The MoXI Model Exchange Tool Suite*

Chris Johannsen[1], Karthik Nukala[2], Rohit Dureja[3], Ahmed Irfan[2], Natarajan Shankar[2], Cesare Tinelli[4], Moshe Y. Vardi[5], Kristin Yvonne Rozier[1]

[1] Iowa State University {cgjohann, kyrozier}@iastate.edu
[2] SRI International
{karthik.nukala, ahmed.irfan, natarajan.shankar}@sri.com
[3] Advanced Micro Devices, Inc. (rohit.dureja@amd.com)
[4] The University of Iowa (cesare-tinelli@uiowa.edu)
[5] Rice University (vardi@cs.rice.edu)

Abstract. We release the first tool suite implementing MoXI (Model eXchange Interlingua), an intermediate language for symbolic model checking designed to be an international research-community standard and developed by a widespread collaboration under a National Science Foundation (NSF) CISE Community Research Infrastructure initiative. Although we focus here on hardware verification, the MoXI language is useful for software model checking and verification of infinite-state systems in general. MoXI builds on elements of SMT-LIB 2; it is easy to add new theories and operators. Our contributions include: (1) introducing the first tool suite of automated translators into and out of the new model-checking intermediate language; (2) composing an initial example benchmark set enabling the model-checking research community to build future translations; (3) compiling details for utilizing, extending, and improving upon our tool suite, including usage characteristics and initial performance data. Experimental evaluations demonstrate that compiling SMV-language models through MoXI to perform symbolic model checking with the tools from the last Hardware Model Checking Competition performs competitively with model checking directly via nuXmv.

1 Overview

As model checking becomes more integrated into the standard design and verification process for safety-critical systems, the platforms for model-checking research have become more limited (e.g., for the SMV language [47], neither CadenceSMV [46] nor NuSMV [24] are actively maintained; only closed source nuXmv [15] remains). Continuing advances in the field require utilizing higher-level languages that offer sufficient expressive power to describe modern, complex systems and enable validation by industrial system designers. At the same time, contributing advances to back-end model-checking algorithms requires the ability to compare across the full range of state-of-the-art algorithms without regard for which open- or closed-source model checkers implement them or what input languages those tools accept. Comparing new advances in model-checking algorithms to state-of-the-art algorithms requires re-implementing entire model

* This work was funded by NSF:CCRI Awards #2016592, #2016597, and #2016656.

checkers, e.g., [30]. We need a sustainable tool flow that can model the system in the most domain-appropriate high-level modeling language, analyze it with the full range of state-of-the-art model-checking algorithms, and return counterexamples or certificates in the original modeling language.

Our tool suite represents an initial step in unifying model-checking research platforms. We seed an extensible framework designed around a model-checking intermediate language, MoXI (Model eXchange Interlingua). MoXI aims to serve as a common language for the international research community that can connect popular front-end modeling languages with the state of the art in back-end model-checking algorithms. Our vision is that MoXI will enable researchers to model-check a new or extended modeling language simply by writing translators to and from MoXI. Similarly, developing a new backend model-checking algorithm will only require writing a translator to and from MoXI to enable comparisons with existing algorithms and evaluations on every benchmark model, regardless of its original modeling language.

Our initial tool suite accepts models in the higher-level language SMV [47] and efficiently interfaces with the back-end model checkers that competed in the last Hardware Model Checking Competition (HWMCC) [13]. We choose SMV because it is a popular, expressive modeling language successfully used in a wide range of industrial verification efforts [14,17,23,29,30,33,34,36,42,45,48,49,54,61,63–65]. SMV is important because, uniquely from other model-checking input languages, it includes high-level constructs critically required for modeling and validating safety-critical systems, such as many aerospace operational systems from Boeing's Wheel Braking System [14] to NASA's Automated Airspace Concept [34,45,64,65] to a variety of Unmanned Aerial Systems [55,59]. SMV has been used extensively by the hardware model-checking community as well (e.g., at FMCAD [38]) and has appealing qualities that could further the integration of formal methods with the embedded-systems community. Two freely available model checkers, CadenceSMV [46] and NuSMV [24] (which is integrated into today's nuXmv [52]), previously provided viable research platforms. However, today, CadenceSMV's 32-bit pre-compiled binary and nuXmv's closed-source releases are no longer suitable for research, e.g., into improved model-checking algorithms. We provide accessibility to continue the progression of high-level language model checking in SMV via an open-source research platform that allows the use of new algorithms under the hood.

Pushing the state of the art are several open-source, award-winning model-checking tools, including AVR [35], Pono [44], BtorMC [51], and ABC [18]. These tools support a hardware-oriented bit-level input language like Aiger or a bit-precise, word-level format like Btor2. Unfortunately, such languages do not enable the direct modeling of modern complex systems as SMV does, hindering validation efforts. For instance, it is challenging to convince industrial system designers that Aiger models correctly capture their higher-level systems. Perhaps driven by HWMCC, most systems for translating from high-level models to Aiger currently focus on hardware designs, without providing a natural way to describe other computational systems, e.g., embedded systems. Also, the

problem of translating counterexamples produced by low-level model-checking algorithms back into meaningful counterexamples for a non-hardware-centric higher-level language model, such as one in SMV, remains a challenge.

