directions for exploration in model checking. In this paper, we present `Pono`, an SMT-based model checking tool, with the goal of providing an open research platform for advancing these efforts.

`Pono` is designed with three use cases in mind: 1) *push-button verification*; 2) *expert verification*; and 3) *model checker development*. For 1, `Pono` provides competitive implementations of standard model checking algorithms. For 2, it exposes a flexible API, affording expert users fine-grained control over the tool. This can be useful in traditional model checking tasks (e.g., manually guiding the tool to an invariant, or adjusting the encoding for better performance), but it also enables the tool to be easily adapted for other tasks. In addition, `Pono` is designed using a completely generic SMT solver interface, making it trivial to experiment with different back-end solvers. For 3, `Pono` is open-source [7] and designed to be easily modifiable and extensible with a simple, modular, and hierarchical architecture. Taken together, these features make it relatively easy to do controlled experiments by comparing results obtained using `Pono`, while varying only the SMT solver or the model checking algorithm. `Pono` has already been used in a variety of research projects, both for model checking and other custom applications. It has also been used in two graduate level courses at Stanford University, where students used both the command-line interface and the API. With this promising start, we hope it will have a long and productive existence supporting research, education, and industry.

2 Design

`Pono` is designed around the manipulation and analysis of transition systems. A symbolic transition system is a tuple $\langle X, I, T \rangle$, where X is a set of (sorted) uninterpreted constants referred to as the current-state variables of the system and coupled with corresponding next-state variables X'; $I(X)$ is a formula constraining the initial states of the system; and $T(X, X')$ is a formula expressing the transition relation, which encodes the dynamics of the system. The transition system representation provides a clean and general interface, allowing `Pono` to target both hardware and software model checking. `Pono` is designed to fully leverage the expressivity and reasoning power of modern SMT solving. Its formulas use the language and semantics of the SMT-LIB standard [17], and its model checking algorithms use an SMT solving oracle. To streamline the interaction with SMT solvers, `Pono` uses `Smt-Switch` [59], an open-source C++ API for SMT solving. `Smt-Switch` provides a convenient, efficient, and generic interface for SMT solving. `Smt-Switch` supports a variety of SMT solver back-ends and can switch between them easily.

The diagram in Fig. 1 displays the overall architecture of `Pono`. The blocks with a dashed outline are globally available and used throughout the codebase. The `Pono` API provides access to all of the components shown, supporting the design goal of giving expert users control and flexibility.

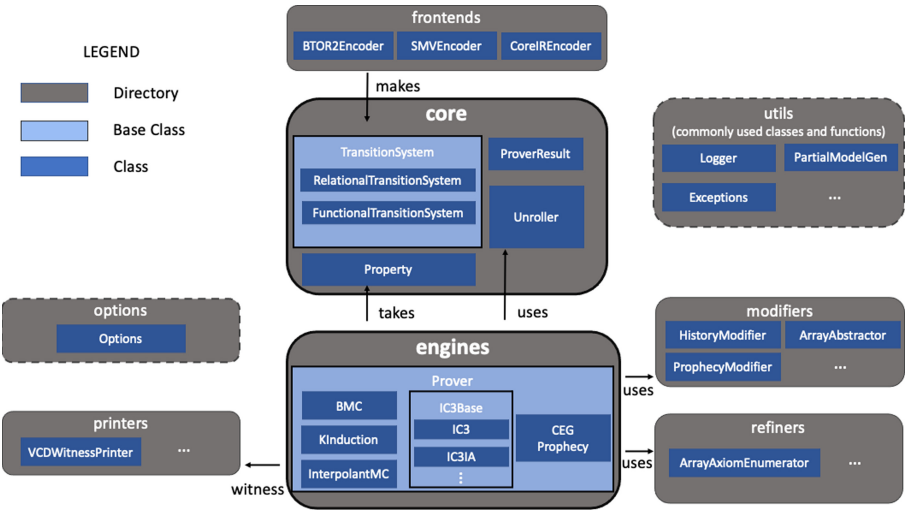

Fig. 1. Architecture diagram

Core. The `TransitionSystem` class in `Pono` represents symbolic transition systems as structured `Smt-Switch` terms. Key data structures include the following: i) `inputvars`: a vector of `Smt-Switch` symbolic constants representing primary inputs to the system (i.e., they are part of X, but their primed versions are not used and cannot appear in T); ii) `statevars`: a vector of `Smt-Switch` symbolic constants corresponding to the non-input state variables (the remaining variables in X); iii) `next_map`: a map from current (X) to next-state (X') variables; iv) `init`: an `Smt-Switch` formula representing $I(X)$; and v) `trans`: an `Smt-Switch` formula representing $T(X, X')$.

