MoXIchecker: An Extensible Model Checker for MoXI

Salih Ates, Dirk Beyer, Po-Chun Chien, and Nian-Ze Lee

LMU Munich, Munich, Germany

Abstract. MoXI is a new intermediate verification language introduced in 2024 to promote the standardization and open-source implementations for symbolic model checking by extending the SMT-LIB 2 language with constructs to define state-transition systems. The tool suite of MoXI provides a translator from MoXI to Btor2, which is a lower-level intermediate language for hardware verification, and a translation-based model checker, which invokes mature hardware model checkers for Btor2 to analyze the translated verification tasks. The extensibility of such a translation-based model checker is restricted because more complex theories, such as integer or real arithmetics, cannot be precisely expressed with bit-vectors of fixed lengths in Btor2. We present MoXIchecker, the first model checker that solves MoXI verification tasks directly. Instead of translating MoXI to lower-level languages, MoXIchecker uses the solver-agnostic library PySMT for SMT solvers as backend for its verification algorithms. MoXIchecker is *extensible* because it accommodates verification tasks involving more complex theories, not limited by lower-level languages, facilitates the implementation of new algorithms, and is solver-agnostic by using the API of PySMT. In our evaluation, MoXIchecker uniquely solved tasks that use integer or real arithmetics, and achieved a comparable performance against the translation-based model checker from the MoXI tool suite.

Keywords: Formal verification · Symbolic model checking · Intermediate language · MoXI · Btor2 · SMT · SAT · PySMT · Exchange formats

1 Introduction

Symbolic model checking [1, 2] embraces a wide range of automatic techniques to formally verify a model against a specification by encoding and searching the state space symbolically. It has been applied to hardware, software, and cyber-physical systems to ensure their safety and correct functionality. However, symbolic model checking has not been adopted as widely as other "push-button" techniques for quality assurance, such as testing, especially in industry. A major challenge is the lack of standardized exchange formats and open-source implementations [3, 4]. Even though model checkers from the same research community work on the same type of computational models, they often use different input formats, which hinders the propagation of techniques. Moreover, some model checkers are closed-source and make the comparison of verification algorithms complicated, because techniques may need to be re-implemented in a different framework to achieve fair comparison (this makes expensive transferability studies necessary [5, 6]).

Recently, a new intermediate verification language MoXI [3], the *model exchange interlingua*, has been proposed to address the aforementioned challenge. MoXI aims to be (1) as expressive as necessary to accommodate real-world applications described in user-facing, higher-level modeling languages and (2) as simple as possible to facilitate its translation to tool-oriented, lower-level intermediate languages, for which efficient and effective model checkers are available. It augments the SMT-LIB 2 [7] format with constructs to define state-transition systems by using formulas in first-order logic to encode their initial and transition conditions. MoXI inherits the expressiveness of SMT-LIB 2 and offers abundant background theories to represent various computational models, ranging from hardware circuits and software programs to cyber-physical systems. The precise semantics of SMT-LIB 2 also enables the translation from MoXI to lower-level intermediate languages. Using SMT formulas to precisely describe state-transition systems has also been studied in the VMT [8] intermediate language.

Compared to other intermediate verification languages, such as the SMV [9] language for finite-state transition systems or the Boogie [10] language for software programs, using MoXI to represent model-checking problems frees backend verification engines from encoding the semantics of frontend modeling languages into SMT formulas. This separation of frontend and backend will help decompose monolithic model checkers into several modular and reusable components, e.g., standalone translators from higher-level frontend languages to MoXI and efficient model-checking engines for MoXI verification tasks. A deeper discussion can be found in a recent survey on transformation for verification [11].

1.1 Existing Tool Suite for MoXI

The tool suite of MoXI [12] offers translators from SMV to MoXI and from MoXI to the word-level modeling language Btor2 [13], the prevailing format for hardware model checking. The tool suite also implements a translation-based model checker, MoXI-MC-Flow, by translating a MoXI task to an equisatisfiable Btor2 task and invoking Btor2 model checkers, such as AVR [14], BtorMC [13], and Pono [15], on the translated task. Translation-based verification approaches have been actively studied in the literature. For example, sequential circuits in Verilog [16] can be translated to SMV models [17, 18] or C programs [19] for verification. Btor2 circuits have been translated to C programs and analyzed by software verifiers [20, 21, 22]. C programs can also be translated to SMV or Btor2 models and verified by hardware model checkers [23, 24].