Section 2 provides a basic introduction to MoXI, sufficient to enable understanding of the tool suite functionality; a description of the full language and its semantics appears in [57,58]. Section 3 details the extensible research and verification suite of tools, including translators between the languages SMV, MoXI (in concrete and JSON dialects), and Btor2; utilities for validation; and a full model-checking implementation. Here, we provide a detailed example of behaviorally equivalent models in SMV, MoXI, and Btor2. Our efforts to validate their correctness appear in Section 4. Section 5 demonstrates the efficiency of model checking SMV-language models with a tool portfolio including nuXmv and via translation through MoXI, which performs better than checking with nuXmv alone. The tool[6] and all of the benchmarks[7] used in this experiment are available online for others to utilize in building additional translators to extend our tool suite and the use of MoXI as an intermediate language for symbolic model checking. Section 6 concludes with a discussion of future work.

2 Intermediate Language

MoXI (detailed in [57]) is an intermediate language designed to serve as a common input and output standard for model checkers for finite- and infinite-state systems. It is general enough to encode high-level modeling languages like SMV yet simple enough to enable efficient model checking, including through low-level languages such as Btor2 or SAT/SMT-based engines. Key features include a simple and easily parsable syntax, a rich set of data types, minimal syntactic sugar (at least for now), well-understood formal semantics, and a small but comprehensive set of commands.

MoXI maximizes machine-readability. Therefore, it does not support several human-interface features found in high-level languages such as SMV, TLA+ [43], PROMELA [37], Simulink [27], SCADE [28], and Lustre [19]; nor does it directly support the full features of hardware modeling languages such as VHDL [40], or Verilog [39]. However, many models and queries expressed in these languages can be reduced to MoXI representations. MoXI development was directly informed by previous intermediate formats for formal verification, their successful applications, and their limitations. The eventual form of MoXI stems from a combination of previous work as well as direct conversations with model checking and SMT researchers, including the developers of Aiger [2–4], Btor2 [51], Kind 2 [22], NuSMV [21], nuXmv [16,20], SAL/SALLY [9,32,50], VMT [26,41], and SMT-LIB (the standard I/O language for SMT solvers) [6,7]. MoXI also benefited from the feedback from a technical advisor board of prominent researchers and practitioners in academia and industry [58].

[6] https://github.com/ModelChecker/moxi-mc-flow
[7] https://modelchecker.github.io/benchmarks

MoXI's base logic is the same as that of SMT-LIB Version 2: many-sorted first-order logic with equality, quantifiers, *let* binders, and algebraic datatypes. MoXI extends this logic to (first-order) temporal logic while adopting a discrete and linear notion of time with standard finite and infinite trace-based semantics. MoXI also extends the SMT-LIB language with new commands for defining and verifying multi-component reactive systems. For the latter, it focuses on the specification and checking of reachability conditions (or, indirectly, state and transition invariants) and deadlocks, possibly under fairness conditions on system inputs. Each system definition command defines a transition system by specifying an initial state condition, a transition relation, and system invariants. These are provided as SMT formulas, with minimal syntactic restrictions, for flexibility and future extensibility. Each defined system is parameterized by a state signature, provided as a sequence of typed variables, and can be expressed as the synchronous composition of other systems.[8] The signature partitions state variables into input, output, and local variables. Each system verification command expresses one or more reachability queries over a previously defined system. The queries can be conditional on environmental assumptions on the system's inputs and fairness conditions on its executions. Together with the ability to write observer systems, this allows the expression of arbitrary LTL specifications via standard encodings [56]. Responses to a system verification command can contain (finite or lasso) witness traces for reachable properties or proof certificates for unreachable ones.

Figure 1 contains an example (adapted from [5]) of a three-bit counter and its modular definition in MoXI, together with a *reachability* query and a sample response to the query. Figure 2 contains an extension of that model with an observer system and a query for checking the observational equivalence of the three-bit counter with a bit-vector counter of matching width. The various components of each system definition or check command are provided as attribute-value pairs, following the syntax of SMT-LIB annotations. Transition predicates use primed variables to denote next-state values.

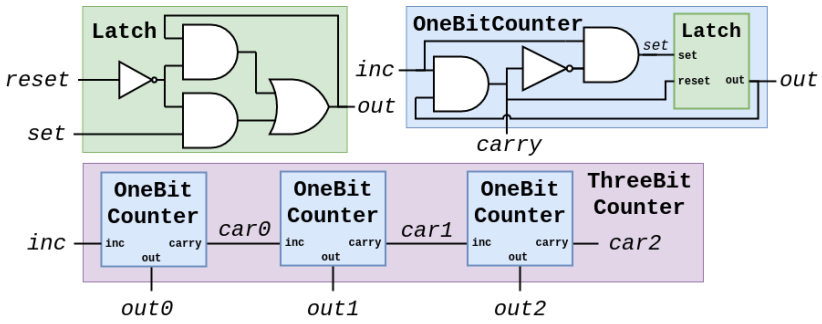
3 Tool Suite

We provide a suite of tools for translating into and out of MoXI and validating MoXI scripts. The tools are implemented in type-annotated Python with a focus on finite-state systems (for now). Figure 3 illustrates the end-to-end toolchain for model checking using MoXI, including relationships between the various tools.

3.1 Translators

The tool suite provides four translators that take as input a model, query, or witness specified in a source language and output a *behaviorally equivalent* model, query, or witness in the configured target language.

