Symbolic Model-Checking Intermediate-Language 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 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, CadenceSMV[42] and NuSMV[21] are both deprecated; only closed source nuXmv [12] remains). Continuing advances in the field requires the ability to utilize higher-level languages that offers 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 the state-of-the-art algorithms, without regard for which open- or closed-source model checkers implement them, or what input languages those tools accept. Currently, comparing new advances in model-checking algorithms to the state of the art requires re-implementing entire model checkers, e.g., [27]. 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.

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

Our tool suite represents an initial step at 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 back-end model-checking algorithm will only require writing a translator to and from MoXI to enable comparisons with existing algorithms, and evaluations on every available benchmark model, regardless of its original modeling language.

Our initial tool suite accepts models in the higher-level language SMV, and efficiently interfaces with the back-end model checkers that competed in the last Hardware Model Checking Competition (HWMCC) [10]. We choose SMV because it is a popular, expressive modeling language, successfully used in a wide range of industrial verification efforts [49,32,57,20,44,43,59,14,29,38,11,60,61,41,30,26,27]. 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 [11] to NASA's Automated Airspace Concept [60,61,41,30] to a variety of Unmanned Aerial Systems [54,50]. SMV has been used extensively by the hardware model-checking community as well (e.g., at FMCAD [34]) and has appealing qualities that could further the integration of formal methods with the embedded-systems community. Two freely-available model checkers, CadenceSMV [42] and NuSMV [21] (which is integrated into today's nuXmv [47]), previously provided viable research platforms. Yet, today CadenceSMV's 32-bit pre-compiled binary and nuXmv's increasingly restricted, 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 [31], Pono [40], BtorMC [46], and ABC [15], These tools are based on a hardware-oriented bit-level input language like Aiger, or a bit-precise, word-level format like Btor2. Unfortunately, such languages do not support direct modeling of modern complex systems the way SMV does. This hinders validation as it is very hard, for instance, 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 [53,52]. Section 3 details the extensible research and verification suite of tools, including translators between the languages SMV, MoXI (in concrete and JSON di-

alects), 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 via translation through MOXI, which performs better than checking with NUXMV alone. All of the benchmarks used in this experiment are available online[6] 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 [52]), 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 initially); well-understood formal semantics; a small but comprehensive set of commands.

Since MOXI maximizes machine-readability, it does not support all human-interface features found in high-level languages such as SMV, TLA+ [39], PROMELA [33], Simulink [24], SCADE [25], and Lustre [16]; nor does it directly support the full features of hardware modeling languages such as VHDL [36], or Verilog [35]. However, 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 and direct conversations with model checking and SMT researchers, including the developers of BTOR2 [46], SMV [18,17], NUXMV [13,17], AIGER [1,2,3], SAL/SALLY [45,8], VMT-FBK [23,37], Kind 2 [19], and SMT-LIB (the standard I/O language for SMT solvers) [56,6,5]. MOXI also benefited from the feedback from a technical advisor board made of prominent researchers and practitioners in academia and industry [53].

The base logic of MOXI 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 via the specification of an initial state condition, a transition relation and system invariants. These are provided as SMT formulas, with minimal syntactic restrictions on them, for flexibility and future extensibility. Each defined system is parameterized by a state signature, provided as a sequence of typed variables,

[6] `modelchecker.github.io/benchmarks`

and can be expressed as the synchronous composition of other systems.[7] 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 [51]. 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 [4]) 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 Tools

We provide a suite of tools for translating into and out of and validating MOXI scripts implemented in type-annotated Python and focusing on finite-state systems for now. Figure 3 illustrates the end-to-end toolchain, with relationships between the tools.