While MoXI-MC-Flow can solve MoXI verification tasks by translating them to Btor2 [12], the translation-based approach limits the expressiveness of the model-checking flow because verification problems cannot be precisely represented in Btor2 if they involve more complex background theories, such as integer or real arithmetics. Moreover, to extend MoXI-MC-Flow with new algorithms, tool developers need to dig into the Btor2 model checkers.

1.2 Motivation to Develop MoXIchecker

To address the extensibility gap of the translation-based model-checking flow for MoXI, we implemented MoXIchecker, the first model checker that solves MoXI verification tasks directly without translating them to other intermediate languages. MoXIchecker takes as input a MoXI verification task, constructs the SMT formulas used to define the task, and implements its verification algorithms using the API of PySMT [25], a solver-agnostic Python library for SMT solvers. Currently, MoXIchecker supports the quantifier-free theories of bit-vectors, arrays, integers, and reals, and the implemented algorithms include BMC [26], k-induction [27], and IC3/PDR [28].

The benefits of MoXIchecker compared to MoXI-MC-Flow are threefold. First, MoXIchecker enjoys the complete expressiveness of SMT-LIB 2 and is applicable to verification tasks involving more complex background theories, as long as there exists an SMT solver supporting the used theory. In contrast, MoXI-MC-Flow is inadequate if the used theory is not representable in lower-level intermediate languages focusing on bit-vectors of fixed lengths and arrays. Second, MoXIchecker allows for convenient extension and fast prototyping of model-checking algorithms. To develop a new algorithm in MoXIchecker, one can simply work with the SMT formulas describing the model and manipulate them via the API of PySMT. In contrast, adding a new algorithm to the hardware model checkers used by MoXI-MC-Flow involves dealing with Btor2 circuits. Moreover, MoXIchecker enables fair comparison of algorithms because the number of confounding variables (e.g., same parser, same SMT solver, same libraries) is kept to a minimum. Third, MoXIchecker has a robust frontend design because constructing SMT formulas that describe a MoXI verification task via PySMT is purely syntactical and less error-prone than translating the SMT formulas to Btor2.

Furthermore, MoXIchecker is meant for use in education. It is an ideal framework for playing around with algorithms in course projects. The tool has a clean architecture and a slim code base.

Contributions. To sum up, our contributions in this paper include:

1. MoXIchecker, the first model checker that verifies MoXI tasks directly,
2. implemented as an extensible framework to accommodate various background theories and facilitate the development of algorithms for MoXI,
3. MoXIchecker's first three algorithms, BMC, k-induction, and IC3/PDR, and
4. an evaluation of MoXIchecker with MoXI-MC-Flow on about 400 MoXI verification tasks.

In our experiments, MoXIchecker solved a similar number of bit-vector tasks as MoXI-MC-Flow, which used highly-optimized Btor2 model checkers as backend. Moreover, MoXIchecker was able to uniquely solve tasks using real arithmetics, which MoXI-MC-Flow cannot handle. These contributions are significant and novel because MoXIchecker supports the standardization of symbolic model checking around MoXI and provides an extensible framework for open-source implementations of verification algorithms for MoXI.

2 Background

In this section, we provide background knowledge for symbolic model checking and the intermediate verification language MoXI.

2.1 Symbolic Model Checking

The problem of symbolic model checking [1, 2] is to decide whether a model, usually represented as a *state-transition system* [29, 30], satisfies a specification. A state-transition system $\mathcal{M}$ can be described by an *initial condition* $I(s)$, a *transition condition* $T(s, s')$, and an *invariance condition* $Inv(s)$, where s and s' range over possible states of $\mathcal{M}$. Condition $I(s)$ evaluates to $\top$ if state s is an initial state of $\mathcal{M}$, and $T(s, s')$ evaluates to $\top$ if state s can transit to state s' via one step in $\mathcal{M}$ (we use $\top$ for *true*). A state $\widehat{s}$ is *reachable* if $I(\widehat{s})$ evaluates to $\top$ or $I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, \widehat{s})$ is satisfiable for some $k \geq 1$. Condition $Inv(s)$ is a constraint imposed on all reachable states in $\mathcal{M}$ (a reachable state that violates Inv is excluded for analysis).

A specification φ can be represented by a formula in linear temporal logic (LTL) [31], which is evaluated over the execution traces of a state-transition system. In the following, we refer to the tuple $(\mathcal{M}, \varphi)$ as a *verification task*, which asks if state-transition system $\mathcal{M}$ satisfies specification φ. *Reachability safety* is an essential category of specifications, inspecting the reachability of some target states marked by a *reachable condition* $Q(s)$. A reachability-safety verification task is described by the tuple (I, T, Inv, Q), where I, T, and Inv define a state-transition system $\mathcal{M}$ and Q defines an LTL formula "**always** $\neg Q$" as a specification φ for $\mathcal{M}$. A reachability-safety verification task is *safe* (resp. *unsafe*) if the target states are unreachable (resp. reachable).