[8] We plan to include asynchronous composition in a later release.




```
1  (define-system Latch :input ((set Bool) (reset Bool))
2    :output ((out Bool))
3    :init (not out)
4    :trans ((= out' (or (and set (not reset))
5                        (and (not reset) out))))
6  )
7  (define-system OneBitCounter :input ((inc Bool))
8    :output ((out Bool) (carry Bool)) :local ((set Bool))
9    :subsys (L (Latch set carry out))
10   :inv (and (= set (and inc (not carry)))
11             (= carry (and inc out)))
12 )
13 (define-system ThreeBitCounter :input ((inc Bool))
14   :output ((out0 Bool) (out1 Bool) (out2 Bool))
15   :local ((car0 Bool) (car1 Bool) (car2 Bool))
16   :init (and (not out0) (not out1) (not out2))
17   :subsys (C1 (OneBitCounter inc out0 car0))
18   :subsys (C2 (OneBitCounter car0 out1 car1))
19   :subsys (C3 (OneBitCounter car1 out2 car2))
20 )
21 (check-system ThreeBitCounter :input ((inc Bool))
22   :output ((out0 Bool) (out1 Bool) (out2 Bool))
23   :local ((car0 Bool) (car1 Bool) (car2 Bool))
24   :reachable (r (and (not out0) out1 (not out2)))
25   :query (query1 (r))
26 )
```

```
1  (check-system-response ThreeBitCounter
2    :query (query1 :result sat :trace query1_trace)
3    :trace (query1_trace :prefix query1_trail)
4    :trail (query1_trail
5      (0 (out0 0) (out1 0) (out2 0) (inc 1) (car0 0) (car1 0) (car2 0))
6      (1 (out0 1) (out1 0) (out2 0) (inc 1) (car0 1) (car1 0) (car2 0))
7      (2 (out0 0) (out1 1) (out2 0) (inc 1) (car0 0) (car1 0) (car2 0))
8  ))
```

Fig. 1: (Top) The three-bit counter circuit composes three one-bit counters together, where each counter uses a latch to store that counter's current value. (Middle) A MoXI implementation of the circuit uses `define-system` (lines 1-20) to describe and compose each counter component. It then queries (lines 21-26) whether the counter can output 2. (Bottom) A possible query response provides a trace showing that the counter outputs 2 within 3 execution steps. We write `Bool` values as integers here for compactness.

```
(set-logic QF_BV)
(define-system BitVecCounter :input ((inc Bool))
  :output ((out (_ BitVec 3)))
  :init (= out #b000)
  :trans (= out' (ite inc (bvadd out #b001) out))
)
(define-system Monitor
  :local ((inc Bool) (out_bit (_ BitVec 3)) (out_bv (_ BitVec 3))
    (bit0 Bool) (bit1 Bool) (bit2 Bool))
  :subsys (C1 (ThreeBitCounter inc bit0 bit1 bit2 ))
  :subsys (C2 (BitVecCounter inc out_bv ))
  :inv (= out_bit (to_bv3 bit0 bit1 bit2))
)
(check-system Monitor
  :local ((inc Bool) (out_bit (_ BitVec 3)) (out_bv (_ BitVec 3))
    (bit0 Bool) (bit1 Bool) (bit2 Bool))
  :reachable (reach1 (distinct out_bit out_bv))
  :query (query1 (reach1))
)
```

```
(check-system-response Monitor
  :query (query1 :result unsat)
)
```

Fig. 2: (Top) Extending the MoXI model shown in Figure 1, a `Monitor` (lines 33-40) computes the output for a `ThreeBitCounter` and a bit-vector-based counter (lines 28-32). The function `to_bv3` (definition is omitted for space constraints) converts bit values to the corresponding bit-vector value. The `check-system` command (lines 40-45) queries whether their outputs can possibly differ. (Bottom) The `check-system-response` reports an unsatisfiable query, proving the two counters equivalent.

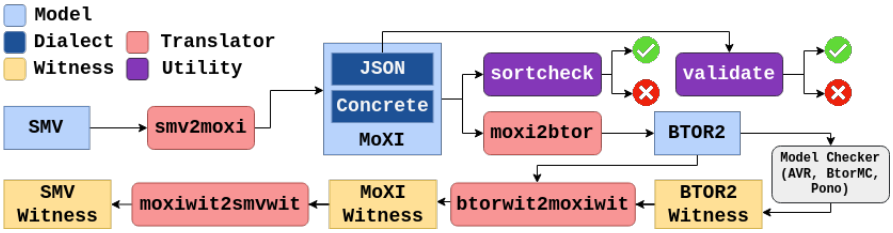


Fig. 3: Starting with a nuXmv model, `smv2moxi` generates a behaviorally equivalent MoXI model in either the MoXI concrete syntax or a JSON dialect syntax. `moxi2btor` translates this MoXI model to a set of Btor2 models, one for each query, which an off-the-shelf model checker (e.g., AVR [35], Pono [44], BtorMC [51]) solves. Then, `btorwit2moxiwit` creates a MoXI witness from the Btor2 witness using the Btor2 model to map variable names properly, and similarly for `moxiwit2smvwit`. The sort checker validates MoXI input against any of the SMT-LIB logics listed in Section 3.2. The validator checks JSON dialect input against our provided schema.