3.1 Translators

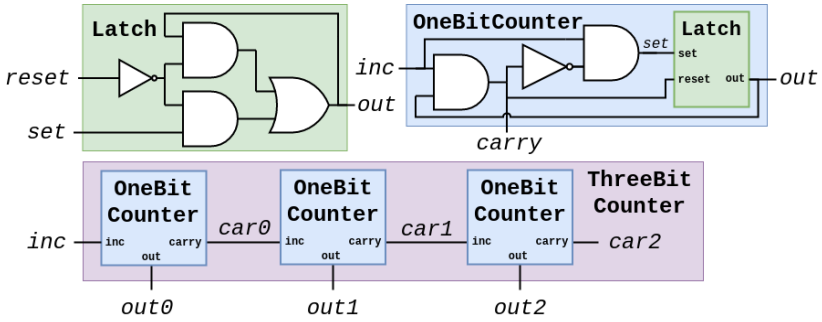
The tool suite contains four translators that take a model/query/witness as input and output a *behaviorally equivalent* model/query/witness in the target language.

`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 [13]). 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`.) As Figure 4 shows, the translation preserves the hierarchy between the SMV modules and submodule instantiations.

The MOXI encoding directly captures SMV variable declarations (`VAR`, `IVAR`, `FROZENVAR`), constants `CONSTANT`, macro and function declarations (`DEFINE`, `FUN`), state machine component declarations (`INIT`, `TRANS`, `INVAR`, `ASSIGN`), fairness constraints (`FAIRNESS`, `JUSTICE`, `COMPASSION`), and invariant specifications (`AG` [property], `INVARSPEC`). To support LTL specifications (`LTLSPEC`), `smv2moxi` runs PANDA [51], 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; (2) running the C preprocessor (SMV supports C-style macros) and PANDA [51] (for LTL specifications); (3) parsing via a SLY-generated [7] parser; (4) running a SMV type checker; (5) translating to MOXI. We emphasize that tool guarantees apply to *well-formed* SMV models as determined by NUXMV.

[7] 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 one-bit counter uses a latch to store that counter's current value. (Middle) A MoXI implementation of the circuit uses `define-system` commands (lines 1-20) to describe and compose each component of the counter. 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). Function `to_bv3` (whose definition is omitted for space constraints) converts the returned 3 bits to the corresponding bit-vector value. The `check-system` command (lines 40-45) queries whether it is possible that their outputs differ. (Bottom) The resulting `check-system-response` shows that the reachability query is unsatisfiable, proving the two counters equivalent.

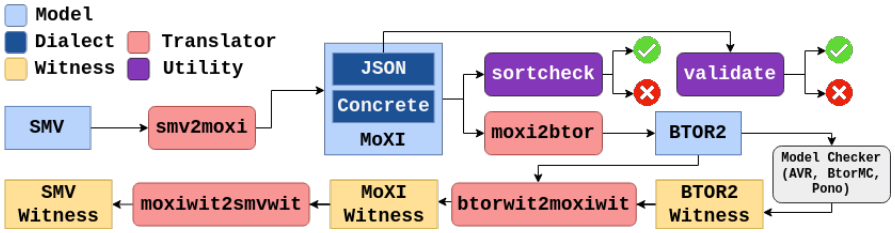


Fig. 3: Starting with a NuXmv model, `smv2moxi` generates a behaviorally equivalent MoXI model in either the MoXI concrete syntax or a JSON dialect syntax. `moxi2btor` translates this MoXI model to a set of Btor2 models, one for each query, which an off-the-shelf model checker (e.g., AVR [31], Pono [40], BtorMC [46]) 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.

moxi2btor translates MOXI to BTOR2 by creating a BTOR2 file for each `:query` attribute in each `check-system` command.

There are some important differences between MOXI and BTOR2 that 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, an 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 [7]; (2) running `sortcheck` (Section 3.2); (3) translating to a set of BTOR2 files, each behaviorally equivalent to its corresponding `:query`.

btorwit2moxiwit translates BTOR2 witnesses to `check-system-responses` of MOXI. It assumes `moxi2btor` created the BTOR2 input files used to generate the witness and uses information that `moxi2btor` encodes in comments of each BTOR2 file. For example, to map bit vectors to enumeration values for variables of such sorts.

moxiwit2smvwit translates MOXI responses to SMV-language response output.