In the research community of hardware model checking, verification tasks of sequential circuits can be encoded by the word-level language Btor2 [13].

2.2 The Intermediate Verification Language MoXI

MoXI [3] extends the SMT-LIB 2 [7] format with constructs to describe verification tasks. Inheriting the expressiveness of SMT-LIB 2, MoXI offers a variety of background theories, ranging from bit-vectors and arrays (`QF_BV` and `QF_ABV`) to linear and nonlinear arithmetics over integers and reals (`QF_LIA`, `QF_LRA`, `QF_NIA`, and `QF_NRA`), to represent models of hardware, software, and cyber-physical systems. As for specifications, MoXI supports reachability-safety queries with fairness constraints. We refer interested readers to the language design of MoXI [3] for more details. In the following, we use an example to show how a verification task is represented in MoXI.

Figure 1 shows a verification task of a three-bit counter in MoXI. Line 1 sets the background theory to `QF_BV`, which allows for quantifier-free formulas over the theory of bit-vectors with fixed sizes. Lines 2 to 8 define the behavior of the three-bit counter with command `define-system` and name the counter `main`. Counter `main` has an output variable `s`, which is a bit-vector of length three (attribute `:output` in line 4). Counter `main` has no inputs or local variables (attributes `:input` in line 3 and `:local` in line 5, respectively).

```
1  (set-logic QF_BV)
2  (define-system main
3    :input ()
4    :output ((s (_ BitVec 3)))
5    :local ()
6    :init (= s #b000)
7    :trans (= s′ (bvadd s #b010))
8    :inv true)
9  (check-system main
10   :input ()
11   :output ((s (_ BitVec 3)))
12   :local ()
13   :reachable (rch_1
         (= (bvurem s #b010) #b001))
14   :query (qry_rch_1 (rch_1)))
```

Fig. 1: An example verification task in MoXI

The initial condition in line 6 (attribute `:init`) initializes output `s` of counter `main` to `#b000`. The transition condition in line 7 (attribute `:trans`) increments the value of `s` by `#b010` in each step. Note that a primed variable is treated as the next-state variable of its unprimed counterpart by MoXI. That is, `s'` holds the value of `s` after one step. The invariance condition in line 8 (attribute `:inv`) imposes `true` as a constraint on all reachable states of counter `main`. The specification for counter `main` is described by command `check-system`. The reachability condition `rch_1` in line 13 (attribute `:reachable`) states that the value of `s` is an odd number, i.e., the remainder of `s` divided by `#b010` equals `#b001`. Line 14 poses a query `qry_rch_1` (attribute `:query`) to examine whether the LTL formula "**always** ¬`rch_1`" is satisfied by all execution traces of counter `main`.

The MoXI tool suite [12] provides an alternative representation of MoXI verification tasks in JSON format to facilitate tool development and information exchange. Figure 6 in Appendix A shows the corresponding JSON file for the verification task in Fig. 1. Our tool MoXIchecker takes MoXI verification tasks in JSON format as input.

To analyze a MoXI verification task, the model checker MoXI-MC-Flow in the MoXI tool suite translates the MoXI task to an equisatisfiable Btor2 verification task and invokes hardware model checkers for Btor2, e.g., AVR [14], BtorMC [13], and Pono [15], from the Hardware Model Checking Competitions [32].

3 Software Architecture of MoXIchecker

Figure 2 shows the software architecture of MoXIchecker, the first model checker for MoXI without translating verification tasks to lower-level languages. Implemented in the programming language Python, MoXIchecker is open-source on GitLab[1] and released under the Apache License 2.0. On a MoXI verification task in JSON format, MoXIchecker uses the standard JSON package of Python to load the input file and constructs SMT formulas for the initial, transition, invariance, and reachable conditions by calling the API of the solver-agnostic library PySMT [25] for SMT solvers. It then performs model checking on the reachability-safety verification task (I, T, Inv, Q). The output of MoXIchecker on a MoXI verification task is a verdict to indicate whether the task is safe or unsafe.