There are two kinds of transition systems: `RelationalTransitionSystem` and `FunctionalTransitionSystem`. The former has no restrictions on the form of the transition relation, while the latter is restricted to only functional updates: an equality (update assignment) with a next-state variable on the left and a function of current-state and input variables on the right. Some model checking algorithms take advantage of this structure [46,47]. Built-in checks ensure compliance with the restrictions.

A `Property` is an `Smt-Switch` formula representing a property to check for invariance.[1] A `ProverResult` is an `enum` which can be one of the following: i) `UNKNOWN` (result could not be determined, including incompleteness due to checking only up to some bound); ii) `FALSE` (the property does not hold); iii) `TRUE` (the property holds); and iv) `ERROR` (there was an internal error). The `Unroller` is a class for producing unrolled transition systems, i.e., encoding a finite-length symbolic execution by introducing fresh variables for each timestep.

[1] `Pono` currently supports invariant checking. Support for temporal properties is left to future work.

Engines. Model checking algorithms are implemented as subclasses of the abstract class `Prover` and stored in the `engines` directory. We cover the current suite of engines in more detail in Sect. 3.

Frontends. Although users can manually build transition systems through the API, it is also convenient to generate transition systems from structured input formats. `Pono` includes the following frontends: i) `BTOR2Encoder`: uses the open-source `btor2tools` [2] library to read the `BTOR2` [66] format for hardware model checking; ii) `SMVEncoder`: supports a subset of `nuXmv`'s [30] SMT-based theory extension of `SMV` [61], which added support for infinite-state systems; iii) `CoreIREncoder`: encodes the `CoreIR` [11] circuit intermediate representation. Note that Verilog [10] can be supported by using a translator from Verilog to either `BTOR2` or SMV. Examples of translators include Yosys [72] and Verilog2SMV [53], both of which are open-source.

Printers. `Pono` prints witness traces when a property does not hold. The supported formats are the `BTOR2` witness format and the `VCD` standard format used by EDA tools [10]. For theories such as arithmetic that are not supported by these formats, `Pono` implements simple extensions, ensuring that all variable assignments are included in witness traces.

Modifiers and Refiners. `Pono` includes functions that perform various transformations on transition systems, including: adding an auxiliary variable [14]; building an implicit predicate abstraction [70]; and computing a static cone-of-influence reduction for a functional transition system under a given property. It also includes functions for refining an abstract transition system.

Utils and Options. `utils` contains a collection of general-purpose classes and functions for manipulating and analyzing `Smt-Switch` terms and transition systems. `options` contains a single class, `PonoOptions`, for managing command-line options.

API. `Pono`'s native API is in C++. In addition, `Pono` has Python bindings that interact with the `Smt-Switch` Python bindings, both written in *Cython* [20]. These bindings behave very similarly to "pure" Python objects, allowing introspection and *pythonic* use of the API.

We follow best practices for modern C++ development and code quality maintenance, including issue tracking, code reviews, and continuous integration (via *GitHub Actions*). The build infrastructure is written in CMake [3] and is configurable. The `Pono` repository also provides helper scripts for installing its dependencies. We support `GoogleTest` [5] for unit testing and `gperftools` [12] for code profiling. Tests can be parameterized by both the SMT solver and the algorithm or type of transition system. We utilize *PyTest* [9] to manage and parameterize unit tests for the python bindings.

3 Capabilities

In this section, we highlight some key capabilities of `Pono`. The design makes use of abstract interfaces and inheritance to make it easy to add or extend

functionality. Base class implementations of core functionality are provided but are kept simple to prioritize readability and transparency. And, of course, they can be overridden using inheritance and virtual functions.

