


Pono 2.0: A Versatile SMT-Based Model Checker for Safety and Liveness (Long Tool Paper)

Áron Ricardo Perez-Lopez[1(✉)], Po-Chun Chien[1,2], Florian Lonsing[3], Samantha Archer[1], Ahmed Irfan[4], and Clark Barrett[1]

[1] Stanford University, Stanford, USA
{arpl,barrett}@cs.stanford.edu, samanthaarcher@stanford.edu
[2] LMU Munich, Munich, Germany
po-chun.chien@sosy.ifi.lmu.de
[3] Linz, Austria
fml@florianlonsing.com
[4] SRI International, Menlo Park, USA
ahmed.irfan@sri.com

Abstract. We introduce an updated version of the Pono model checker. Pono is a versatile SMT-based model checker that integrates multiple verification algorithms and interfaces with a wide range of SMT solvers through a solver-agnostic back end. It emphasizes usability, offering support for commonly used input formats and providing C++ and Python APIs for programmatic access. The new version 2.0 introduces several important new features, including support for liveness properties, new interpolation-based safety-checking engines, a new VMT-LIB front end, and a number of usability and performance enhancements. An evaluation of the new version demonstrates significant improvements in performance over its previous version and comparable performance to other state-of-the-art model checkers. These results highlight Pono 2.0's effectiveness as a general-purpose and easily extensible verification platform.

Keywords: Model checking · Safety · Liveness · Craig interpolation · IC3 · SMT · CEGAR · Verification witness · Btor2 · VMT-LIB

1 Introduction

Model checking is a widely-used technique for automatically verifying hardware and software systems. Early algorithms for symbolic verification of finite-state systems used Binary Decision Diagrams [26]. Gradually, methods based on Boolean satisfiability (SAT) have become more prominent due to their robustness and scalability [12,22,44,62]. More recently, solvers for satisfiability modulo theories (SMT) have been employed, enabling the representation of finite-state and even infinite-state transition systems with richer structural information. For

A. Sampaio and M. Stoelinga (Eds.): FM 2026, LNCS 16557, pp. 3–26, 2026.
https://doi.org/10.1007/978-3-032-26220-2_1

example, a hardware register can be modeled as a single bit-vector rather than as individual bits, lifting reasoning from the bit level to the word level.

Pono is an open-source SMT-based model checker [58].[1] Its name, "Pono," is Hawaiian for *proper*, *correct*, or *goodness*, reflecting our goal of providing a reliable tool for verifying the correctness of systems. The tool is designed to be flexible and extensible, enabling both experimentation and competitive performance with state-of-the-art model checkers. Pono can model theory-rich transition systems, including bit-vectors, integers, reals, and arrays, and supports popular input formats such as Btor2 [68] and SMV [28,32,61] and, as of version 2.0, VMT-LIB [38], with support for MoXI [70] forthcoming. It is solver-agnostic and interfaces with multiple SMT solvers via Smt-Switch [59].[2] By default, Pono uses Bitwuzla [65] for bit-vector and array solving, MathSAT5 [36] for interpolation queries, and cvc5 [4] for everything else.

Pono supports checking safety and, as of version 2.0, also liveness properties. It boasts a versatile set of algorithms based on bounded model checking [12], k-induction [71], interpolation [62,76,77], IC3/PDR [22,35,49], liveness-to-safety reduction [11,39], and counterexample-guided abstraction refinement [40,52,57].

Since the initial release in 2021 [58], we have made numerous improvements that benefit both the user experience and performance.

1.1 Use Cases

Pono is designed to support three primary use cases: push-button verification, expert verification, and model checking research, goals described in the Pono 1.0 paper [58]. With the release of Pono 2.0, we can now point to many concrete examples and success stories in these three categories.

Push-Button Verification. Several recent verification frameworks, including μArchiFi [74], MoXI-MC-Flow [53], DreamMiner [79], and Btor2-Select [56], as well as industrial tool suites like TabbyCAD[3] integrate Pono as a back-end model checker for fully automated verification tasks. These systems directly invoke Pono's binary executable on model-checking problems without user intervention, demonstrating its robustness and ease of use. Notably, Pono achieved **first place** in the word-level array track of the Hardware Model Checking Competition (HWMCC) 2025 [17], further confirming its competitiveness in push-button verification.

Expert Verification. For advanced users, Pono provides flexible C++ and Python APIs for manipulating and instrumenting transition systems and properties. This enables fine-grained control over the verification process, as demonstrated by works on automated system configuration [75] and translation validation [63] for agile hardware design workflows.

[1] https://github.com/stanford-centaur/pono
[2] https://github.com/stanford-centaur/smt-switch
[3] https://www.yosyshq.com/tabby-cad-datasheet

Model Checking Research. Pono's modular and solver-agnostic architecture makes it a convenient platform for evaluating new verification algorithms and solver configurations. For instance, it has been used to explore different interpolation configurations in Bitwuzla [66]. In the evaluation part (Sect. 6) of this paper, Pono serves as a unified framework for exercising multiple model-checking algorithms. Its extensible design continues to make it an effective foundation for both research and tool development.

1.2 Contributions

To the best of our knowledge, Pono 2.0 is the first tool to support verification of liveness properties for systems represented using the Btor2 format. We also introduce new interpolation-based model-checking engines [76,77] as well as improvements to existing engines, enhancing both flexibility and performance. Usability has also been improved, with a new VMT-LIB [38] front end and a more robust witness-generation component. Pono 2.0 integrates seamlessly with state-of-the-art SMT solvers such as Bitwuzla [65] and cvc5 [4], allowing users to exploit advanced solver capabilities. Finally, we present a comprehensive evaluation on a diverse benchmark set, demonstrating Pono's effectiveness across a variety of verification tasks.

The rest of the paper is organized as follows. In Sect. 2, we provide a more formal description of the model checking problem addressed by Pono. Section 3 provides an overview of Pono's architecture, and Sect. 4 goes into more detail about the features that were added or improved between versions 1.0 and 2.0. We compare Pono with other state-of-the-art model checkers in Sect. 6 and conclude the paper in Sect. 7.

2 Background

Every SMT variable has a *sort* derived from a background theory, e.g., `Int`—an integer, or `(_ BitVec 32)`—a 32-bit vector. These variables can be combined into Boolean-valued *atoms* using predefined theory-specific functions, e.g., the atom `(= (bvmul a b) #b100)` states that the result of performing bit-vector multiplication on `a` and `b` equals 4. Finally, atoms can be connected with Boolean operators like "and" ($\wedge$) and "implies" ($\implies$) to create *formulas*.

The input of Pono is a *transition system* $\langle X, I, T \rangle$ consisting of: (1) X, a set of state variables (2) $I(X)$, the set of initial states, expressed as a formula over the state variables, and (3) $T(X, X')$, the transition relation, expressed as a formula over the state variables and their primed copies, which represent the current and next state, respectively. Pono allows the designation of some variables as *input* variables, which do not appear in the initial state condition or the next-state set X' of the transition relation. This makes them effectively free variables whose value can be chosen arbitrarily at every step.