3.2 Utilities

sortcheck We provide a reference 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 in keeping front-end tools 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 widely available JSON parsers (e.g., simdjson, RapidJSON), to obtain industrial-strength MOXI parsers in the language of their choice "for free." We plan to accompany each MOXI release with a corresponding JSON Schema, enabling seamless front-end compatibility with the latest MOXI standard along with language/platform independence.

We provide a tool, `validate`, that invokes an off-the-shelf JSON validator from Python's `jsonschema` package to validate an MOXI file (in the JSON dialect) against the official MOXI JSON Schema.

4 Tool Suite Validation

We validate our tools using a combination of manual inspection, sort checking of translated output, and comparison of witnesses between those generated by NUXMV and our end-to-end tool suite. We use CATBTOR [46] 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.

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: For a simple delay circuit, the toolchain translates the SMV model 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.

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.

Manual Inspection We provide an initial set of hand-written MOXI bechmarks to perform manual validation. Each benchmark is well-sorted according to `sortcheck`, generates well-sorted BTOR2 via `moxi2btor` according CATBTOR, and generates correct, manually-inspected witnesses via BTORMC and `btorwit2moxiwit`.[8]

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 numerous 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 [13] but were accepted by NUXMV nonetheless, particularly with regard to 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 again the NUXMV benchmarks 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. Figure 6 shows that our tool chain (using BTORMC, AVR, or PONO as its back end) returns results in time competitive with NUXMV when the latter is set to use BMC or k-induction.

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 other researchers adopting MOXI as an intermediate language.

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 that contain only boolean/word/array types. We use the HWMCC 2020 versions of AVR and PONO, the version of BTORMC from the latest version of Boolector [46], 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 [9] and k-induction [55] 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 compared to NUXMV-bmc while ensuring all BTOR2 witnesses are correctly translated to SMV traces. For safe queries, we measure competitive performance with vb-kind solving 6% more benchmarks compared to NUXMV-kind. The vb performance gains are due to its ability to use

[8] Many thanks to Daniel Larraz for writing many of the MOXI examples.

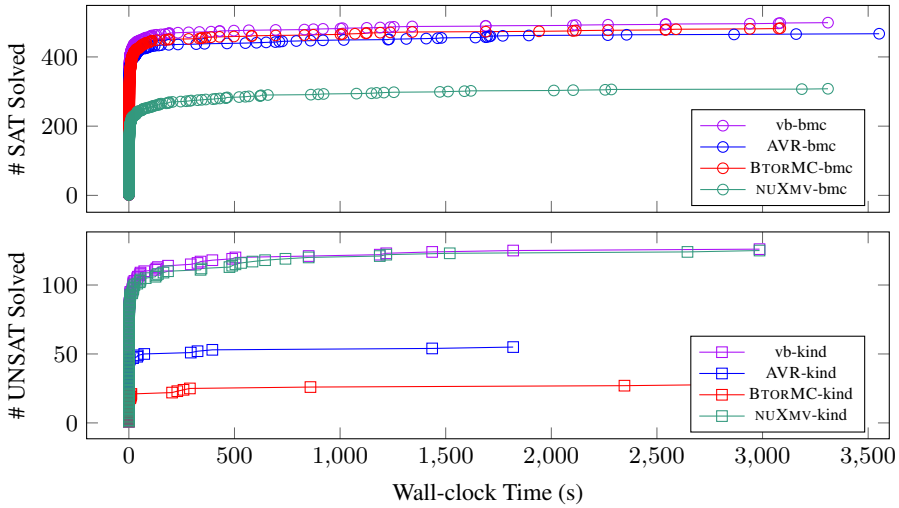


Fig. 6: Performance comparison on unsafe and safe queries with BMC and k-induction, respectively, across different model checkers. vb-* represents the virtual-best solver. Wall-clock time for the non-nuXmv plots include translation time.

a variety of model checkers with different SMT solver backends of varying strengths, e.g., nuXmv uses MathSAT [22], AVR uses Yices [28], and Pono uses Boolector [46], while ensuring correct model and witness translation through MoXI.

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 to this tool suite a translator between that language and MoXI. 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 that algorithm. Benchmarking against other model-checking algorithms no longer requires re-implementing existing tools in order to achieve an apples-to-apples comparison.