(1) `smv2moxi` translates specifications written in (a common subset of) the SMV language into MoXI. Broadly, this tool supports Finite State Machine (FSM) definitions (nuXmv manual, Section 2.3 [16]). It currently supports only statically typed expressions; for example, all module instantiations of the same

defined module must share the same signature. (For a module `M` with parameters `p1` and `p2`, the types of `p1`, `p2` must be the same across all instantiations of `M`.) Figure 4 shows that the translation preserves the hierarchy between the SMV modules and submodule instantiations.

The MoXI encoding captures SMV macro and function declarations (`DEFINE`, `FUN`), variable declarations (`VAR`, `IVAR`, `FROZENVAR`), state machine declarations (`INIT`, `TRANS`, `INVAR`, `ASSIGN`), invariant specifications (`AG` [property], `INVARSPEC`) and fairness constraints (`FAIRNESS`, `JUSTICE`, `COMPASSION`). To support LTL specifications (`LTLSPEC`), `smv2moxi` runs PANDA [56], an open-source tool offering a portfolio of LTL-to-symbolic automaton translations in SMV format.

The `smv2moxi` tool consists of (1) preprocessing that renames identifiers deviating from the SMV grammar (discussed in Section 4); (2) running the C preprocessor (SMV supports C-style macros) and PANDA [56] (for LTL specifications); (3) parsing via a SLY-generated [8] parser; (4) running an SMV type checker; (5) translating to MoXI. We emphasize that tool guarantees apply to *well-formed* SMV models as determined by nuXmv.

(2) `moxi2btor` translates MoXI to Btor2 by creating a Btor2 file for each `:query` attribute in each `check-system` command. Some crucial differences between MoXI and Btor2 present non-trivial challenges. Firstly, Btor2 does not support hierarchical models. `moxi2btor` flattens the system hierarchy in its translation as a result. Secondly, MoXI allows for declarative-style initial, transition, and invariant conditions while Btor2 allows only assignment-style. Figure 4 shows how `moxi2btor` encodes each system's conditions using three variants of each variable. Thirdly, a MoXI query with multiple reachability properties asks for a trace that eventually satisfies each property. In Btor2, multiple `bad` properties in a file ask for a trace that eventually satisfies at least one such property. Figure 4 again shows how the translation resolves this difference. The `moxi2btor` tool's workflow consists of (1) parsing via a SLY-generated parser [8]; (2) running `sortcheck` (Section 3.2); (3) translating to a set of Btor2 files, each behaviorally equivalent to its corresponding `:query`.

(3) `btorwit2moxiwit` translates Btor2 witnesses to MoXI witnesses using the `check-system-response` syntax. It assumes `moxi2btor` created the Btor2 input files used to generate the witness and uses information that `moxi2btor` encodes in the comments of each Btor2 file, e.g., to map bit vectors to enumeration values for variables of such sorts.

(4) `moxiwit2smvwit` translates MoXI witnesses to SMV-language witnesses.

3.2 Utilities

`sortcheck` We provide a sort-checker for MoXI that supports the following SMT-LIB logics: `QF_BV`, `QF_ABV`, `QF_LIA`, `QF_NIA`, `QF_LRA`, and `QF_NRA`.

`validate` We define a JSON Schema for MoXI and support a JSON dialect for MoXI in our tools. Given the evolving nature of new languages and their standards, tool writers often pay an unnecessary overhead keeping front-end tools

SMV

```
MODULE Delay(i,o)
INIT
  (o = 0ud8_0);
TRANS
  (next(o) = i);

MODULE main
IVAR
i: unsigned
   word[8];
VAR
o: unsigned
   word[8];
D: Delay(i,o);


INVARSPEC
! (o = 0ud8_2);
```

MoXI

```
(set-logic QF_BV)
(define-system Delay
  :input ((i (_ BitVec 8))
          (o (_ BitVec 8)))
  :init (= o #x00)
  :trans (= o' i)
)
(define-system main
  :input ((i (_ BitVec 8)))
  :output ((o (_ BitVec 8)))
  :local ((D.i (_ BitVec 8))
          (D.o (_ BitVec 8)))
  :inv (and
    (= D.i i) (= D.o o))
  :subsys
    (D (Delay D.i D.o))
)
(check-system main
  :input ((i (_ BitVec 8)))
  :output ((o (_ BitVec 8)))
  :local ((D.i (_ BitVec 8))
          (D.o (_ BitVec 8)))
  :reachable (rch
    (not (not (= o #x02))))
  :query (qry_rch (rch))
)
```

BTOR2

```
1  sort bitvec 8
2  sort bitvec 1
3  state 1 D.o.init
4  state 1 D.o.cur
5  state 1 D.o.next
6  state 1 D.i.cur
7  state 1 o.cur
8  state 1 i.cur
9  init 1 4 3
10 next 1 4 5
11 constd 1 0
12 eq 2 3 11
13 constraint 12
14 eq 2 5 6
15 constraint 14
16 constd 2 1
17 constraint 16
18 eq 2 4 7
19 eq 2 6 8
20 and 2 19 18
21 constraint 20
22 constd 2 0
23 constd 1 2
24 eq 2 7 23
25 not 2 24
26 not 2 25
27 state 2 F_rch
28 init 2 27 22
29 ite 2 27 16 26
30 next 2 27 29
31 bad 27
```