If the domain of every variable in a transition is finite, e.g., a bit-vector, as defined by the theory of fixed-size bit-vectors, the transition system is finite. If

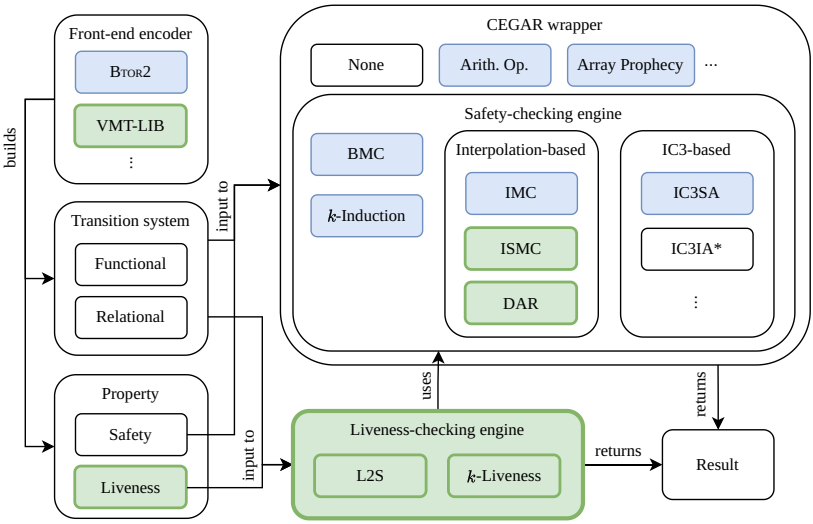


Fig. 1. System diagram of Pono (new and enhanced components in 2.0 are highlighted in green and blue, respectively; *: IC3IA is also interpolation-based)

there are some variables with infinite domain, e.g., integers from the theory of Linear Integer Arithmetic, the transition system will have an infinite number of states. This distinction is important in several model checking algorithms.

The goal of a model-checking procedure is to determine whether a certain temporal property holds for a given transition system. We restrict ourselves to properties expressible in Linear Temporal Logic (LTL), which covers many useful properties. LTL properties can be divided into two classes. *Safety* properties, also called invariant properties, assert that every reachable state of the system satisfies a predicate $P(X)$. In contrast, *liveness* properties [2] assert that some condition holds *eventually* along any path (rather than always).

The Btor2 input format [68] is a word-level extension of the AIGER format [18] and represents safety properties with specially designated `bad` formulas, which are negations of safety properties. In other words, a `bad` property is something that should *never* hold in any reachable state. Liveness properties are expressed using lists of `justice` formulas. As with the `bad` keyword, the `justice` keyword indicates the set of formulas that should *not* hold infinitely often. More precisely, the liveness property does not hold if there exists a path starting from an initial state along which each of the justice formulas holds infinitely often. This is equivalent to saying that the property does hold if there is some path along which eventually at least one of the justice formulas becomes false forever. The space of reachable states can be restricted with `constraint` formulas—only states for which all constraint formulas are true are considered reachable.

```
IMPORT smt_switch, pono
solver = smt_switch.create_bitwuzla_solver(False)
solver.set_opt("incremental", "true")
ts = pono.FunctionalTransitionSystem(solver)
# build the transition system here ...
prop = pono.SafetyProperty(solver, solver.make_term(...))
engine = pono.KInduction(prop, ts, solver)
result = engine.check_until(10)  # unroll at most 10 steps
```

Fig. 2. Example Python API usage

3 Architecture

Figure 1 provides a high-level overview of the system and its core components. The architecture of Pono is described in detail in its previous tool paper [58]. Here, we revisit the major components and highlight the changes in version 2.0. Pono can be either executed directly as a command-line tool or linked into another application as a library. With Pono 2.0, the command-line interface (CLI) can be used without manual compilation, by running the OCI container built from the main branch.[4] This section describes the C++ API that Pono provides for other applications, but all of this functionality is also available via the CLI.

Pono builds on Smt-Switch [59] for constructing SMT formulas and solving queries. Accordingly, most operations described below use Smt-Switch `Term`s or their collections (`TermVec`, `UnorderedTermSet`, etc.) for input arguments and return values.

The core data structure in Pono is the `TransitionSystem`. As the name implies, this represents a transition system, and all other principal operations are carried out on this class. `TransitionSystem`s can be constructed by manually calling `add_statevar()` and `add_inputvar()` or by using a `Frontend`. The original version of Pono provided `Btor2Frontend` and `SMVFrontend` for constructing `TransitionSystem`s from Btor2 [68] and SMV [28,32,61] files, respectively. Pono 2.0 further adds support for the VMT-LIB [38] input format.

The sets of state and input variables in a `TransitionSystem` can be obtained using the `statevars()` and `inputvars()` methods, respectively, while `init()` returns the initial state condition. In functional systems, `assign_next()` can be used to set the values for each variable's next state and `state_updates()` can be used to retrieve them. In a relational style, `trans()` and `set_trans()` can be used to query and set the transition relation, respectively.

Pono supports all background theories provided by the underlying SMT solvers via Smt-Switch. Thus, when using Pono's APIs, users can construct transition systems over arbitrary solver-supported sorts. However, certain engines only support a subset of these theories, and when loading models through formats such as Btor2 or VMT-LIB, the available theories are restricted by the expressiveness of the respective input format.

[4] https://ghcr.io/stanford-centaur/pono:latest

Table 1. Summary of new and improved features in Pono 2.0

Feature	New in version 2.0	Description
Front ends	VMT-LIB, Btor2 `justice`	Sect. 4.1
BMC engine	bound start, bound step	Sect. 4.2
k-Induction engine	bound step	
IMC engine	backward interpolation, frontier-set simplification	
ISMC engine	entirely new	
DAR engine	entirely new	
IC3SA engine	array support	
L2S translation	entirely new	Sect. 4.3
k-liveness translation	entirely new	
Arith. op. abstraction	abs. with free variables, support for more engines	Sect. 4.4
CEG prophecy	finite-index arrays	
Witness generation	support for all engines	Sect. 4.5
SMT backends	configurable options	Sect. 4.7

The workhorses of Pono are the `Provers`. These are the classes that implement the various supported model checking algorithms. The inputs to a `Prover` are a `TransitionSystem` and a `Property`. The `Property` is the temporal property that the `Prover` is asked to prove or disprove. In Pono 1.x, this is always a safety property encoded by an Smt-Switch `Term` representing a formula. Pono 2.0 additionally supports liveness properties and introduces two distinct classes, `SafetyProperty` and `LivenessProperty`. Since many model-checking algorithms are specialized to either safety or liveness properties, Pono 2.0 has `SafetyProver` and `LivenessProver` subclasses that take (and attempt to prove) the corresponding type of `Property`. New and updated `Provers` in Pono 2.0 are described in detail in Sect. 4.

There are also Python bindings for accessing most of the C++ API. An example using the Python bindings is illustrated in Fig. 2. A Bitwuzla solver instance `solver` is created using Smt-Switch. Then, a transition system `ts` and a safety property `prop` are constructed and given to a k-induction engine. Finally, the `check_until()` method is invoked to perform model checking.

4 New Features in Pono 2.0

In this section, we highlight the new features and improvements in Pono 2.0, which are summarized in Table 1.

4.1 Front-End Parsers

In Pono 1.0, the supported front ends are Btor2, SMV, and CoreIR formats, but only for safety properties. Version 2.0 adds support for liveness properties in Btor2 (using the keyword `justice`) and extends safety verification to the VMT-LIB format. VMT-LIB is based on the SMT-LIB [5] format, and supports reasoning over a richer set of theories than does Btor2.

4.2 Safety Model Checking

Pono provides a diverse set of model-checking engines for verifying safety properties, including approaches based on unrolling, IC3/PDR [22,44], and interpolation. In version 2.0, two new interpolation-based engines have been added, interpolation sequence-based model checking and dual-approximated reachability, and many of the other engines have been improved.

Bounded Model Checking (BMC) and k -Induction. BMC [12] is a common verification technique for checking safety properties. It works by unrolling the transition system $T(X, X')$ from the initial state $I(X)$ and encoding the paths that reach bad states $\neg P(X)$ as SAT/SMT queries. A satisfiable result indicates a reachable bad state (a counterexample), while an unsatisfiable result means no bad state is reachable within the unrolled depth. k-induction [71] extends BMC by adding an inductive step which attempts to prove that no bad state is reachable beyond the unrolling depth, provided the property holds for the previous k steps. If the inductive step succeeds, the property holds for all reachable states, but if not, the result is inconclusive, as the property may still hold but just not be k-inductive.