We start by describing the interface and engines provided for push-button verification. Next, we take a closer look at two ways that the basic architecture can be extended. We then show how to use `Pono` to reason about a transition system using algebraic datatypes, demonstrating the expressive power provided by the SMT back-end.

Main Engines. All model checking algorithms in `Pono` are derived classes of the abstract base class `Prover`. The base class defines a simple public interface through a set of virtual functions:

- `initialize` initializes any objects and data structures the prover needs.
- `check_until` takes a non-negative integer parameter, k (the effort level), and calls the prover engine (the meaning of k is algorithm-dependent: in BMC [21] and k-induction [68], k is the unrolling length and in IC3-style [25] algorithms, it is the number of *frames*). The interface allows `check_until` to be called repeatedly with increasing values of k. An incremental algorithm can take advantage of this to reuse proof effort from previous calls. Engines that produce full proofs can do so as long as they do it within the provided effort level.
- `prove` attempts to prove a property without any limit on the bound.
- `witness` is called after a failed call to `prove` or `check_until`. It provides variable assignments for each step in a counterexample trace.
- `invar` is called after a successful full proof; it returns an inductive invariant that implies the property. The invariant is an `Smt-Switch Term` over current-state variables. Not all algorithms support this functionality.

`Pono` has several engines, all of which have been lifted to the SMT-level. We now list the main engines and include the corresponding lines of code (LoC) in the primary source file (the LoC includes all comments and license headers): 1. Bounded Model Checking [21] (88 LoC); 2. K-Induction [68] (161 LoC); 3. Interpolant-based Model Checking [62] (230 LoC); 4. IC3-style algorithms [25] (see below for LoC). The engines leverage the reusable infrastructure described in Sect. 2 (e.g., the `Unroller` for the unrolling based techniques).

IC3 Variants. IC3 is widely recognized as one of the best-performing algorithms for SAT-based model checking [43]. Liftings to SMT are an area of active research and have produced several variations with promising results [23,24,34,35,47,51,54,55,71]. To support this active research direction, `Pono` includes a special IC3 base class `IC3Base`, which implements a framework common to all variations of the algorithm.[2] The framework has several parameters that can be provided by specific instances of the algorithm: `IC3Formula` is a configurable data structure used to represent formulas constraining IC3 frames; `inductive_generalization` is the method used for inductive generalization; `predecessor_generalization`

[2] For details on how the IC3 algorithm works, we refer the reader to [25,43].

is the method used for predecessor generalization; and `abstract` and `refine` are methods that can be implemented for abstraction-refinement approaches to IC3 [35,47]. The implementation of `IC3Base` is 1086 lines of code. Current instantiations of `IC3Base` implemented in `Pono` include: i) IC3: a standard Boolean IC3 implementation [25,43] (152 LoC); ii) IC3Bits: a simple extension of IC3 to bit-vectors, which learns clauses over the individual bits (113 LoC); iii) Model-based IC3: a naive implementation of IC3 lifted to SMT, which learns clauses of equalities between variables and model values (397 LoC); iv) IC3IA: IC3 via Implicit Predicate Abstraction [35] (456 LoC); v) IC3SA: a basic implementation of IC3 with Syntax-Guided Abstraction for hardware verification [47] (984 LoC); vi) SyGuS-PDR: a syntax-guided synthesis approach for inductive generalization targeting hardware designs [73] (1047 LoC).

Counterexample-Guided Abstraction Refinement (CEGAR). CEGAR [57] is a popular framework for iteratively solving difficult model checking problems. It is typically parameterized by the underlying model checking algorithm, which operates on an abstract system that is iteratively refined as needed. `Pono` provides a generic `CEGAR` base class, parameterized by a model checking engine through a template argument. We describe two example uses of the CEGAR infrastructure implemented in `Pono`.

Operator Abstraction. This simple CEGAR algorithm uses uninterpreted functions (UF) to abstract potentially expensive theory operators (e.g. multiplication). The implementation is parameterized by the set of operators to replace with UFs. The refinement step analyzes a counterexample trace by restoring the concrete theory operator semantics. If the trace is found to be spurious, constraints are added to enforce the real semantics for the abstracted operators (e.g., equalities between certain abstract UFs and their theory operator counterparts), thus ruling out the spurious counterexample.