Different from MoXI-MC-Flow in the MoXI tool suite [12], which translates verification tasks in MoXI to Btor2 [13] and invokes hardware model checkers, MoXIchecker implements its model-checking engines using the API of PySMT. Currently, MoXIchecker supports `QF_BV`, `QF_ABV`, `QF_LIA`, `QF_LRA`, `QF_NIA`, and

[1] https://gitlab.com/sosy-lab/software/moxichecker

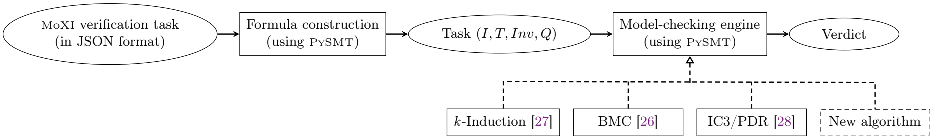

Fig. 2: Software architecture of MoXIchecker

`QF_NRA` as the background theory. The elegant software architecture of MoXIchecker facilitates the addition of new background theories and yields a slim code base (about 700 lines in release 0.2).

We adapted and integrated the implementations of BMC [26], k-induction [27], and IC3/PDR [28] in PySMT[2] into our framework. In addition, we demonstrate the extensibility of MoXIchecker by contributing a k-induction implementation that takes advantage of incremental solving of SMT solvers by reusing solver stacks. Compared to the non-incremental version in PySMT, the incremental k-induction was more efficient and solved more tasks in the evaluation.

3.1 Example

We demonstrate the working of MoXIchecker by invoking it on the verification task in Fig. 1. MoXIchecker consumes the JSON file of the MoXI verification task in Fig. 6 as input and constructs SMT formulas $s = 0$, $s' = s + 2$, $\top$, and $s\%2 = 1$, as the initial, transition, invariance, and reachability conditions, respectively. Note that variable s is a bit-vector of length three, and variable s' is its next-state counterpart. To honor the invariance condition, MoXIchecker conjoins it with initial and transition conditions, respectively. As the invariance condition is $\top$ in the example verification task, we omit it in the following explanation.

To solve the verification task, MoXIchecker considers "**always** $\neg(s\%2 = 1)$" as the specification for the state-transition system. By applying k-induction [27], MoXIchecker shows that both the *base case* $(s = 0) \Rightarrow \neg(s\%2 = 1)$ and the *step case* $\neg(s\%2 = 1) \wedge (s' = s + 1) \Rightarrow \neg(s'\%2 = 1)$ for $k = 1$ hold. Therefore, MoXIchecker concludes that counter `main` in Fig. 1 satisfies its specification.

3.2 Discussion

In the following, we discuss the ongoing development progress of MoXIchecker, the differences between MoXIchecker and a related model-checking framework PyVMT [33], and the trade-offs of using the programming language Python.

The current version of MoXIchecker (release 0.2) misses the support for creating subsystems (via attribute `:subsys` in command `define-system`), a key feature of MoXI to compose multiple state-transition systems. MoXIchecker also needs further enhancement to handle fairness constraints (via attribute `:fairness` in command `check-system`) and background theories with quantifiers. Compared to MoXI-MC-Flow, which has not yet supported quantifiers, supporting quantifiers in MoXIchecker is straightforward because it is not limited by lower-level modeling languages like Btor2. In addition, MoXI defines a format for *verification witnesses* [34, 35], e.g., an error trace if the specification is violated or an

[2] https://github.com/pysmt/pysmt/blob/master/examples/model_checking.py

invariant if the specification is satisfied. We are actively extending the language support of MoXIchecker to cover all features of MoXI.

PyVMT [33] is a Python library to construct and verify transition systems specified in the language VMT [8]. It offers API functions to read and construct a VMT model, and verifies the model by invoking backend model-checking engines (e.g., nuxmv [36] or IC3ia [37]). In contrast, MoXIchecker is a standalone tool that consumes a MoXI model as input and implements its verification algorithms without calling external model checkers. Moreover, MoXIchecker has the potential for modular verification because MoXI supports constructing bigger systems with smaller subsystems, while VMT only allows flattened transition systems.

Finally, using Python to implement MoXIchecker simplifies the development process thanks to the convenient language features of Python. It makes MoXIchecker ideal for introducing students to model checking. While using Python may limit the efficiency of the tool (especially when a model-checking approach also requires time-consuming operations outside SMT solving), mature acceleration approaches for Python, e.g., writing the time-consuming parts in Cython and compiling into C code, can be applied to mitigate the performance issue.