Both BMC and k-induction were supported in Pono 1.0. In version 2.0, we introduce additional options to control the unrolling process. Users can specify the step size for each unrolling, and BMC can optionally increase the unrolling bound exponentially. For BMC, there is now also an option to set the starting point for the unrolling. These enhancements have proven useful for bug finding. Notably, these engines helped Pono find the highest number of satisfiable instances in the word-level array track at HWMCC'25 [17].

Interpolation-Based Model Checking (IMC). IMC [62] is a well-known algorithm for safety verification that combines bounded model checking [12] with Craig interpolation [41]. The algorithm incrementally unrolls the transition system and checks for reachability of a bad state. When the BMC check is unsatisfiable, it iteratively extracts interpolants to overapproximate the set of reachable states. The algorithm continues until a counterexample is found (i.e.,

the BMC check is satisfiable) or the disjunction of computed interpolants forms a fixed point (i.e., an inductive invariant).

IMC was already available in Pono 1.0 [58]. In Pono 2.0, we extend it with two new features. First, we introduce backward derivation of interpolants. That is, instead of calling `get_interpolant(A, B)`, where `A` and `B` are the two partitions of an unsatisfiable BMC formula [62], we compute `not(get_interpolant(B, A))`. This allows the SMT solver to keep the formula `B`, which is usually more complicated than `A` (as `B` involves several copies of the unrolled transition relation T) and remains identical across multiple interpolation queries, on the solver stack, facilitating incremental solving. Prior work [8] also shows that this approach often produces "better" interpolants, making IMC converge to a fixed point faster. Second, we apply frontier-set simplification, an optimization suggested in the original IMC paper [62]. Rather than taking the disjunction of previous interpolants, we retain only the latest one for the next interpolation query, which reduces solving effort while preserving the correctness.

Interpolation Sequence-Based Model Checking (ISMC). A new addition in Pono 2.0 is the ISMC engine [76]. Similar to IMC, ISMC is also based on unrolling of the transition system and Craig interpolation, but differs in how the interpolants are computed and used to construct a fixed point. Instead of deriving a single interpolant at each step, which represents a one-step overapproximation, ISMC derives a length n *inductive sequence* of interpolants, with each interpolant overapproximating reachable states at steps 1 to n. At each iteration, ISMC unrolls the system one step further and checks for reachability of bad states. If the BMC check is unsatisfiable, a new interpolation sequence is computed and conjoined with previously derived ones to form an evolving reachability sequence (a concept similar to *frames* in IC3/PDR [22,44]). The algorithm continues until either a counterexample is found or the reachability sequence forms a fixed point. Unlike IMC, ISMC preserves previously computed interpolants when the unrolling bound is increased, allowing information to be reused across iterations. Furthermore, ISMC issues only a single interpolation query per unrolling and immediately continues with the next unrolling, whereas IMC may pose multiple queries. This often enables ISMC to detect counterexamples more quickly in practice, although convergence to a fixed point may take longer [7,76].

Dual Approximated Reachability (DAR). Another new engine in Pono 2.0 is DAR [77], which is also based on interpolation. Compared to ISMC, which maintains only a forward reachability sequence (a sequence of overapproximations of states reachable from the initial state), DAR additionally maintains a backward reachability sequence (a sequence of overapproximations of states from which a bad state is reachable). Moreover, DAR interpolates in both forward and backward directions to simultaneously refine these sequences.

DAR primarily poses local interpolation queries that pair one forward and one backward element with a single step of the transition relation. Interpolants derived from these local queries extend and refine both the forward and backward sequences. When the information from local queries is insufficient for further refinement, DAR resorts to global queries that use multiple unrollings to derive

stronger interpolants. The algorithm proceeds until either a counterexample is found or one of the two reachability sequences forms a fixed point. Although DAR typically requires more interpolation queries than IMC or ISMC, these queries tend to be smaller and more tractable for the SMT solver. Previous studies [7,77] have shown that DAR complements IMC and ISMC by solving problem instances that are hard for the other two algorithms.

IC3 with Syntax-Guided Abstraction (IC3SA). IC3SA [49] is a word-level IC3 variant originally implemented in AVR [50]. Instead of reasoning at the bit level, it abstracts the state space using equivalence classes between the terms in the syntax of the transition system and the property. This abstraction ensures that the abstract state space has a size that is independent of the registers' bit widths, while at the same time capturing equality relations among the syntactic terms of the model. It can be further combined with uninterpreted functions for data abstraction. In Pono 1.0, the IC3SA implementation supported only bit-vectors, which is the version described in the original paper. Version 2.0 extends it to handle arrays (with bit-vector indices and values).

4.3 Liveness Model Checking

Pono 2.0 introduces support for verifying liveness properties in finite-state transition systems. Two approaches are implemented, both of which reduce liveness verification to safety verification and leverage the existing safety-checking infrastructure in Pono.

Liveness to Safety (L2S). L2S [11] is a standard reduction technique that transforms liveness checking into safety checking by detecting *revisited states* to identify lassos. It introduces a nondeterministic *save* oracle, modeled by a primary input variable, and a copy of the state variables to record the saved state. When the save variable is triggered, the current state is stored in the copied variables as the saved state. If the system later revisits the same state, a lasso is detected, indicating a counterexample to the liveness property. If no such revisit is possible, the property is proven. Our implementation supports multiple justice signals and can find both counterexamples and proofs. In Pono, L2S is implemented as a preprocessing step that transforms the transition system in place, allowing any safety-checking engine to be used directly on the resulting system. This design also allows other components, such as k-liveness, to reuse the transformation procedure seamlessly.

k-Liveness. The k-liveness algorithm [39] reduces liveness checking to safety checking by instrumenting the transition system with a counter that tracks how many times a justice signal has been triggered. Our implementation currently supports only a single justice signal. If no execution exists that triggers the signal k times, the algorithm reports that the property holds. Otherwise, it increments k and repeats the process. While k-liveness, in its original form, can only determine when a property holds, Pono extends it with additional procedures to also detect cases when the property is violated. Specifically, when

a trace is found that triggers the justice signal k times, Pono checks for the existence of a lasso (i.e., a revisited state) in two ways: (1) by analyzing the trace found in the previous step, and (2) by performing BMC in lock-step on the L2S-transformed system, as suggested in the original publication [39]. The first approach is incomplete but more lightweight, as it requires no additional solving. In practice, k-liveness has been shown to be complementary to L2S, particularly for proving when a liveness property holds. In Pono, users can configure the increment step for k as well as the counter encoding (binary or one-hot).

4.4 Counterexample-Guided Abstraction Refinement (CEGAR)

Pono provides CEGAR [40] wrappers on top of its safety-checking engines. In general, CEGAR works by constructing an abstract system that overapproximates the transition relation and then checking safety using this abstraction. If the property is proven (i.e., the bad state is unreachable), it holds for the concrete system as well. Otherwise, the obtained counterexample is analyzed: if it is spurious, the abstraction is refined, and the process repeats.

Arithmetic Operation Abstraction. A common approach to speeding up hardware model checking is to abstract complex arithmetic operations such as multiplication, division, or remainder using uninterpreted functions (UFs) [52]. During refinement, Pono reconstructs a BMC query guided by the abstract counterexample and adds concretization lemmas for each UF as assumptions. If the BMC query is unsatisfiable, an unsat core is extracted, and the operations involved in the core are concretized.