Counterexample-Guided Prophecy. This CEGAR approach replaces array variables with initially memoryless variables of uninterpreted sort and replaces the `select` and `store` array operators with UFs [58]. Due to the array theory semantics, it is not always possible to remove spurious counterexamples with quantifier-free refinement axioms over existing variables. However, instead of using potentially expensive quantifiers, the algorithm adds auxiliary variables (history and prophecy variables) [14], which can rule out spurious counterexamples of a given finite length. This approach has the effect of removing the need for array solving and can sometimes prove properties using prophecy variables that would otherwise require a universally quantified invariant.

Case Study with Algebraic Datatypes. To illustrate the flexibility of `Pono`'s SMT-based formalism, we next describe a case study with generalized algebraic theories (GATs) [29]. GATs are a rich formalism which can be used for high-level specifications of software or mathematical constructs. While the equality of two terms in a GAT is undecidable, one can ask the bounded question: "Does there exist a path of up to n rewrites to take a source term to a target term?"

To model this question, we use algebraic datatypes to represent dependently-typed abstract syntax trees (ASTs), paths through an AST (e.g., the 2nd argument of the 3rd argument of a term's 1st argument), and rewrite rules (e.g., $succ(n+1) = succ(m+1) \equiv succ(n) = succ(m)$). `Smt-Switch` supports algebraic datatypes through the CVC4 [18] back-end. A rewrite function is encoded as a transition relation. The decision of which rule to apply and at which subpath to apply it is controlled by input variables, and a state variable represents the current AST term (initially set to the source term). We check the property that the target term is not reachable from the source term. Consequently, any discovered counterexample is a valid rewrite sequence, serving as a proof of an equality that holds in the theory.

The workflow accepts a GAT input, produces an SMT encoding optimized for that particular theory, and then parses user-provided source and target terms into this theory before running bounded model checking. We used `Pono` to successfully find equalities in the theories of Boolean algebras, preorders, monoids, categories, and read-over-write arrays. This case study demonstrates `Pono`'s ability to model and model check unconventional systems.

4 Related Work

Existing academic model checkers span a wide range of supported theories, modeling capabilities, and implemented algorithms. An important early model checker was `SMV` [61], which pioneered *symbolic* model checking of temporal logic properties [67] through BDDs [28]. `NuSMV` [32] and `NuSMV2` [33] refined and extended the tool, followed by `nuXmv` [30] – a closed-source tool which added support for various SMT-based verification techniques using the SMT solver MathSAT5 [36]. `Spin` [52] is a well-known explicit-state model checker with extensive support for partial order reduction and other optimizations.

Several model checkers specifically target hardware verification. `ABC` [26] is a well-established, state-of-the-art bit-level hardware model checker based on SAT solving. `CoSA` [60] is an open-source model checker implemented in Python using the Python solver-agnostic SMT solving library, PySMT [45]. Although `CoSA` also relies on a generic API similar to `Smt-Switch`, the Python implementation introduces significant overhead, limiting its ability to include efficient procedures that must be implemented outside of the underlying SMT solver (e.g., CEGAR loops and some IC3 variants). `AVR` [48] is a state-of-the-art SMT-based hardware model checker supporting several standard model checking algorithms. It also implements a novel technique: IC3 via syntax-guided abstraction [47]. Importantly, `AVR` won the hardware model checking competition in 2020 [22], outperforming the previous state-of-the-art SAT-based model checker, `ABC`. `AVR` is currently closed-source, making it unsuitable for several of the use-cases targeted by our work, but a binary is available on GitHub [1].

There are several SMT-based model checkers focused on parameterized protocols. `MCMT` [46], the open-source extension `Cubicle` [49], and related systems [15,16] perform backward-reachability analysis over infinite-state arrays.

Other open-source SMT-based model checkers include: i) `ic3ia` [13] – an example implementation of IC3IA built on MathSAT [36]; ii) `Kind2` [31] – a model checker for Lustre programs; iii) `Sally` [42] – a model checker for infinite-state systems that uses the SAL language [65] and MCMT, an extension of the SMT-LIB text format for declaring transition systems; iv) `Spacer` [56] – a Constrained Horn Clauses (CHC) solver built into the open-source Z3 [64] SMT solver, also based on an IC3-style algorithm; and v) `Intrepid` [27] – a model checker focusing primarily on the control engineering domain.