This release enables future instantiations of HWMCC [10] to center 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 [48] would provide an additional trusted translation between languages, beyond the validation techniques in Section 4. Writing a back end to Yosys [58], the open-source RTL synthesis framework, to generate files directly from Verilog designs would facilitate creating a larger set of realistic benchmarks to add to the initial set in Section 5. These benchmarks would be good candidates for use in a future HWMCC. Finally, we expect that 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 under way for the Kind 2 checker.

References

1. The AIGER and-inverter graph (AIG) format version 20071012. http://fmv.jku.at/aiger/FORMAT, accessed: 2016-07-25
2. AIGER 1.9 and beyond. http://fmv.jku.at/hwmcc11/beyond1.pdf, accessed: 2016-07-25
3. AIGER website. http://fmv.jku.at/aiger/, accessed: 2016-07-25
4. Alur, R.: Principles of cyber-physical systems. MIT press (2015)
5. Barrett, C., Moura, L., Stump, A.: SMT-COMP: Satisfiability Modulo Theories Competition. In: Proc. 17th Int'l Conf. on Computer Aided Verification. Lecture Notes in Computer Science, vol. 3576, pp. 20–23. Springer (2005)
6. 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)
7. Beazley, D.: Sly (sly lex yacc). https://sly.readthedocs.io/en/latest/ (2018)
8. 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/
9. 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
10. Biere, A., Froleyks, N., Preiner, M.: Hardware Model Checking Competition (HWMCC). https://fmv.jku.at/hwmcc20/index.html (2020)
11. 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)
12. 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)
13. 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)
14. 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)
15. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: International Conference on Computer Aided Verification. pp. 24–40. Springer (2010)
16. 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)
17. 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)
18. Cavada, R., Cimatti, A., Jochim, C.A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: Nusmv 2.6 user manual (2016)

19. 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)
20. Choi, Y., Heimdahl, M.: Model checking software requirement specifications using domain reduction abstraction. In: IEEE ASE. pp. 314–317 (2003)
21. 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)
22. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS. pp. 93–107 (2013)
23. 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)
24. Documentation, S.: Simulation and model-based design (2020), https://www.mathworks.com/products/simulink.html
25. Documentation, SCADE: Ansys SCADE Suite (2023), https://www.ansys.com/products/embedded-software/ansys-scade-suite
26. 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
27. 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)
28. Dutertre, B.: Yices 2.2. In: International Conference on Computer Aided Verification. pp. 737–744. Springer (2014)
29. 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
30. 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
31. 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)
32. 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)
33. Holzmann, G.: Design and Validation of Computer Protocols. Prentice-Hall Int. Editions (1991)
34. Hunt, W.: FMCAD organization home page. http://www.cs.utexas.edu/users/hunt/FMCAD/
35. IEEE: IEEE standard for Verilog hardware description language (2005)
36. IEEE: IEEE standard for VHDL language reference manual (2019)
37. Kessler, F.B.: Verification modulo theories. http://www.vmt-lib.org/, accessed: 2017-09-30

38. 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
39. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
40. 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)
41. 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)
42. McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
43. 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
44. Miller, S.P., Tribble, A.C., Whalen, M.W., Per, M., Heimdahl, E.: Proving the shalls. STTT **8**(4-5), 303–319 (2006)
45. 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)
46. 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)
47. The nuXmv model checker; available at https://nuxmv.fbk.eu/, 2015
48. 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)
49. Raimondi, F., Lomuscio, A., Sergot, M.J.: Towards model checking interpreted systems. In: FAABS 02, LNAI 2699. pp. 115–125. Springer (2002)
50. 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)
51. 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)
52. 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)
53. 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)
54. 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)

55. 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)
56. SMTLib. `https://smtlib.cs.uiowa.edu/`
57. 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)
58. Wolf, C.: Yosys open synthesis suite (2016)
59. Yoo, J., Jee, E., Cha, S.: Formal modeling and verification of safety-critical software. Software, IEEE **26**(3), 42–49 (2009)
60. 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)
61. 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)