Although CEGAR with UF abstraction was already implemented in the first version of Pono [58], it was only integrated with the IC3SA and IC3IA engines. In Pono 2.0, we extend CEGAR integration to additional engines, including BMC, k-induction, IMC, and the newly-implemented ISMC and DAR. In addition to UF-based abstraction, Pono 2.0 also supports abstracting arithmetic operations using free variables, providing a coarser abstraction.

To be integrated into this CEGAR framework, a reachability engine must support two capabilities: exporting counterexamples (via `Prover::witness()`) for refinement and resetting its internal state (via `Prover::reset_env()`) to start a new CEGAR iteration. This design enables a largely black-box integration of engines. In the general case, the solver is reinitialized between iterations, favoring modularity over performance. For selected engines such as IC3IA and IC3SA, Pono provides a tighter integration that reuses solver state across refinements, improving efficiency while requiring additional implementation effort.

Counterexample-Guided Prophecy (CEGP). CEGP is a CEGAR-based approach for systems with arrays [57]. Array variables are abstracted using uninterpreted sorts, and array operations such as `select` and `store` are replaced with uninterpreted functions (UFs). This abstraction enables the use of SMT solvers that do not natively support the theory of arrays or do not have the functionality to compute interpolants. During refinement, it is not always possible to

eliminate spurious counterexamples using only quantifier-free lemmas over existing variables. To avoid the need for potentially expensive quantified lemmas, the approach introduces auxiliary *history* and *prophecy* variables [1]. These variables rule out spurious traces at the given bound and have been shown to help prove properties that would otherwise require universally quantified invariants [57].

Building on the CEGP implementation in Pono 1.0 [58], which only supported arrays with infinite-domain indices (e.g., mathematical integers), the current version extends support to finite-domain indices such as bit-vectors, which are prevalent in hardware model checking. Although our current implementation of CEGP in Pono 2.0 is incomplete for finite-domain indices due to its reliance on the refiner based on the array-property fragment [23], it remains practically useful. In particular, at HWMCC'25 [17], CEGP contributed to Pono solving the most uniquely-proved instances in the word-level array track. Additionally, Pono 2.0 also allows CEGP to be combined with arithmetic-operation abstraction, further enhancing its applicability to complex hardware-verification tasks.

4.5 Witness Generation

Pono can export violation witnesses for safety properties in Btor2 and VCD formats, and for liveness properties in Btor2 format. The witnesses can also be retrieved programmatically via the `Prover::witness()` method of the API. However, in the original version, this functionality was limited to unrolling-based engines such as BMC, k-induction, and IMC. In Pono 2.0, witness generation for violated properties has been extended to all engines, including IC3-based and CEGAR-based engines, as well as the newly integrated ISMC and DAR. Extensive testing has been conducted to improve the robustness of witness generation. Notably, Pono achieved a 100% witness confirmation rate at HWMCC'25 [17].

Pono does not yet support exporting correctness proofs as *witness circuits* for safety properties [46]. However, advanced users can retrieve the inductive invariant via the `Prover::invar()` API call. When the `check-invar` option is enabled, Pono can also internally validate the inductiveness of the computed invariant to ensure its correctness.

4.6 Parallel Portfolio

Parallel portfolios are a common approach for combining the strengths of multiple engines to improve overall performance. In Pono 2.0, we provide improved, separate scripts to run portfolios on Btor2 tasks with and without arrays, supporting safety properties and the exporting of violation witnesses. The portfolios were constructed with a data-driven approach: we first benchmarked a representative set of engines on HWMCC benchmark tasks from 2024 [16] and earlier, then applied an iterative greedy selection to choose engines that maximize the number of solved tasks when combined with the existing selections. Expert knowledge was used to further refine the portfolios to avoid overfitting. The portfolios are designed to be run on machines with at least 16 cores but

can be easily adapted to different hardware and use cases. Pono's parallel portfolio was essential to its strong performance in the word-level array track in HWMCC '25 [17].

4.7 SMT Solver Backends

Pono relies on Smt-Switch [59] to interface with SMT solvers in a uniform and solver-agnostic manner. For Pono 2.0, both Smt-Switch and the solver-integration layer have been extended to improve performance and configurability.

Updates in Smt-Switch. Pono 2.0 builds on the latest version of Smt-Switch, which has received several major updates since its initial release [59]. First, the CVC4 [6] backend has been upgraded to cvc5 [4], providing better performance, modernized APIs, and continued solver maintenance. Second, Bitwuzla [65] has been fully integrated with an updated API and extended to support interpolation, replacing Boolector [68] (which is no longer actively maintained) as the default SMT solver in Pono 2.0. Finally, incremental interpolation support has been added for MathSAT5 [36], allowing formulas to be reused across interpolation queries for improved efficiency. (This is also supported by Bitwuzla.) Together, these developments enable Pono to leverage the latest progress in SMT technology.

Configurability. To complement the backend updates, Pono 2.0 introduces several configurability enhancements. Users can now specify a dedicated interpolating solver that differs from the main SMT solver, providing additional flexibility for experimentation and tuning. Advanced users can also pass custom solver options (the options that can be set via `set-option` in SMT-LIB) through Pono, allowing fine-grained control over backend configurations. This feature has been particularly useful in evaluating the impact of different interpolation heuristics and parameters in Bitwuzla [66].

5 Related Work

This section reviews several model-checking tools related to Pono.

AVR [50] is a word-level model checker for safety properties expressed in the Btor2 and VMT-LIB formats. Its main engine is IC3SA [49], for which it is the original implementation. It also has basic implementations of BMC and k-induction, as well as a number of arithmetic and data abstraction techniques. AVR interfaces with Boolector [68], MathSAT5 [36], Yices2 [43], and Z3 [64] as SMT solver backends. In comparison, Pono additionally supports cvc5 [4] and Bitwuzla [65]. Pono also includes several additional verification engines (e.g., interpolation-based), offers a Python API, provides greater configurability, and supports liveness checking.

nuXmv [28] is a model-checking tool suite developed at Fondazione Bruno Kessler along with the MathSAT5 solver. It is the successor to nuSMV [32], which in turn evolved from the original SMV solver. Accordingly, its native

input language is SMV, though it also supports VMT-LIB and Btor2. nuXmv offers a diverse set of verification engines, including BMC, k-induction, IMC, IC3, and IC3IA, and supports both liveness and safety properties. However, it is closed-source and exclusively relies on MathSAT5 as its SMT backend, thereby limiting its extensibility.

Kind2 [29] is an SMT-based model checker that uses the Lustre input format [27]. It implements BMC, k-induction, IC3, and IC3IA and supports a wide variety of SMT solvers (Bitwuzla, cvc5, MathSAT5, OpenSMT [25], SMT-Interpol [31], Yices2, Yices, and Z3). Kind2 calls the solver binaries directly, passing the queries in SMT-LIB format. Although this provides similar flexibility to that provided by Smt-Switch in Pono, as evidenced by the number of supported solvers, it can incur a significant overhead and restricts solver usage to interactions via the command line.

rIC3 [72] and ABC [24] are two very successful bit-level model checkers for AIGER [18], which represents finite-state systems over the QF_BV theory. Btor2 extends AIGER to the word level and adds support for arrays, allowing higher-level system descriptions. While rIC3 also accepts Btor2 as an input, it does not support arrays and preprocesses the model by bit-blasting it to a purely propositional representation. rIC3 primarily handles safety properties and has a preliminary implementation of liveness analysis[5], whereas ABC supports both safety and liveness verification.