Fig. 4: The toolchain translates the SMV model for a delay circuit on the left to the MoXI model in the center by creating a `define-system` command for each `MODULE`. It then generates the Btor2 model on the right, introducing three variants of each `check-system` variable (`.init`, `.cur`, `.next`) and setting constraints such as the `:init` and `:next` of `Delay` on lines 13 and 15 respectively. The Btor2 "flag" variable `F_rch` (line 27) encodes if formula `rch` has been true at least once during the execution; the presence of multiple Btor2 `bad` properties asks for a trace where at least one such property is eventually true, we conjunct the *flag* variables to ask for a trace where every property is eventually true.

up to date. By supporting the representation of MoXI constructs in the JSON dialect, we expect to facilitate tool development, improve tool interoperability, and ensure conformance to the language standard. Tool writers can use off-the-shelf JSON parsers (e.g., simdjson, RapidJSON) to obtain industrial-strength MoXI parsers in the language they choose "for free." We plan to include a JSON schema for each MoXI release, enabling seamless front-end compatibility with the latest MoXI standard along with language/platform independence. The `validate` utility invokes a JSON validator from Python's `jsonschema` package to validate a MoXI script (in the JSON dialect) against the MoXI JSON schema.

4 Tool Suite Validation

We validate our tools using a combination of manual inspection, sort checking of translated output, and comparing witnesses between those generated by

SMV

```
Trace Description:
  nuxmv2btor
  counterexample
Trace Type:
  Counterexample
  -> State: 1.1 <-
    D.i = 0ud8_2
    D.o = 0ud8_0
    o = 0ud8_0
  -> Input: 1.2 <-
    i = 0ud8_0
  -> State: 1.2 <-
    D.i = 0ud8_0
    D.o = 0ud8_2
    o = 0ud8_2
```

MoXI

```
(check-system-response main
  :query (qry_rch
    :result sat
    :trace qry_rch_trace
  )
  :trace (qry_rch_trace
    :prefix qry_rch_trail
  )
  :trail (qry_rch_trail
    (0 (D.i #b00000010)
       (D.o #b00000000)
       (o #b00000000)
       (i #b00000010))
    (1 (D.i #b00000000)
       (D.o #b00000010)
       (o #b00000010)
       (i #b00000000))
))
```

BTOR2

```
sat ____ b0 ____ #0
0 00000000 D.o.init
1 00000000 D.o.cur
2 00000010 D.o.next
3 00000010 D.i.cur
4 00000000 o.cur
5 00000010 i.cur
6 0 F_rch
#1
1 00000010 D.o.cur
2 00000000 D.o.next
3 00000000 D.i.cur
4 00000010 o.cur
5 00000000 i.cur
#2
1 00000000 D.o.cur
4 00000000 o.cur
6 1 F_rch
```

Fig. 5: The witness translation after model checking the BTOR2 file in Figure 4 works right to left: it maps each BTOR2 `.cur` variable to its MoXI counterpart and discards the last frame of the witness due to the delay caused by using *flag* variables. Similarly, it maps each MoXI variable to its SMV counterpart.

nuXmv and our end-to-end tool suite. We use catbtor [51] for sort checking and BtorMC, AVR, and Pono for bounded model checking (BMC) of Btor2 files. For benchmark generation, we use the set of nuXmv input files provided in the most recent release of nuXmv.

Manual Inspection. We provide an initial set of hand-written MoXI benchmarks to perform manual validation. Each benchmark is well-sorted according to `sortcheck`, generates well-sorted Btor2 via `moxi2btor` according to catbtor, and generates correct, manually-inspected witnesses via BtorMC and `btorwit2moxiwit`.[9]

Sort Checked Translations. Using the benchmarks distributed with nuXmv as input, we check that the output of `smv2moxi` and `moxi2btor` are well-sorted according to `sortcheck` and catbtor. We discovered discrepancies in benchmarks distributed with nuXmv while developing these utilities, where the benchmarks did not conform to the grammar defined in Chapter 2 of the nuXmv User Manual [16] but were accepted by nuXmv nonetheless, particularly concerning identifiers. The preprocessor of `smv2moxi` transforms these identifiers into valid ones. There were also numerous ill-typed benchmarks that `smv2moxi`'s type checker correctly rejects.

Output Comparison. Using the nuXmv benchmarks again as input, we run nuXmv and our tool suite to generate witnesses for each specification. Both nuXmv and our tool suite agree on the result of every model-checking query. Section 5 describes how our toolchain (using BtorMC, AVR, or Pono as its back end) shows a similar number of timeouts compared with nuXmv when the latter is set to use BMC or k-induction.

[9] Many thanks to Daniel Larraz for writing many of the MoXI examples.

5 Benchmarks

We provide an initial set of MoXI benchmarks for the model-checking community generated from the set of SMV input files provided in the most recent release of nuXmv. Noting that many of the SMV benchmarks are results of a Btor2 to nuXmv translation themselves, we stress that this set of benchmarks is intended to be an *initial* set. We expect to achieve greater benchmark diversity with continued toolchain development and increased adoption of MoXI by other researchers.

Experimental Evaluation. We compare the end-to-end performance of model-checking SMV-language models with a portfolio comprising nuXmv and Btor2 model checkers: AVR, Pono, and BtorMC, on a set of 960 `QF_ABV`-compatible SMV benchmarks, i.e., SMV models with boolean, word or array types. We use the HWMCC 2020 versions of AVR and Pono, the version of BtorMC from the latest version of Boolector [51], and the latest public release of nuXmv (version 2.0.0). Each checker is configured with a 1-hour time limit and 8GB memory limit and runs BMC [12] and k-induction [60] with a max bound of 1000. (We do not run BtorMC with k-induction due to a bug in its implementation.)