`Pono` is open-source, SMT-based, and implements a variety of model checking algorithms over transition systems. Furthermore, in contrast to the tools which focus on more limited domains, it has support for a wide set of SMT theories including fixed-width bit-vectors, arithmetic, arrays, and algebraic datatypes. To our knowledge all current open-source SMT-based model checkers tie the implementation directly to an existing SMT solver or use PySMT or the SMT-LIB text format to interact with arbitrary solvers. In contrast, `Pono` makes use of the C++ API of `Smt-Switch` to efficiently manipulate SMT terms and solvers in memory without a need for a textual interface. This allows `Pono` to provide both flexibility and performance. Finally, like the new model checker `Intrepid`, `Pono` provides an extensive API, which can be adapted and extended as needed. However, the focus is broader than `Intrepid` in terms of application domains.

5 Evaluation

In this section, we evaluate `Pono`[3] against current state-of-the-art model checkers across several domains. Our evaluation is not intended to be exhaustive. Rather, we highlight the breadth of `Pono` by selecting four sets of benchmarks in three diverse categories and a few reasonable competitors for each. The benchmarks are drawn from the following theories: i) unbounded quantifier-free arrays indexed by integers; ii) quantifier-free linear arithmetic over reals and integers; and iii) hardware verification over quantifier-free bit-vectors and (finite, bit-vector indexed) arrays. We ran all experiments on a 3.5 GHz Intel Xeon E5-2637 v4 CPU with a timeout of 1 h and a memory limit of 16 Gb. For all results, we also include the average runtime of solved instances in seconds. For portfolio solving, we ran each configuration in its own process with the full time and memory resources. In the first two categories, `Pono` used MathSAT5 [36] as the underlying SMT solver and interpolant [37,40,62] producer. For the hardware benchmarks, it used MathSAT5, Boolector [66], or both, depending on the configuration.

Arrays. We evaluate `Pono` on the integer-indexed array benchmark set of [44]. These are Constrained Horn Clauses (CHC) benchmarks inspired by software verification problems. Although there are no quantifiers in the benchmarks themselves, most cannot be proved safe without strengthening the property with quantified invariants. We compare against: i) `freqhorn` [44], a state-of-the-art CHC solver for this type of problem; ii) `prophic3` [8], a recent method that

[3] GitHub commit c175a302857ff00229a0919d5cc8fc3f78d04a26.

	Pono	prophic3	prophic3-SA	freqhorn	nuXmv
solved	**71** (16s)	**71** (20s)	66 (31s)	69 (6s)	4 (51s)

Fig. 2. Results on Freqhorn Array benchmarks (81 total), all expected to be safe.

result	SystemC (43 total)		Lustre (951 total)		
	Pono	nuXmv	Pono	nuXmv	kind2
safe	18 (673s)	**21** (571s)	**521** (10s)	516 (8s)	506 (2s)
unsafe	14 (325s)	**15** (479s)	412 (5s)	412 (1s)	409 (0.2s)
total	32 (521s)	**36** (533s)	**933** (8s)	928 (5s)	915 (1s)

Fig. 3. Results on arithmetic benchmarks.

outperforms freqhorn [58]; and iii) nuXmv, which does not support quantified invariants, to illustrate that most of these benchmarks do require them; freqhorn takes the CHC format natively, and we used scripts from the ic3ia and nuXmv distributions to translate the CHC input to SMV and the Verification Modulo Theories (VMT) format [38] – an annotated SMT-LIB file representing a transition system – for the other tools. We ran Pono with Counterexample-Guided Prophecy using IC3IA as the underlying model checking technique. We ran prophic3 with both of the option sets used in their paper, and we ran the default configuration of freqhorn. Our results are shown in Fig. 2. We observe that Pono solves the same number of benchmarks as the reference implementation prophic3 and is a bit faster.

Arithmetic. We next evaluate Pono on two sets of arithmetic benchmarks, both from the nuXmv distribution's example directory. The first uses linear real arithmetic, and the second uses linear integer arithmetic. Figure 3 displays the results on both benchmark sets.