CoSA [60], an early prototype for what eventually became Pono, is an SMT-based model checker implemented in Python and built on PySMT [47]. It takes CoreIR [42] as input and supports only safety verification. Similarly, MoXIchecker [3] and PyVMT [48] are two other model-checking frameworks built on PySMT, which take MoXI [70] and VMT models as input, respectively. As of the latest release (0.9.6), PySMT supports a set of SMT solvers similar to that supported by Smt-Switch, except that it does not yet support Bitwuzla.

Constrained Horn Clauses (CHCs) offer an alternative way of encoding model checking problems. Instead of explicitly unrolling or symbolically exploring the transition system, CHC-based approaches encode verification tasks as sets of logical implications whose satisfiability corresponds to system correctness, i.e., the property holds. Spacer [54] and FreqHorn [45] are CHC solvers built on top of Z3, while the more recent Golem [20] employs OpenSMT as its back end.

6 Evaluation

Pono is intended to be used as an automated verification tool in hardware and software design flows. Therefore, we aim to make its performance competitive with other state-of-the-art tools. Pono 2.0 finished in first place in the bit-vectors with arrays track of HWMCC 2025 and in fifth place in the bit-vector-only track (behind two bit-level tools [55,72], a portfolio tool whose portfolio includes Pono itself [56], and AVR [50]). Here we provide a more comprehensive

[5] https://github.com/gipsyh/rIC3/pull/53

experimental evaluation of Pono's performance on various model checking tasks in both finite (bit-vectors) and infinite (integers and reals) domains.

Evaluated Tools. We compared Pono 2.0 (commit 60dd6b3)[6] against its previous version 1.0, along with other state-of-the-art model checkers, including nuXmv (2.1.0), and the HWMCC'25 versions of rIC3 (commit 5253520) and AVR (with fixes to enable it to run in our environment; commit 03bc7f6).

All compared tools are top contenders in recent editions of HWMCC. AVR was the overall winner at HWMCC'20. rIC3 won the bit-vectors-only tracks at HWMCC'24 and '25. nuXmv consistently ranked second or third in various tracks in HWMCC'20 and '24. Pono 1.0 was the overall winner at HWMCC'19 (under its previous name CoSA2) and the third-place finisher at HWMCC'20.

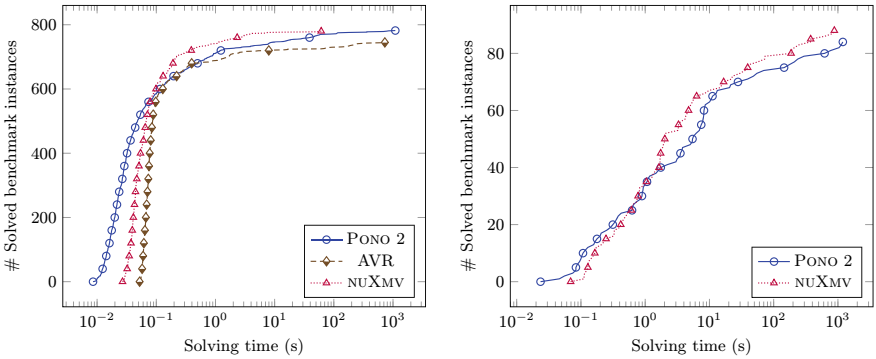
In the following, we present the portfolio results based on the virtual best solver computed from 16 configurations per tool, approximating the HWMCC setup. For Pono 2.0, we used modified HWMCC'25 portfolios; for Pono 1.0, an expanded HWMCC'20 portfolio (the original one has only 6 engines). We employed rIC3's 16-engine portfolio for HWMCC'25. AVR's HWMCC'25 portfolio contains 22 engines; we selected the top-performing 16. nuXmv's competition portfolio is not publicly available, so we evaluated all combinations of configuration options in its user manual [21] and selected the best-performing 16.

Pono 2.0 in this evaluation uses Bitwuzla [65,66] for bit-vector and array solving and interpolation,[7] cvc5 [4] for arithmetic logic solving, and MathSAT5 [36] for arithmetic logic interpolation. Pono 1.0 uses Boolector [67] for bit-vector and array solving and MathSAT5 for interpolation. rIC3 uses SAT solvers GipSAT [73], CaDiCaL [13], and Kissat [14] for bit-vector solving. AVR uses both Boolector [68] and Yices2 [43] for bit-vector and array solving and only Yices2 for arithmetic logic solving. nuXmv uses MathSAT5 for solving and interpolation for all logics. The evaluated portfolios are provided in the accompanying reproduction package [69].

Experimental Setup. All experiments were conducted on machines running Ubuntu 24.04 (64-bit), each equipped with an AMD Ryzen 9 7950X 16-core processor and 128 GiB of RAM. Each benchmark process corresponded to a run of a single model-checking engine, and was limited to one CPU core, 20 min of CPU time, and 8 GiB of memory. BenchExec [9] was employed to ensure reliable resource limits and measurement.

[6] While the experimental evaluation was conducted using a specific pre-release development commit of Pono, all features and engines described in this paper are fully integrated and available in the official 2.0 release.

[7] Bitwuzla has full interpolation support for bit-vectors and limited support for interpolation in other theories including uninterpreted functions and arrays; see their paper for details. MathSAT5 has similar limitations.



(a) 796 QF_LIA tasks (b) 99 QF_LRA tasks

Fig. 3. Number of VMT-LIB safety tasks solved by each tool

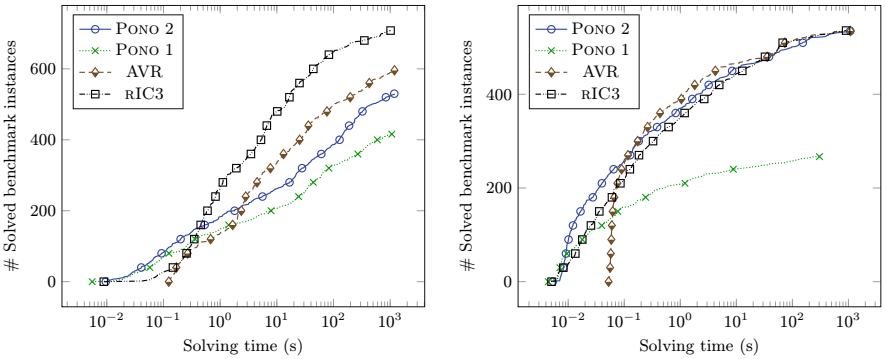
6.1 Safety Checking

We evaluated Pono's performance on a range of safety benchmarks covering integer and real arithmetic, fixed-size bit-vectors, and bit-vector arrays. For each category, we compared Pono 2.0 with state-of-the-art model checkers that support the corresponding input formats and theories, measuring both the number of instances solved and run times. The results highlight the improvements introduced in Pono 2.0 over its predecessor, as well as its competitive performance against other tools across diverse benchmark families.

Integer Arithmetic. We collected 796 verification tasks from the VMT-LIB benchmark set [29,37] using quantifier-free linear integer arithmetic (QF_LIA). Pono 2.0 was compared with AVR and nuXmv, both of which support the VMT-LIB format natively. (Pono 1.0 was excluded as it does not support VMT-LIB.) As shown in Fig. 3a, Pono solved 783 instances, outperforming AVR (746) and slightly surpassing nuXmv within the time limit. All three tools solved the majority of the tasks in 10 s. Notably, Pono was able to solve two instances uniquely and was the fastest on 551 tasks.

Real Arithmetic. Pono 2.0 was compared with nuXmv on 99 VMT-LIB tasks featuring quantifier-free linear real arithmetic (QF_LRA) [33,37]. We exclude AVR from the comparison as it does not support QF_LRA. The results shown in Fig. 3b indicate that Pono's performance (85 solved) is comparable to nuXmv (89 solved). In particular, Pono completed 37 tasks faster than nuXmv and was able to solve one instance that nuXmv could not.