Figure 6 shows our evaluation, with portfolio performance depicted as *virtual-best (vb)*. While we consider this a proof-of-concept evaluation, we observe that SMV-language model checking using Btor2 model checkers, enabled via a translation through MoXI, delivers superior performance on unsafe queries compared to model checking with nuXmv alone: vb-bmc solves 57% more benchmarks than nuXmv-bmc while ensuring all Btor2 witnesses are correctly translated to SMV traces. We measure competitive performance with vb-kind solving 6% more benchmarks than nuXmv-kind for safe queries. The vb performance gains are due to its ability to use a variety of model checkers with different SMT solver backends of varying strengths, e.g., nuXmv uses MathSAT [25], AVR uses Yices [31], and Pono uses Boolector [51], while ensuring correct model and witness translation through MoXI. Section 4 of Rozier et al. [57] includes experimental data using each tool's IC3-based algorithms.

6 Conclusion and Future Work

The presented tool suite provides the foundational step in developing an open-source, state-of-the-art symbolic model-checking framework for the research community. It constitutes the first tool support for the new intermediate language MoXI, the first experimental evidence of the potential for efficient translation through MoXI, and a basis upon which the hardware and software model-checking communities can build. Adding support for checking models in a high-level modeling language is now as easy as adding a translator between that language and MoXI to this tool suite. Similarly, experimenting with a novel back-end model-checking algorithm to check all supported input modeling languages only requires writing a new MoXI translator interfacing with that algo-

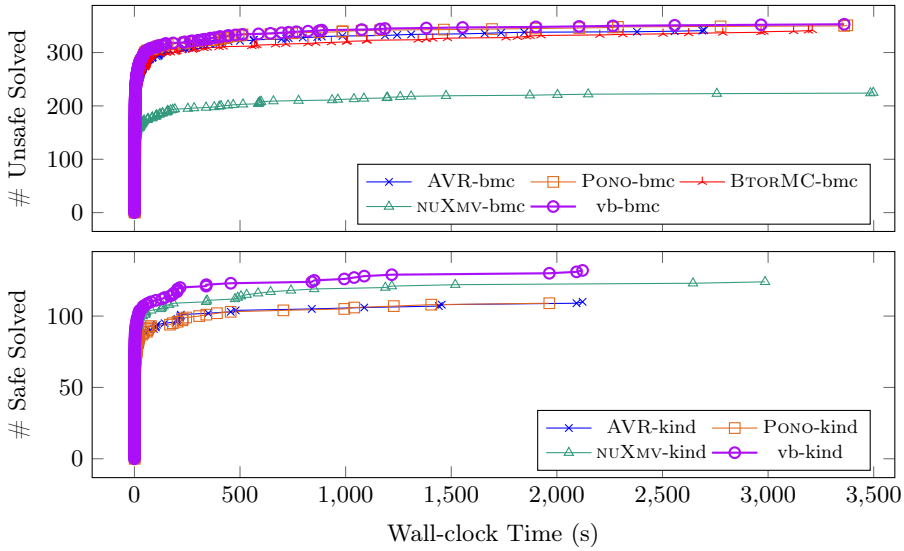


Fig. 6: Performance comparison on unsafe and safe queries with BMC and k-induction across different model checkers. vb-* represents the virtual best solver. Wall-clock time for the non-nuXmv plots includes translation time.

rithm. Benchmarking against other model-checking algorithms no longer require re-implementing existing tools to achieve an apples-to-apples comparison.

Connecting this toolchain to existing tools enables the immediate application of verification techniques for Btor2 to MoXI beyond just hardware model checkers. For example, a software model checker can verify a MoXI model via Btor2C [11], making at least 59 other backend verifiers for MoXI available [10].

This release enables future instantiations of HWMCC [13] to add competition tracks centered around MoXI, with extensions from the model-checking research community. Specifying, proving correct, and extracting efficient C code for our translation using a theorem prover such as PVS [53] would provide an additional trusted translation between languages beyond the validation techniques in Section 4. We are writing a back end to Yosys [62], the open-source RTL synthesis framework, to generate files directly from Verilog designs and facilitate a more extensive set of realistic benchmarks to add to the initial set in Section 5. Additionally, once MoXI certificates are fully defined, we can translate Btor2-Cert [1] certificates back to MoXI from Btor2-Cert-supported verifiers. Finally, we expect developers of model checkers for higher-level modeling languages than a language like Btor2 may choose to support MoXI directly. We have work in this direction underway for the Kind 2 checker [22].