Linear Real Arithmetic. We chose the systemc QF_LRA example benchmarks, because this is the largest set of linear real arithmetic benchmarks in the subset of SMV supported by Pono.[4] We ran both nuXmv and Pono with BMC and IC3IA in a portfolio. For both model checkers, BMC did not contribute any unique solves. We observe that Pono is quite competitive with nuXmv on nuXmv's own benchmarks.

Linear Integer Arithmetic. We also evaluate Pono on a set of Lustre benchmarks which use quantifier-free linear integer arithmetic. We obtained the Lustre benchmarks from the Kind [50] website [6] and the SMV translation of the benchmarks from the distribution of nuXmv. We compare against both nuXmv and Kind2 [31], the latest version of Kind. We ran all tools with a portfolio of techniques. For Pono and nuXmv we ran BMC and IC3IA. For Kind2 we ran two configurations suggested by the authors: the default configuration with Z3 [64] and the default configuration, but with Yices2 [41] as the main SMT solver. Since the default

[4] Pono does not yet support enumeration types.

result	BV (324 total)				BV + Array (315 total)		
	Pono	AVR	CoSA2	sygus-apdr	Pono	AVR	CoSA2
safe	183 (283s)	**215** (115s)	98 (283s)	115 (545s)	252 (224s)	**274** (63s)	209 (299s)
unsafe	47 (314s)	47 (220s)	41 (232s)	15 (279s)	19 (208s)	19 (352s)	19 (204s)
total	230 (289s)	**262** (134s)	139 (268s)	130 (514s)	271 (223s)	**293** (82s)	228 (291s)

Fig. 4. Results on HWMCC2020 benchmarks.

configurations of Kind2 run 8 techniques in parallel, we gave each configuration 8 cores. Additionally, we ran Kind2's BMC and IC3 implementations using MathSAT5 as the SMT solver, because this is closest to the other model checkers' configurations. The default with Z3 was the best configuration of Kind2. We observe that Pono solves the most benchmarks overall. Once again, BMC contributed no unique solves for any model checker.

Hardware Verification. Finally, we evaluate Pono on the 2020 Hardware Model Checking Competition (HWMCC) benchmarks. The benchmarks are split into bitvector-only and bitvector plus array categories. We evaluate against AVR [1,48] and CoSA2 [4] (a previous name and version of Pono), the winners of HWMCC 2020 and HWMCC 2019, respectively. We also compare against sygus-apdr (the reference implementation of SyGuS-PDR [73]) on the bitvector benchmarks (as sygus-apdr targets bitvectors). We ran all 16 configurations of AVR from their HWMCC 2020 entry: several configurations of BMC and k-induction, and 11 configurations of IC3SA. We ran the 4 configurations of CoSA2 from the HWMCC 2019 entry: two BMC configurations, k-induction, and interpolant-based model checking. We ran sygus-apdr with 4 different parameters controlling the grammar for lemmas. For the bitvector-only benchmarks, we ran Pono with 10 configurations: 3 configurations of IC3IA, 2 configurations of IC3SA, 2 configurations of SyGuS-PDR, IC3Bits, k-induction, and BMC. For the array benchmarks, we ran 5 configurations: 3 configurations of IC3IA (one with Counterexample-Guided Prophecy), k-induction, and BMC. We show our results on the HWMCC 2020 benchmarks in Fig. 4. AVR wins in both categories, although Pono is fairly competitive, outperforming the other tools.

These results show that Pono is well on its way to being both widely applicable and performance-competitive. The arithmetic experiments demonstrate the capabilities of its IC3IA engine, but other engines have some room for improvement. In particular, both IC3SA and SyGuS-PDR were recently added to Pono, and its implementation of these algorithms still lags the corresponding implementations in AVR and sygus-apdr, respectively. There are also some features that are known to help performance and are not yet implemented in Pono. For example, the best configurations of AVR use UF data abstraction. This differs from our UF operator abstraction in that it replaces all abstracted data with uninterpreted sorts and learns targeted data refinement axioms.