Fixed-Size Bit-Vectors. We compared Pono 2.0 with the tools that natively support Btor2 (rIC3, AVR, and Pono 1.0) on two benchmark sets using quantifier-free bit-vectors (QF_BV). These comprise all 839 available word-level Btor2 tasks from previous HWMCCs [15,16,19], as well as the SV-COMP '25 [10] ReachSafety tasks translated to Btor2 by CPV [30,51]. Since



(a) 839 HWMCC tasks (b) 698 SV-COMP ReachSafety tasks

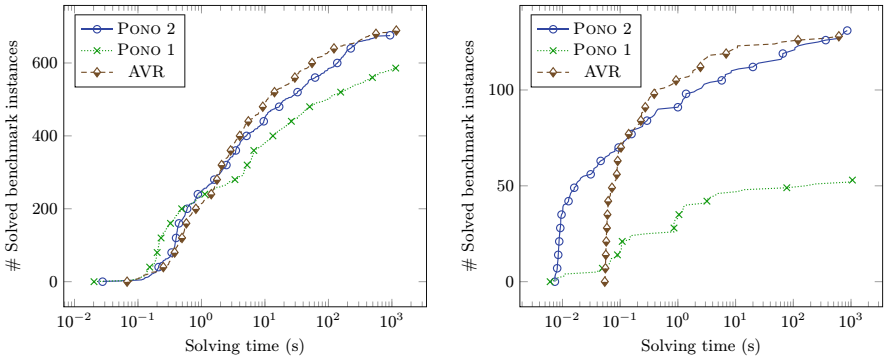
Fig. 4. Number QF_BV safety tasks solved by each tool

the HWMCC and SV-COMP sets overlap, and the latter contains several thousand instances, we sampled up to 16 benchmarks per family, 698 in total, from the SV-COMP set after excluding those already present in the HWMCC set.

Figures 4a and 4b summarize the results on the HWMCC and SV-COMP benchmark set, respectively. On the HWMCC benchmark set, rIC3 (709 solved) and AVR (597 solved) both outperformed Pono 2.0 (534 solved). Despite this, Pono 2.0 was the fastest on 212 of the tasks and uniquely solved 51 of them. On the SV-COMP benchmark set, Pono 2.0 solved the same number of instances as rIC3 (537) and one more than AVR. It was the fastest on 157 (more than any other tool) and uniquely solved six instances. Across both benchmark sets, Pono 2.0 consistently performed better than 1.0, which solved only 417 HWMCC and 268 SV-COMP instances.

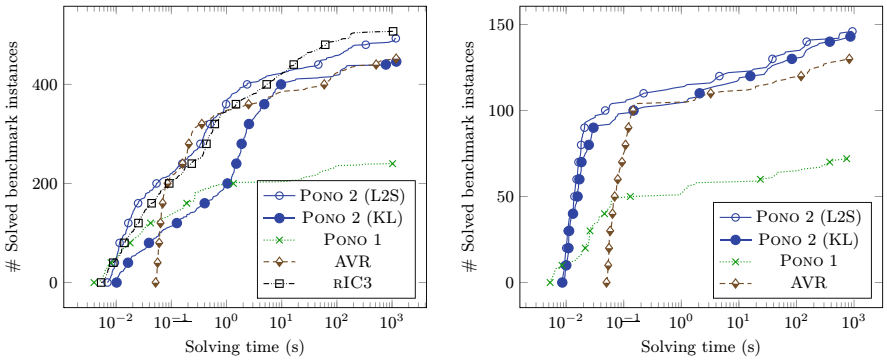
Several new features in Pono 2.0 are crucial for this improved performance, including arbitrary starting points and exponentially increasing step size in BMC, backward interpolation in IMC (Sect. 4.2), and arithmetic abstraction using free variables (Sect. 4.4) .

Bit-Vector Arrays. Following a similar process as for QF_BV, we collected benchmark tasks using quantifier-free arrays with bit-vector indices and values (QF_ABV) from previous HWMCCs (863) and the SV-COMP '25 ReachSafety track, removed duplicate instances, and sampled up to 16 benchmarks per family from the SV-COMP set (yielding 502). Pono 2.0 was compared with AVR and Pono 1.0, the only other tools that natively support QF_ABV in Btor2. On the HWMCC set (Fig. 5a, Pono 2.0 solved 677 tasks (fastest on 171; uniquely solved 13), outperforming Pono 1.0 (587) and closely matching AVR (690). On the SV-COMP set (Fig. 5b), Pono 2.0 solved 132 instances (fastest on 92; uniquely solved 8), surpassing both Pono 1.0 (54) and AVR (129).



(a) 863 HWMCC tasks (b) 502 SV-COMP ReachSafety tasks

Fig. 5. Number of QF_ABV safety tasks solved by each tool



(a) 839 QF_BV tasks (b) 220 QF_ABV tasks

Fig. 6. Number of translated SV-COMP '25 Termination tasks solved by each tool

6.2 Liveness Checking

Pono's implementations of L2S and k-liveness (KL), both utilizing the same underlying safety-checking engines as in the previous experiments, were evaluated alongside Pono 1.0, AVR, and rIC3. The benchmark tasks were collected from the SV-COMP '25 Termination track and translated from C to Btor2 by CPV [30]. Since the other tools do not support liveness (the keyword `justice` in Btor2) directly, they were run on tasks preprocessed with L2S by CPV .

Fixed-Size Bit-Vectors. Figure 6a compares Pono 2.0 with AVR, rIC3, and Pono 1.0 on 839 QF_BV liveness tasks. Pono 2.0 (running L2S) outperformed AVR and its predecessor, and is second only to rIC3. Pono's L2S also solved 48 unique tasks and was fastest on 488. Among the two liveness algorithms, L2S performed noticeably better than k-liveness (494 vs. 447 solved), although one task was solved exclusively by k-liveness, and it was the fastest for 7.

Bit-Vector Arrays. As shown in Fig. 6b, Pono 2.0 achieved better performance than either AVR or Pono 1.0 on liveness benchmark tasks using QF_ABV. L2S and k-liveness exhibited similar performance, solving 147 and 144 instances, respectively. L2S solved 3 uniquely and 114 fastest, while k-liveness had 33 fastest but no unique solves.

7 Conclusion and Future Work

We present Pono 2.0, a major evolution of our SMT-based model checker. It introduces support for liveness, new and improved safety engines, a new input format, and usability and performance enhancements. Thanks to its modular and extensible design, Pono 2.0 is both a dependable tool for push-button verification and a foundation for exploring novel verification techniques. In the future, we plan to continue expanding Pono's capabilities, including support for the MoXI [70] language, exporting correctness proofs as witness circuits [46], and integrating new liveness-checking algorithms such as rlive [34,78].

Acknowledgements. We would like to thank Makai Mann, the original author of Pono [58]; Luiza Pires Ribeiro, who helped identify issues in Pono's witness-generation component; and Anna Eaton, who initiated work on MoXI support for Pono. We also thank all other contributors listed in the public commit log. We are especially grateful to Aina Niemetz and Mathias Preiner for their work on making Bitwuzla [65] a powerful SMT solver for Pono, including its support for producing interpolants for bit-vectors [66]. We thank Yoni Zohar for his work on Smt-Switch [59] and for contributions to several solvers used by Pono. Finally, we thank the developers of cvc5 [4] and MathSAT5 [36], which are also used by Pono.

Funding Information. Po-Chun Chien is supported by the German Research Foundation (DFG) under grants 378803395 (ConVeY) and 536040111 (Bridge), as well as a fellowship from the German Academic Exchange Service (DAAD). This work was also supported by the Stanford Center for Portable Accelerated Learning (Portal) and the Stanford Center for Automated Reasoning.