References

1. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Btor2-cert: A certifying hardware-verification framework using software analyzers. In: International

Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 129–149. Springer (2024)
2. The AIGER and-inverter graph (AIG) format version 20071012. http://fmv.jku.at/aiger/FORMAT, accessed: 2016-07-25
3. AIGER 1.9 and beyond. http://fmv.jku.at/hwmcc11/beyond1.pdf, accessed: 2016-07-25
4. AIGER website. http://fmv.jku.at/aiger/, accessed: 2016-07-25
5. Alur, R.: Principles of cyber-physical systems. MIT press (2015)
6. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB), https://smt-lib.org
7. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (2010)
8. Beazley, D.: Sly (sly lex yacc). https://sly.readthedocs.io/en/latest/ (2018)
9. Bensalem, S., Ganesh, V., Lakhnech, Y., noz, C.M., Owre, S., Rueß, H., Rushby, J., Rusu, V., Saïdi, H., Shankar, N., Singerman, E., Tiwari, A.: An overview of SAL. In: Holloway, C.M. (ed.) LFM 2000: Fifth NASA Langley Formal Methods Workshop. pp. 187–196. NASA Langley Research Center, Hampton, VA (Jun 2000), http://www.csl.sri.com/papers/lfm2000/
10. Beyer, D.: State of the art in software verification and witness validation: Sv-comp 2024. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 299–329. Springer (2024)
11. Beyer, D., Chien, P.C., Lee, N.Z.: Bridging hardware and software analysis with btor2c: A word-level-circuit-to-c translator. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 152–172. Springer (2023)
12. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking Without BDDs. In: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems. pp. 193–207. TACAS, Springer-Verlag, Berlin, Heidelberg (1999), http://dl.acm.org/citation.cfm?id=646483.691738
13. Biere, A., Froleyks, N., Preiner, M.: Hardware Model Checking Competition (HWMCC). https://fmv.jku.at/hwmcc20/index.html (2020)
14. Bozzano, M., Cimatti, A., Fernandes Pires, A., Jones, D., Kimberly, G., Petri, T., Robinson, R., Tonetta, S.: Formal design and safety analysis of AIR6110 wheel brake system. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV. pp. 518–535. Springer (2015)
15. Bozzano, M., Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: nuXmv 1.0 User Manual. Tech. rep., FBK - Via Sommarive 18, 38055 Povo (Trento) – Italy (2014)
16. Bozzano, M., Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: nuxmv 2.0. 0 user manual. Fondazione Bruno Kessler, Tech. Rept., Trento, Italy (2019)
17. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: The COMPASS approach: Correctness, Modelling, and Performability of Aerospace Systems. In: Computer Safety, Reliability, and Security, pp. 173–186. Springer (2009)
18. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: International Conference on Computer Aided Verification. pp. 24–40. Springer (2010)

19. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: A declarative language for programming synchronous systems. In: Proc. 14tj Annual ACM Symposium on Principles of Programming Languages. pp. 178–188 (1987)
20. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) Proc. 26th Int. Conf. on Computer Aided Verification. Lecture Notes in Computer Science, vol. 8559, pp. 334–342. Springer (2014)
21. Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: Nusmv 2.6 user manual (2016)
22. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Proc. 28th Int'l Conf. on Computer Aided Verification. Lecture Notes in Computer Science, vol. 9780, pp. 510–517. Springer (2016)
23. Choi, Y., Heimdahl, M.: Model checking software requirement specifications using domain reduction abstraction. In: IEEE ASE. pp. 314–317 (2003)
24. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Nusmv 2: An opensource tool for symbolic model checking. In: CAV, Proc. 14th Int'l Conf. pp. 359–364. LNCS 2404, Springer (2002)
25. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS. pp. 93–107 (2013)
26. Cimatti, A., Griggio, A., Tonetta, S., et al.: The vmt-lib language and tools. In: Proceedings of the 20th Internal Workshop on Satisfiability ModuloTheories co-located with the 11th International Joint Conference onAutomated Reasoning {(IJCAR} 2022) part of the 8th Federated LogicConference (FLoC 2022), Haifa, Israel, August 11-12, 2022. vol. 3185, pp. 80–89. CEUR-WS. org (2022)
27. Documentation, S.: Simulation and model-based design (2020), https://www.mathworks.com/products/simulink.html
28. Documentation, SCADE: Ansys SCADE Suite (2023), https://www.ansys.com/products/embedded-software/ansys-scade-suite
29. Dureja, R., Rozier, E.W.D., Rozier, K.Y.: A case study in safety, security, and availability of wireless-enabled aircraft communication networks. In: Proceedings of te 17th AIAA Aviation Technology, Integration, and Operations Conference (AVIATION). American Institute of Aeronautics and Astronautics (June 2017). https://doi.org/http://dx.doi.org/10.2514/6.2017-3112
30. Dureja, R., Rozier, K.Y.: FuseIC3: An algorithm for checking large design spaces. In: Proceedings of Formal Methods in Computer-Aided Design (FMCAD). IEEE/ACM, Vienna, Austria (October 2017)
31. Dutertre, B.: Yices 2.2. In: International Conference on Computer Aided Verification. pp. 737–744. Springer (2014)
32. Dutertre, B., Jovanović, D., Navas, J.A.: Verification of fault-tolerant protocols with sally. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NASA Formal Methods. pp. 113–120. Springer International Publishing (2018)
33. Gan, X., Dubrovin, J., Heljanko, K.: A symbolic model checking approach to verifying satellite onboard software. Science of Computer Programming (2013) (March 2013), http://dx.doi.org/10.1016/j.scico.2013.03.005
34. Gario, M., Cimatti, A., Mattarei, C., Tonetta, S., Rozier, K.Y.: Model checking at scale: Automated air traffic control design space exploration. In: Proceedings of 28th International Conference on Computer Aided Verification (CAV 2016). LNCS, vol. 9780, pp. 3–22. Springer, Toronto, ON, Canada (July 2016). https://doi.org/10.1007/978-3-319-41540-6_1