4 Evaluation

To demonstrate the performance and extensibility of MoXIchecker, we compared it to MoXI-MC-Flow, the translation-based model checker for MoXI [12], which invokes hardware model checkers for Btor2 as backend. Our experiments aim to answer the following research questions:

- RQ1: Is MoXIchecker effective and efficient compared to MoXI-MC-Flow on `QF_BV` and `QF_ABV` tasks?
- RQ2: Can MoXIchecker solve tasks using more complex background theories, which MoXI-MC-Flow cannot solve?

4.1 Experimental Setup

We evaluated MoXIchecker and MoXI-MC-Flow on two sets of MoXI verification tasks in JSON format. The first benchmark set consists of 412 `QF_BV` tasks (247 safe and 165 unsafe) and 41 `QF_ABV` tasks (28 safe and 13 unsafe), all sourced from the MoXI tool suite [12]. Due to the lack of publicly available verification tasks involving more complex theories, we handcrafted 9 tasks using the theories of `QF_LIA`, `QF_LRA`, `QF_NIA`, and `QF_NRA` to test the support for more complex theories of MoXIchecker.

We used MoXIchecker version 0.2 and MoXI-MC-Flow at commit 52f720b1 in the experiments. MoXIchecker called SMT solvers Z3 [38] and MathSAT5 [39] for `QF_BV`, `QF_ABV`, `QF_LIA`, and `QF_LRA` tasks; for tasks using nonlinear arithmetics, MoXIchecker employed Z3. MoXI-MC-Flow invoked Btor2 model checkers AVR [14] and Pono [15] to solve `QF_BV` and `QF_ABV` and tasks. AVR and Pono used Yices2 [40] and Boolector3 [13] as their backend SMT solvers, respectively. `QF_LIA` and `QF_NIA` tasks were also solved by Btor2 model checkers through encoding integers as 32-bit bit-vectors. The version of MoXI-MC-Flow used in

Table 1: Summary of verification results on 453 `QF_BV` and `QF_ABV` tasks

Tool	MoXIchecker				MoXI-MC-Flow	
Backend	MathSAT	MathSAT$^{\text{incr}}$	Z3	Z3$^{\text{incr}}$	AVR	Pono
Correct results	217	**222**	212	217	221	221
`QF_BV`	190	195	195	**200**	193	193
Proofs	54	56	56	**57**	54	56
Alarms	136	139	139	**143**	139	137
`QF_ABV`	27	27	17	17	**28**	**28**
Proofs	15	15	15	15	15	15
Alarms	12	12	2	2	**13**	**13**
Errors and Unknown	236	231	241	236	232	232

our evaluation had no support for reals. Both MoXIchecker and MoXI-MC-Flow used k-induction for verification. (For MoXI-MC-Flow, AVR and Pono were configured to use k-induction on translated Btor2 tasks.)

All experiments were conducted on machines running the GNU/Linux operating system (x86_64-linux, Ubuntu 22.04 with Linux kernel 5.15), each equipped with 33 GB of RAM and a 3.4 GHz Intel Xeon E3-1230 v5 CPU with 8 processing units. Each task was limited to 2 CPU cores, 15 min of CPU time, and 15 GB of RAM. We used BenchExec [41] to ensure reliable resource measurement and reproducible results.

4.2 Experimental Results

RQ1: Performance of MoXIchecker. Table 1 summarizes the experimental results of MoXIchecker and MoXI-MC-Flow on 412 `QF_BV` and 41 `QF_ABV` verification tasks. MoXIchecker, when using MathSAT5 as the backend solver and incremental solving, delivered the most correct results. Notably, MoXIchecker solved 17 tasks that MoXI-MC-Flow failed to translate to Btor2.

Despite being implemented in Python, MoXIchecker demonstrated a comparable performance to MoXI-MC-Flow, which employs highly-optimized hardware model checkers written in C++ as backend. This is mainly because the bottleneck of SMT-based verification algorithms lies in solving SMT formulas. A preliminary run-time profiling for MoXIchecker by cProfile showed that solving formulas accounted for more than 90 % of the run-time for the more time-consuming tasks. The results suggest that using Python to construct and manipulate SMT formulas does not incur much overhead for MoXIchecker.

In our evaluation, MoXIchecker was also more efficient than MoXI-MC-Flow in terms of CPU-time consumption. Figure 3 shows a quantile plot comparing MoXIchecker and MoXI-MC-Flow on the `QF_BV` tasks. A data point (x, y) in the plot indicates that there are x tasks, each of which can be correctly solved by the respective tool within a time bound y seconds. The figure shows that MoXIchecker ran faster than MoXI-MC-Flow, especially for tasks that can be solved quickly, because MoXI-MC-Flow had a slower startup time due to its translation process (note the higher y-intercept of roughly 3 s in Fig. 3).