Data Availability Statement. A reproduction package for this article is available on Zenodo [69]. The source code for both Pono and Smt-Switch is hosted on GitHub under the permissive BSD 3-Clause License.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991). https://doi.org/10.1016/0304-3975(91)90224-P
2. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985). https://doi.org/10.1016/0020-0190(85)90056-0
3. Ates, S., Beyer, D., Chien, P.C., Lee, N.Z.: MoXIchecker: An extensible model checker for MoXI. In: Protzenko, J., Raad, A. (eds.) Proc. VSTTE. LNCS, vol. 15525, pp. 1–14. Springer (2024). https://doi.org/10.1007/978-3-031-86695-1_1

4. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Proc. TACAS (1). LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
5. Barrett, C., Fontaine, P., Tinelli, C.: The satisfiability modulo theories library (SMT-LIB) (2016). https://smt-lib.org
6. Barrett, C.W., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proc. CAV. LNCS, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
7. Beyer, D., Chien, P.C., Jankola, M., Lee, N.Z.: A transferability study of interpolation-based hardware model checking for software verification. Proc. ACM Softw. Eng. **1**(FSE), 2028–2050 (2024). https://doi.org/10.1145/3660797
8. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: adoption to software verification. J. Autom. Reason. **69**(1), 5 (2025). https://doi.org/10.1007/S10817-024-09702-9
9. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/S10009-017-0469-Y
10. Beyer, D., Strejcek, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Gurfinkel, A., Heule, M. (eds.) Proc. TACAS (3). LNCS, vol. 15698, pp. 151–186. Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9
11. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. **66**(2), 160–177 (2002). https://doi.org/10.1016/S1571-0661(04)80410-9
12. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003). https://doi.org/10.1016/S0065-2458(03)58003-2
13. Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., Pollitt, F.: CaDiCaL 2.0. In: Gurfinkel, A., Ganesh, V. (eds.) Proc. CAV (1). LNCS, vol. 14681, pp. 133–152. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_7
14. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. SAT Competition – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020). https://tuhat.helsinki.fi/ws/files/142452772/sc2020_proceedings.pdf.#page=50
15. Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition 2020. https://hwmcc.github.io/2020/. Accessed 15 Oct 2025
16. Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition 2024. In: Narodytska, N., Rümmer, P. (eds.) Proc. FMCAD, p. 7. TU Wien Academic Press (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_6
17. Biere, A., Froleyks, N., Preiner, M.: Hardware model checking competition 2025. In: Irfan, A., Kaufmann, D. (eds.) Proc. FMCAD. p. 7. TU Wien Academic Press (2025). https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_6
18. Biere, A., Heljanko, K., Wieringa, S.: AIGER 1.9 and beyond. Tech. Rep. 11/2, Institute for Formal Models and Verification, Johannes Kepler University (2011). https://doi.org/10.35011/fmvtr.2011-2
19. Biere, A., Preiner, M.: Hardware model checking competition 2019. https://fmv.jku.at/hwmcc19/, Accessed 15 Oct 2025

20. Blicha, M., Britikov, K., Sharygina, N.: The Golem Horn solver. In: Enea, C., Lal, A. (eds.) Proc. CAV (2). LNCS, vol. 13965, pp. 209–223. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_10
21. Bozzano, M., et al.: nuXmv 2.1.0 user manual. Tech. rep., Fondazione Bruno Kessler (2024). https://nuxmv.fbk.eu/downloads/nuxmv-user-manual.pdf
22. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Proc. VMCAI. LNCS, vol. 6538, pp. 70–87. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
23. Bradley, A.R., Manna, Z.: The calculus of computation - decision procedures with applications to verification. Springer (2007). https://doi.org/10.1007/978-3-540-74113-8
24. Brayton, R.K., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P.B. (eds.) Proc. CAV. LNCS, vol. 6174, pp. 24–40. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5
25. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: Esparza, J., Majumdar, R. (eds.) Proc. TACAS. LNCS, vol. 6015, pp. 150–153. Springer (2010). https://doi.org/10.1007/978-3-642-12002-2_12
26. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992). https://doi.org/10.1016/0890-5401(92)90017-A
27. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: LUSTRE: a declarative language for programming synchronous systems. In: Proc. POPL, pp. 178–188. ACM (1987). https://doi.org/10.1145/41625.41641
28. Cavada, R., et al.: The nuXmv Symbolic Model Checker. In: Biere, A., Bloem, R. (eds.) Proc. CAV. LNCS, vol. 8559, pp. 334–342. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
29. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 Model Checker. In: Chaudhuri, S., Farzan, A. (eds.) Proc. CAV (2). LNCS, vol. 9780, pp. 510–517. Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_29
30. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier. In: Finkbeiner, B., Kovács, L. (eds.) Proc. TACAS (3). LNCS, vol. 14572, pp. 365–370. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
31. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An interpolating SMT solver. In: Donaldson, A.F., Parker, D. (eds.) Proc. SPIN. LNCS, vol. 7385, pp. 248–254. Springer (2012). https://doi.org/10.1007/978-3-642-31759-0_19
32. Cimatti, A., et al.: NuSMV 2: An Opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) Proc. CAV. LNCS, vol. 2404, pp. 359–364. Springer (2002). https://doi.org/10.1007/3-540-45657-0_29
33. Cimatti, A., Griggio, A.: Software Model Checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) Proc. CAV. LNCS, vol. 7358, pp. 277–293. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_23
34. Cimatti, A., Griggio, A., Johannsen, C., Rozier, K.Y., Tonetta, S.: Infinite-state liveness checking with rlive. In: Piskac, R., Rakamaric, Z. (eds.) Proc. CAV (1). LNCS, vol. 15931, pp. 215–236. Springer (2025). https://doi.org/10.1007/978-3-031-98668-0_11
35. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Ábrahám, E., Havelund, K. (eds.) Proc. TACAS. LNCS, vol. 8413, pp. 46–61. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_4

36. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) Proc. TACAS. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
37. Cimatti, A., Griggio, A., Tonetta, S.: Verification Modulo Theories (language, benchmarks and tools) (2015). https://vmt-lib.fbk.eu/
38. Cimatti, A., Griggio, A., Tonetta, S.: The VMT-LIB language and tools. In: Déharbe, D., Hyvärinen, A.E.J. (eds.) Proc. SMT. CEUR Workshop Proceedings, vol. 3185, pp. 80–89. CEUR-WS.org (2022). https://ceur-ws.org/Vol-3185/extended9547.pdf
39. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: Cabodi, G., Singh, S. (eds.) Proc. FMCAD, pp. 52–59. IEEE (2012). https://ieeexplore.ieee.org/document/6462555/
40. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Proc. CAV. LNCS, vol. 1855, pp. 154–169. Springer (2000). https://doi.org/10.1007/10722167_15
41. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
42. Daly, R.: CoreIR: A simple LLVM-style hardware compiler. https://github.com/rdaly525/coreir, Accessed 15 Oct 2025
43. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Proc. CAV. LNCS, vol. 8559, pp. 737–744. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
44. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Bjesse, P., Slobodová, A. (eds.) Proc. FMCAD, pp. 125–134. IEEE (2011). https://ieeexplore.ieee.org/document/6148886
45. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Proc. CAV (1). LNCS, vol. 11561, pp. 259–277. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_14
46. Froleyks, N., Yu, E., Preiner, M., Biere, A., Heljanko, K.: Introducing certificates to the hardware model checking competition. In: Piskac, R., Rakamaric, Z. (eds.) Proc. CAV (1). LNCS, vol. 15931, pp. 281–295. Springer (2025). https://doi.org/10.1007/978-3-031-98668-0_14
47. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proc. SMT (2015). https://andrea.micheli.website/papers/pysmt.pdf
48. Girardi, A., Bombardelli, A., Griggio, A., Tomassi, J.: PyVMT: A Python library to interact with transition systems. https://github.com/pyvmt/pyvmt, accessed: 2025-10-15
49. Goel, A., Sakallah, K.A.: Model checking of Verilog RTL using IC3 with syntax-guided abstraction. In: Badger, J.M., Rozier, K.Y. (eds.) Proc. NFM. LNCS, vol. 11460, pp. 166–185. Springer (2019). https://doi.org/10.1007/978-3-030-20652-9_11
50. Goel, A., Sakallah, K.: AVR: Abstractly Verifying Reachability. In: Proc. TACAS (1). LNCS, vol. 7795, pp. 413–422. Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_23
51. Griggio, A., Jonás, M.: Kratos2: An SMT-based model checker for imperative programs. In: Enea, C., Lal, A. (eds.) Proc. CAV (3). LNCS, vol. 13966, pp. 423–436. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_20
52. Ho, Y.S., Chauhan, P., Roy, P., Mishchenko, A., Brayton, R.K.: Efficient uninterpreted function abstraction and refinement for word-level model checking. In:

Piskac, R., Talupur, M. (eds.) Proc. FMCAD, pp. 65–72. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886662
53. Johannsen, C., et al.: The MoXI model exchange tool suite. In: Gurfinkel, A., Ganesh, V. (eds.) Proc. CAV (1). LNCS, vol. 14681, pp. 203–218. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_10
54. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-Based Model Checking for Recursive Programs. In: Biere, A., Bloem, R. (eds.) Proc. CAV. LNCS, vol. 8559, pp. 17–34. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_2
55. Li, J., Zhu, S., Zhang, Y., Pu, G., Vardi, M.Y.: Safety model checking with complementary approximations. In: Parameswaran, S. (ed.) Proc. ICCAD. pp. 95–100. IEEE (2017). https://doi.org/10.1109/ICCAD.2017.8203765
56. Lu, Z., Chien, P.C., Lee, N.Z., Gurfinkel, A., Ganesh, V.: Btor2-Select: Machine learning based algorithm selection for hardware model checking. In: Piskac, R., Rakamaric, Z. (eds.) Proc. CAV (1). LNCS, vol. 15931, pp. 296–311. Springer (2025). https://doi.org/10.1007/978-3-031-98668-0_15
57. Mann, M., Irfan, A., Griggio, A., Padon, O., Barrett, C.W.: Counterexample-guided prophecy for model checking modulo the theory of arrays. Log. Methods Comput. Sci. **18**(3) (2022). https://doi.org/10.46298/LMCS-18(3:26)2022
58. Mann, M., et al.: Pono: A flexible and extensible SMT-based model checker. In: Silva, A., Leino, K.R.M. (eds.) Proc. CAV (2). LNCS, vol. 12760, pp. 461–474. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81688-9_22
59. Mann, M., et al.: Smt-Switch: A solver-agnostic C++ API for SMT solving. In: Li, C.M., Manyà, F. (eds.) Proc. SAT. LNCS, vol. 12831, pp. 377–386. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_26
60. Mattarei, C., Mann, M., Barrett, C.W., Daly, R.G., Huff, D., Hanrahan, P.: CoSA: Integrated verification for agile hardware design. In: Bjørner, N.S., Gurfinkel, A. (eds.) Proc. FMCAD, pp. 1–5. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603014
61. McMillan, K.L.: Symbolic model checking: An approach to the state explosion problem. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1992). https://apps.dtic.mil/sti/tr/pdf/ADA250924.pdf
62. McMillan, K.L.: Interpolation and SAT-based model checking. In: Jr., W.A.H., Somenzi, F. (eds.) Proc. CAV. LNCS, vol. 2725, pp. 1–13. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
63. Melchert, J., Terrill, C., Perez-Lopez, Á.R., Barrett, C., Raina, P.: Automated translation validation of a compiler for statically scheduled accelerators. In: Irfan, A., Kaufmann, D. (eds.) Proc. FMCAD, pp. 198–208. TU Wien Academic Press (2025). https://doi.org/10.34727/2025/isbn.978-3-85448-084-6_26
64. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Proc. TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
65. Niemetz, A., Preiner, M.: Bitwuzla. In: Enea, C., Lal, A. (eds.) Proc. CAV (2). LNCS, vol. 13965, pp. 3–17. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_1
66. Niemetz, A., Preiner, M.: Bit-precise interpolation in Bitwuzla. In: Junges, S., Katz, G. (eds.) Proc. TACAS. LNCS, vol. 16505, pp. 66–88. Springer (2026). https://doi.org/10.1007/978-3-032-22752-2_4
67. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. J. Satisf. Boolean Model. Comput. **9**(1), 53–58 (2014). https://doi.org/10.3233/SAT190101

68. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) Proc. CAV (1). LNCS, vol. 10981, pp. 587–595. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
69. Perez-Lopez, Á.R., Chien, P.C., Lonsing, F., Archer, S., Irfan, A., Barrett, C.: FM 2026 artifact for Pono 2.0: A versatile SMT-based model checker for safety and liveness. Zenodo (2026). https://doi.org/10.5281/zenodo.18680797
70. Rozier, K.Y., et al.: MoXI: An intermediate language for symbolic model checking. In: Neele, T., Wijs, A. (eds.) Proc. SPIN. LNCS, vol. 14624, pp. 26–46. Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_2
71. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Jr., Johnson, S.D. (eds.) Proc. FMCAD. LNCS, vol. 1954, pp. 108–125. Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
72. Su, Y., Yang, Q., Ci, Y., Bu, T., Huang, Z.: The rIC3 hardware model checker. In: Piskac, R., Rakamaric, Z. (eds.) Proc. CAV (1). LNCS, vol. 15931, pp. 185–199. Springer (2025). https://doi.org/10.1007/978-3-031-98668-0_9
73. Su, Y., Yang, Q., Ci, Y., Li, Y., Bu, T., Huang, Z.: Deeply optimizing the SAT solver for the IC3 algorithm. In: Piskac, R., Rakamaric, Z. (eds.) Proc. CAV (1). LNCS, vol. 15931, pp. 237–257. Springer (2025).https://doi.org/10.1007/978-3-031-98668-0_12
74. Tollec, S., Asavoae, M., Couroussé, D., Heydemann, K., Jan, M.: μArchiFI: formal modeling and verification strategies for microarchitectural fault injections. In: Nadel, A., Rozier, K.Y. (eds.) Proc. FMCAD, pp. 101–109. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_18
75. Tsiskaridze, N., et al.: Automating system configuration. In: Proc. FMCAD, pp. 102–111. TU Wien Academic Press (2021). https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4_19
76. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Proc. FMCAD, pp. 1–8. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351148
77. Vizel, Y., Grumberg, O., Shoham, S.: Intertwined forward-backward reachability analysis using interpolants. In: Piterman, N., Smolka, S.A. (eds.) Proc. TACAS. LNCS, vol. 7795, pp. 308–323. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_22
78. Xia, Y., Cimatti, A., Griggio, A., Li, J.: Avoiding the shoals - A new approach to liveness checking. In: Gurfinkel, A., Ganesh, V. (eds.) Proc. CAV (1). LNCS, vol. 14681, pp. 234–254. Springer (2024). https://doi.org/10.1007/978-3-031-65627-9_12
79. Yan, Z., Zhang, H.: Word-level augmentation of formal proof by learning from simulation traces. In: Xiong, J., Wille, R. (eds.) Proc. ICCAD, pp. 115:1–115:9. ACM (2024). https://doi.org/10.1145/3676536.3676686