6 Conclusion

We have presented Pono: a new open-source, SMT-based, and solver-agnostic model checker. We described its capabilities, design, and the emphasis on flexibility and extensibility in addition to performance. We demonstrated empirically that the suite of model checking algorithms is competitive with state-of-the-art tools. Pono has already been used in several research projects and two graduate-level classes. With this promising start, we believe that Pono is poised to have an enduring and beneficial impact on research, education, and model checking applications. Future work includes adding support for temporal properties [67] and improving and adding to Pono's engines, in particular the IC3 variants.

Acknowledgements. This work was partially supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This work was also supported by the Defense Advanced Research Projects Agency, grants FA8650-18-1-7818 and FA8650-18-2-7854. We thank these sponsors and our industry collaborators for their support.

References

1. AVR distribution. https://github.com/aman-goel/avr
2. btor2tools. https://github.com/Boolector/btor2tools
3. CMake. https://cmake.org
4. cosa2. https://github.com/upscale-project/cosa2
5. GoogleTest. https://github.com/google/googletest
6. Kind site. http://clc.cs.uiowa.edu/Kind/index.php?page=experimental-results
7. Pono. https://github.com/upscale-project/pono
8. ProphIC3 (commit: 497e2fbfb813bcf0a2c3bcb5b55ad47b2a678611). https://github.com/makaimann/prophic3
9. pytest 5.4.2. https://github.com/pytest-dev/pytest
10. IEEE Std 1364–2005, pp. 1–590 (2006)
11. CoreIR (2017). https://github.com/rdaly525/coreir
12. Google Perftools (2017). https://github.com/gperftools/gperftools
13. ic3ia. https://es-static.fbk.eu/people/griggio/ic3ia/index.html. Accessed 2020
14. Abadi, M., Lamport, L.: The existence of refinement mappings. In: Proceedings of LICS, pp. 165–175, July 1988
15. Alberti, F., Bruttomesso, R., et al.: SAFARI: SMT-based abstraction for arrays with interpolants. In: Proceedings of CAV, pp. 679–685 (2012)
16. Alberti, F., Ghilardi, S., Sharygina, N.: Booster: an acceleration-based verification framework for array programs. In: Proceedings of ATVA, pp. 18–23 (2014)
17. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). www.smt-lib.org
18. Barrett, C.W., et al.: CVC4. In: Proceedings of CAV, pp. 171–177 (2011)

19. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, pp. 825–885 (2009)
20. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K.: Cython: the best of both worlds. Comput. Sci. Eng. **2**, 31–39 (2011)
21. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings of TACAS, pp. 193–207 (1999)
22. Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition (2020). http://fmv.jku.at/hwmcc20/
23. Birgmeier, J., Bradley, A., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Proceedings of CAV, pp. 831–848 (2014)
24. Bjørner, N., Gurfinkel, A.: Property directed polyhedral abstraction. In: Proceedings of VMCAI, pp. 263–281 (2015)
25. Bradley, A.: SAT-based model checking without unrolling. In: Proceedings of VMCAI, pp. 70–87 (2011)
26. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proceedings of CAV, pp. 24–40 (2010)
27. Bruttomesso, R.: Intrepid: An SMT-based model checker for control engineering and industrial automation. In: SMT Workshop, August 2019
28. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **8**, 677–691 (1986)
29. Cartmell, J.: Generalised algebraic theories and contextual categories. Ann. Pure Appl. Logic 209–243 (1986)
30. Cavada, R., Cimatti, A., et al.: The nuXmv symbolic model checker. In: Proceedings of CAV, pp. 334–342 (2014)
31. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Proceedings of CAV, pp. 510–517 (2016)
32. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: A new symbolic model verifier. In: Proceedings of CAV, pp. 495–499 (1999)
33. Cimatti, A., Clarke, E.M., et al.: NuSMV 2: an opensource tool for symbolic model checking. In: Proceedings of CAV, pp. 359–364 (2002)
34. Cimatti, A., Griggio, A., Irfan, A., et al.: Incremental linearization for satisfiability and verification modulo nonlinear arithmetic and transcendental functions. ACM Trans. Comput. Log. 19:1–19:52 (2018)
35. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with IC3 and predicate abstraction. FMSD **3**, 190–218 (2016)
36. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) Proceedings of TACAS (2013)
37. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Log. (1), 7:1–7:54 (2010)
38. Cimatti, A., et al.: Verification Modulo Theories (2011). http://www.vmt-lib.org
39. Clarke, E., Henzinger, T., et al.: Handbook of Model Checking (2018)
40. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. (3), 250–268 (1957)
41. Dutertre, B.: Yices 2.2. In: Proceedings of CAV, pp. 737–744 (2014)
42. Dutertre, B., Jovanovic, D., Navas, J.A.: Verification of fault-tolerant protocols with Sally. In: Proceedings of NFM, pp. 113–120 (2018)
43. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proceedings of FMCAD, pp. 125–134 (2011)
44. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Proceedings of CAV, pp. 259–277 (2019)

45. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proceedings of SMT Workshop, pp. 373–384 (2015)
46. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Automated Reasoning, pp. 22–29 (2010)
47. Goel, A., Sakallah, K.A.: Model checking of Verilog RTL using IC3 with syntax-guided abstraction. In: Proceedings of NFM, pp. 166–185 (2019)
48. Goel, A., Sakallah, K.A.: AVR: abstractly verifying reachability. In: Proceedings of TACAS, pp. 413–422 (2020)
49. Goel, A., Krstic, S., Leslie, R., Tuttle, M.R.: SMT-based system verification with DVF. In: Proceedings of SMT Workshop, pp. 32–43 (2012)
50. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Proceedings of FMCAD, pp. 1–9 (2008)
51. Ho, Y., Mishchenko, A., Brayton, R.K.: Property directed reachability with word-level abstraction. In: Proceedings of FMCAD, pp. 132–139 (2017)
52. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual (2004)
53. Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2SMV: a tool for word-level verification. In: Proceedings of DATE, pp. 1156–1159 (2016)
54. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Proceedings of FMCAD, pp. 85–92 (2016)
55. K., H.G.V., Fedyukovich, G., Gurfinkel, A.: Word level property directed reachability. In: Proceedings of ICCAD, pp. 107:1–107:9 (2020)
56. Komuravelli, A., Gurfinkel, A., et al.: Automatic abstraction in SMT-based unbounded software model checking. In: Proceedings of CAV, pp. 846–862 (2013)
57. Kroening, D., Groce, A., Clarke, E.M.: Counterexample guided abstraction refinement via program execution. In: Proceedings of ICFEM, pp. 224–238 (2004)
58. Mann, M., Irfan, A., et al.: Counterexample-guided prophecy for model checking modulo the theory of arrays. In: Proceedings of TACAS, pp. 113–132 (2021)
59. Mann, M., Wilson, A., et al.: SMT-Switch: a Solver-agnostic C++ API for SMT Solving. In: Proceedings of SAT (2021)
60. Mattarei, C., Mann, M., Barrett, C., et al.: CoSA: Integrated verification for agile hardware design. In: Proceedings of FMCAD, pp. 1–5 (2018)
61. McMillan, K.: Symbolic model checking - an approach to the state explosion problem. Ph.D. thesis, Carnegie Mellon University (1992)
62. McMillan, K.L.: Interpolants and symbolic model checking. In: Proceedings of VMCAI, pp. 89–90 (2007)
63. McMillan, K.L., Padon, O.: Ivy: a multi-modal verification tool for distributed algorithms. In: Proceedings of CAV, pp. 190–202 (2020)
64. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of TACAS, pp. 337–340 (2008)
65. de Moura, L., et al.: Sal 2. In: Proceedings of CAV, pp. 496–500 (2004)
66. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC and Boolector 3.0. In: Proceedings of CAV, pp. 587–595 (2018)
67. Pnueli, A.: The temporal logic of programs. In: Proceedings of FOCS, pp. 46–57 (1977)
68. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proceedings of FMCAD, pp. 108–125 (2000)
69. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153 (2009)
70. Tonetta, S.: Abstract model checking without computing the abstraction. In: Proceedings of FM, pp. 89–105 (2009)

71. Welp, T., Kuehlmann, A.: QF BV model checking with property directed reachability. In: Proceedings of DATE, pp. 791–796 (2013)
72. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free Verilog synthesis suite. In: Proceedings of Austrochip Workshop (2013)
73. Zhang, H., Gupta, A., Malik, S.: Syntax-guided synthesis for lemma generation in hardware model checking. In: Proceedings of VMCAI (2021)