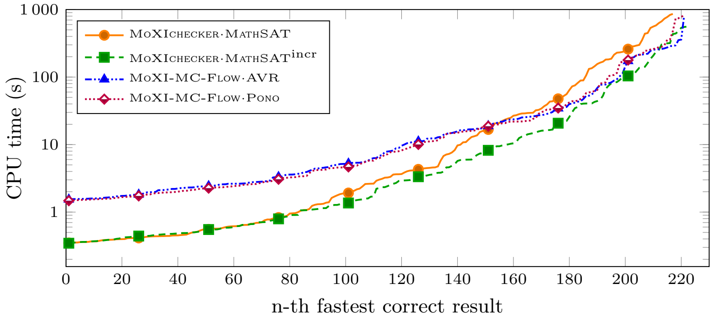

Fig. 3: MoXIchecker vs. MoXI-MC-Flow on 453 `QF_BV` and `QF_ABV` tasks

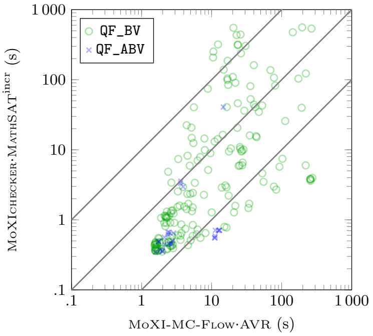

Fig. 4: Efficiency of MoXIchecker vs. MoXI-MC-Flow on `QF_BV` and `QF_ABV` tasks

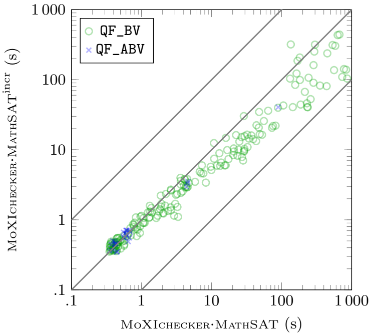

Fig. 5: Effect of incremental SMT solving in MoXIchecker on `QF_BV` and `QF_ABV` tasks

Figure 4 shows a head-to-head comparison of MoXIchecker (cf. ■ in Fig. 3) and MoXI-MC-Flow (cf. ▲ in Fig. 3) in a scatter plot. A data point (x, y) in the plot represents a task that was solved by both MoXI-MC-Flow and MoXIchecker, for which the former took x seconds, while the latter took y seconds. The figure shows that the efficiency of MoXIchecker was competitive against MoXI-MC-Flow. In particular, out of the 205 tasks solved by both, MoXIchecker was faster than MoXI-MC-Flow on 156 tasks.

In addition to the comparison with MoXI-MC-Flow, we evaluated the impact of backend solvers and incremental solving on MoXIchecker. From Table 1, observe that MathSAT5 and Z3 delivered similar performance, with the former being slightly more effective. In contrast, incremental SMT solving had a more pronounced effect on both the effectiveness and efficiency of MoXIchecker. The performance improvement of our k-induction implementation (MoXIchecker·MathSAT$^{\text{incr}}$) over the implementation provided by PySMT (MoXIchecker·MathSAT) is also evident in Fig. 3 and Fig. 5.

RQ2: Extensibility of MoXIchecker. Table 2 lists the results of MoXIchecker and MoXI-MC-Flow on 9 handcrafted model-checking problems involving integer and real arithmetics. MoXIchecker correctly solved all tasks. In contrast, MoXI-MC-Flow produced wrong results or timeouts for the tasks

Table 2: MoXIchecker vs. MoXI-MC-Flow on tasks using integers and reals

Task	Theory	Verdict	MoXIchecker	MoXI-MC-Flow
FibonacciSequence	`QF_LIA`	safe	safe	unsafe
IntIncrement	`QF_LIA`	unsafe	unsafe	safe
IntCounter	`QF_LIA`	safe	safe	timeout
IntMultiply	`QF_NIA`	safe	safe	unsafe
BoundedLinearGrowth	`QF_LRA`	safe	safe	unsupported
DoubleDelay2	`QF_LRA`	unsafe	unsafe	unsupported
OscillatingRatio	`QF_NRA`	safe	safe	unsupported
SafeNonlinearGrowth	`QF_NRA`	safe	safe	unsupported
NonlinearGrowth	`QF_NRA`	unsafe	unsafe	unsupported

containing integers (upper half of Table 2) and had no support for tasks containing reals (lower half of Table 2). Unlike MoXIchecker, which utilized Z3 and thus supported the theories over integers and reals, MoXI-MC-Flow approximated integers with bit-vectors (of length 32 by default). Due to the potential issues of overflow and underflow in bit-vector arithmetics, such approximation is both *unsound* and *incomplete*, therefore causing the incorrect verification results in Table 2. This illustrative experiment shows that, compared to MoXI-MC-Flow, MoXIchecker is (1) more reliable, as it does not yield wrong results due to approximation, and (2) more versatile, as it supports many background theories.