35. Goel, A., Sakallah, K.: Avr: abstractly verifying reachability. In: Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part I 26. pp. 413–422. Springer (2020)
36. Gribaudo, M., Horvath, A., Bobbio, A., Tronci, E., Ciancamerla, E., Minichino, M.: Model-checking based on fluid Petri nets for the temperature control system of the ICARO co-generative plant. Tech. rep., SAFECOMP, 2434, LNCS (2002)
37. Holzmann, G.: Design and Validation of Computer Protocols. Prentice-Hall Int. Editions (1991)
38. Hunt, W.: FMCAD organization home page. http://www.cs.utexas.edu/users/hunt/FMCAD/
39. IEEE: IEEE standard for Verilog hardware description language (2005)
40. IEEE: IEEE standard for VHDL language reference manual (2019)
41. Kessler, F.B.: Verification modulo theories. https://vmt-lib.fbk.eu/, accessed: 2017-09-30
42. Lahtinen, J., Valkonen, J., Björkman, K., Frits, J., Niemelä, I., Heljanko, K.: Model checking of safety-critical software in the nuclear engineering domain. Reliability Engineering & System Safety **105**(0), 104–113 (2012), http://www.sciencedirect.com/science/article/pii/S0951832012000555
43. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
44. Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.: Pono: a flexible and extensible SMT-based model checker. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33. pp. 461–474. Springer (2021)
45. Mattarei, C., Cimatti, A., Gario, M., Tonetta, S., Rozier, K.Y.: Comparing different functional allocations in automated air traffic control design. In: Proceedings of Formal Methods in Computer-Aided Design (FMCAD 2015). IEEE/ACM, Austin, Texas, U.S.A (September 2015)
46. McMillan, K.: The SMV language. Tech. rep., Cadence Berkeley Lab (1999)
47. McMillan, K.L.: Symbolic Model Checking, chap. The SMV System, pp. 61–85. Springer US, Boston, MA (1993), https://doi.org/10.1007/978-1-4615-3190-6_4
48. Miller, S.: Will this be formal? In: TPHOLs 5170, pp. 6–11. Springer (2008), http://dx.doi.org/10.1007/978-3-540-71067-7_2
49. Miller, S.P., Tribble, A.C., Whalen, M.W., Per, M., Heimdahl, E.: Proving the shalls. STTT **8**(4-5), 303–319 (2006)
50. de Moura, L., Owre, S., Shankar, N.: The SAL language manual. CSL Technical Report SRI-CSL-01-02 (Rev. 2), SRI Int'l, 333 Ravenswood Ave., Menlo Park, CA 94025 (Aug 2003)
51. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. 30th Int. Conf. on Computer Aided Verification. Lecture Notes in Computer Science, vol. 10981, pp. 587–595. Springer (2018)
52. The nuXmv model checker; available at https://nuxmv.fbk.eu/, 2015
53. Owre, S., Rushby, J., Shankar, N.: Pvs: A prototype verification system. In: Proc. 11th Int'l Conf. on Automated Deduction. Lecture Notes in Computer Science, vol. 607, pp. 748–752. Springer (1992)
54. Raimondi, F., Lomuscio, A., Sergot, M.J.: Towards model checking interpreted systems. In: FAABS 02, LNAI 2699. pp. 115–125. Springer (2002)

55. Reinbacher, T., Rozier, K.Y., Schumann, J.: Temporal-logic based runtime observer pairs for system health management of real-time systems. In: Proceedings of the 20th International Conference on Tools and Algorithms f or the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science (LNCS), vol. 8413, pp. 357–372. Springer-Verlag (April 2014)
56. Rozier, K.Y., Vardi, M.Y.: A multi-encoding approach for LTL symbolic satisfiability checking. In: 17th International Symposium on Formal Methods (FM2011). Lecture Notes in Computer Science (LNCS), vol. 6664, pp. 417–431. Springer-Verlag (2011)
57. 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: Proceedings of the 30th International Symposium on Model Checking Software (SPIN). LNCS, Springer (April 2024)
58. Rozier, K.Y., Shankar, N., Tinelli, C., Vardi, M.Y.: Developing an open-source, state-of-the-art symbolic model-checking framework for the model-checking research community. Online: https://modelchecker.github.io (2019)
59. Schumann, J., Rozier, K.Y., Reinbacher, T., Mengshoel, O.J., Mbaya, T., Ippolito, C.: Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. In: Proceedings of the 2013 Annual Conference of the Prognostics and Health Ma nagement Society (PHM2013). pp. 381–401 (October 2013)
60. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. 3rd Int'l Conf. on Formal Methods in Computer-Aided Design. Lecture Notes in Computer Science, vol. 1954, pp. 108–125. Springer (2000)
61. Tribble, A., Miller, S.: Software safety analysis of a flight management system vertical navigation function-a status report. In: DASC. pp. 1.B.1–1.1–9 v1 (2003)
62. Wolf, C.: Yosys open synthesis suite (2016)
63. Yoo, J., Jee, E., Cha, S.: Formal modeling and verification of safety-critical software. Software, IEEE **26**(3), 42–49 (2009)
64. Zhao, Y., Rozier, K.Y.: Formal specification and verification of a coordination protocol for an automated air traffic control system. In: Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012). Electronic Communications of the EASST, vol. 53, pp. 337–353. European Association of Software Science and Technology (2012)
65. Zhao, Y., Rozier, K.Y.: Formal specification and verification of a coordination protocol for an automated air traffic control system. Science of Computer Programming Journal **96**(3), 337–353 (December 2014)