5 Conclusion

We introduced MoXIchecker, the first model checker for MoXI that performs model checking with the SMT formulas describing a MoXI task directly. Compared to MoXI-MC-Flow [12], which translates verification tasks to Btor2 [13] and invokes hardware model checkers, MoXIchecker accommodates MoXI verification tasks with various background theories, facilitates the implementation of new model-checking algorithms, abstracts from specific SMT solvers using the API of PySMT, and has a robust frontend design that avoids potential translation bugs. Currently, MoXIchecker supports the quantifier-free theories of bit-vectors, arrays, integers, and reals, and implements BMC [26], k-induction [27], and IC3/PDR [28] for verification. In our evaluation, MoXIchecker achieved a comparable performance against MoXI-MC-Flow on bit-vector tasks and uniquely solved tasks using integer or real arithmetics. We envision MoXIchecker to facilitate open-source implementations for model-checking techniques around MoXI and become a cornerstone for wider adoption of symbolic model checking. For future work, we will enhance the language support of MoXIchecker, improve the existing verification algorithms and implement new ones, and apply MoXIchecker to software programs or cyber-physical systems. In particular, we want to implement algorithms using Craig interpolation [42] for MoXI. Several interpolation-based algorithms [43, 44, 45] for hardware model checking have been transferred to software verification and demonstrated competitive performance [5, 6].

Data-Availability Statement. The MoXIchecker release 0.2 is available at Zenodo [46] and at https://gitlab.com/sosy-lab/software/moxichecker.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 536040111 (Bridge).

References

1. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Proc. LICS. pp. 428–439. IEEE (1990). https://doi.org/10.1109/LICS.1990.113767
2. McMillan, K.L.: Symbolic Model Checking. Springer (1993). https://doi.org/10.1007/978-1-4615-3190-6
3. Rozier, K.Y., Dureja, R., Irfan, A., Johannsen, C., Nukala, K., Shankar, N., Tinelli, C., Vardi, M.Y.: MoXI: An intermediate language for symbolic model checking. In: Proc. SPIN. LNCS , Springer (2024)
4. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
5. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning (2024). https://doi.org/10.1007/s10817-024-09702-9, preprint: https://doi.org/10.48550/arXiv.2208.05046
6. 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) (2024). https://doi.org/10.1145/3660797
7. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., University of Iowa (2010), https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf
8. Cimatti, A., Griggio, A., Tonetta, S.: The VMT-LIB language and tools. In: Proc. SMT. CEUR Workshop Proceedings, vol. 3185, pp. 80–89. CEUR-WS.org (2022). https://ceur-ws.org/Vol-3185/extended9547.pdf
9. McMillan, K.L.: The SMV system. In: Symbolic Model Checking, pp. 61–85 (1993). https://doi.org/10.1007/978-1-4615-3190-6_4
10. DeLine, R., Leino, R.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005). https://www.microsoft.com/en-us/research/publication/boogiepl-a-typed-procedural-language-for-checking-object-oriented-programs/
11. Beyer, D., Lee, N.Z.: The transformation game: Joining forces for verification. Springer (2024). https://www.sosy-lab.org/research/pub/2024-Katoen60.The_Transformation_Game_Joining_Forces_for_Verification.pdf
12. Johannsen, C., Nukala, K., Dureja, R., Irfan, A., Shankar, N., Tinelli, C., Vardi, M.Y., Rozier, K.Y.: Symbolic model-checking intermediate-language tool suite. In: Proc. CAV. LNCS , Springer (2024)
13. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
14. Goel, A., Sakallah, K.: AVR: Abstractly verifying reachability. In: Proc. TACAS. pp. 413–422. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_23

15. Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.W.: Pono: A flexible and extensible SMT-based model checker. In: Proc. CAV. pp. 461–474. LNCS 12760, Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_22
16. IEEE standard for Verilog hardware description language (2006). https://doi.org/10.1109/IEEESTD.2006.99495
17. Minhas, M., Hasan, O., Saghar, K.: Ver2Smv: A tool for automatic Verilog to SMV translation for verifying digital circuits. In: Proc. ICEET. pp. 1–5 (2018). https://doi.org/10.1109/ICEET1.2018.8338617
18. Irfan, A., Cimatti, A., Griggio, A., Roveri, M., Sebastiani, R.: Verilog2SMV: A tool for word-level verification. In: Proc. DATE. pp. 1156–1159 (2016), https://ieeexplore.ieee.org/document/7459485
19. Mukherjee, R., Tautschnig, M., Kroening, D.: v2c: A Verilog to C translator. In: Proc. TACAS. pp. 580–586. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_38
20. Beyer, D., Chien, P.C., Lee, N.Z.: Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator. In: Proc. TACAS (2). pp. 152–172. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12
21. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Btor2-Cert: A certifying hardware-verification framework using software analyzers. In: Proc. TACAS (3). pp. 129–149. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_7
22. Tafese, J., Garcia-Contreras, I., Gurfinkel, A.: Btor2MLIR: A format and toolchain for hardware verification. In: Proc. FMCAD. pp. 55–63. IEEE (2023). https://doi.org/10.34727/2023/ISBN.978-3-85448-060-0_13
23. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS (3). pp. 365–370. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
24. Griggio, A., Jonáš, M.: Kratos2: An SMT-based model checker for imperative programs. In: Proc. CAV. pp. 423–436. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_20
25. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proc. SMT (2015)
26. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
27. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. FMCAD, pp. 127–144. LNCS 1954, Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
28. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI. pp. 70–87. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
29. Hughes, G.E., Cresswell, M.J.: A New Introduction to Modal Logic. Routledge (1996). https://www.worldcat.org/isbn/978-0-41512-600-7
30. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8
31. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Handbook of Model Checking, pp. 27–73. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_2

32. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Proc. FMCAD. p. 9. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102233
33. PyVMT: A Python library to interact with transition systems. https://github.com/pyvmt/pyvmt, accessed: 2024-10-08
34. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
35. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011). https://doi.org/10.1016/j.cosrev.2010.09.009
36. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Proc. CAV. pp. 334–342. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_22
37. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: IC3 modulo theories via implicit predicate abstraction. In: Proc. TACAS. pp. 46–61. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_4
38. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. LNCS 4963, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
39. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
40. Dutertre, B.: Yices 2.2. In: Proc. CAV. pp. 737–744. LNCS 8559, Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_49
41. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
42. 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
43. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
44. 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
45. Vizel, Y., Grumberg, O., Shoham, S.: Intertwined forward-backward reachability analysis using interpolants. In: Proc. TACAS. pp. 308–323. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_22
46. Ates, S., Beyer, D., Chien, P.C., Lee, N.Z.: MoXIchecker release 0.2. Zenodo (2024). https://doi.org/10.5281/zenodo.13895872

```
[ { "command": "set-logic", "logic": "QF_BV" },
  { "command": "define-system",
    "symbol": "main",
    "input": [],
    "output": [{
        "symbol": "s",
        "sort": { "identifier": { "symbol": "BitVec", "indices": [3] }}}],
    "local": [],
    "init": {
      "identifier": { "symbol": "=", "indices": [] },
      "args": [{ "identifier": "s" }, { "identifier": "#b000" }]},
    "trans": {
      "identifier": { "symbol": "=", "indices": [] },
      "args": [
        { "identifier": "s’" },
        {"identifier": { "symbol": "bvadd", "indices": [] },
          "args": [{ "identifier": "s" }, { "identifier": "#b010" }]}]},
    "inv": { "identifier": "true" }},
  { "command": "check-system",
    "symbol": "main",
    "input": [],
    "output": [{
        "symbol": "s",
        "sort": { "identifier": { "symbol": "BitVec", "indices": [3] }}}],
    "local": [],
    "reachable": [
      { "symbol": "rch_1",
        "formula": { "identifier": { "symbol": "=", "indices": [] },
          "args": [
            {"identifier": { "symbol": "bvurem", "indices": [] },
              "args": [{ "identifier": "s" }, { "identifier": "#b010" }]},
            { "identifier": "#b001" }]}}],
    "query": [{ "symbol": "qry_rch_1", "formulas": ["rch_1"] }]}]
```

Fig. 6: A JSON representation of the MoXI verification task in Fig. 1

A Appendix

Figure 6 shows the corresponding JSON file for the verification task in Fig. 1. For details of the JSON representation, we refer interested readers to the MoXI JSON schema[3] in the MoXI tool suite.

[3] https://github.com/ModelChecker/moxi-mc-flow/tree/main/json-schema