# MoXI: An Intermediate Language for Symbolic Model Checking [⋆]

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

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

**Abstract.** Three progressive challenges stand in between the popular, "push-button," industrially valuable technique of symbolic model checking and the level of widespread adoption achieved by other verification techniques: (1) the specification bottleneck; (2) the state-space explosion problem; and (3) the lack of standardization and open-source implementations limiting the impact of advances in (1) and (2). We address this third challenge. Learning from past definitions of intermediate languages and common interfaces, as well as input from the international research community, we define a new, extensible intermediate language for hardware symbolic model checking. Our contributions include: (a) defining the syntax and semantics of **MoXI**, the **Mo**del e**X**change **I**nterlingua designed to become a standard for the international research community; (b) demonstrating that an initial implementation of symbolic model checking through MoXI performs competitively with current state-of-the-art symbolic model checkers; (c) reframing the next symbolic model checking research challenges considering this new community standard.

**Keywords:** Model Checking · Intermediate Language · SMT.

## 1 Introduction

Symbolic model checking has made foundational changes to impactful, real-world system designs, yet its ascent to a common-place verification technique is currently most limited by a few barriers to adoption, centering on its lack of standardization. For just one example, in our own work, symbolic model checking with nuXmv pinpointed hard-to-find, requirements-violating control sequences, and thus changed the design of NASA's Automated Airspace Concept [56, 57]. This led to NASA using nuXmv for the next, much larger, stages of the project,

[⋆] This work was funded by NSF:CCRI Award #2016592, #2016597, #2016656. The GitHub organization provides full artifacts: https://github.com/ModelChecker. Thanks to our international Technical Advisory Board for invaluable feedback; see the project website: https://modelchecker.temporallogic.org.

including model checking with fault-tree analysis of a large set of possible safe configurations of the next-generation system [41], and design-space exploration of over 20,000 possible air traffic control designs [28].

The collection of impactful success stories of symbolic model checking in real-world system development are far too many and too diverse to cite in a single paper; there is no question that model checking provides value beyond its cost in verifying a wide range of systems uphold requirements for safety, security, and other desirable properties (such as consistency or financial soundness). However, model checking is still not as commonly-used as informal verification techniques such as simulation and testing. One primary reason for this is the specification bottleneck [51]: creating and validating system models and temporal logic specifications remains a challenging undertaking. The second barrier to adoption is the famous state-space explosion problem: the combination of the modeling technique used to represent the relevant system characteristics and the back-end model-checking algorithm can result in an untenable search space. However, the third barrier is perhaps both the broadest impediment and the easiest to overcome: a lack of standardization in symbolic model checking prevents the propagation of techniques aimed at lowering to the first two barriers.

The SMV modeling language represents an advancement in ameliorating barrier (1), the specification bottleneck: it is an expressive modeling language that, due to its appealing syntax that intuitively represents many common systems, continues to be successfully used in a wide range of industrial verification efforts [7,8,13,21,22,27,28,31,38,41,44,45,49,54–57]. Two freely-available model checkers previously provided viable research platforms for checking SMV language models: CadenceSMV [43] and NuSMV [14] (which is integrated into today's nuXmv [47]). 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. How can we continue the progression of high-level language model checking in SMV with no open-source research platforms that allow new algorithms under the hood, e.g., to continue mitigating the state-space explosion problem?

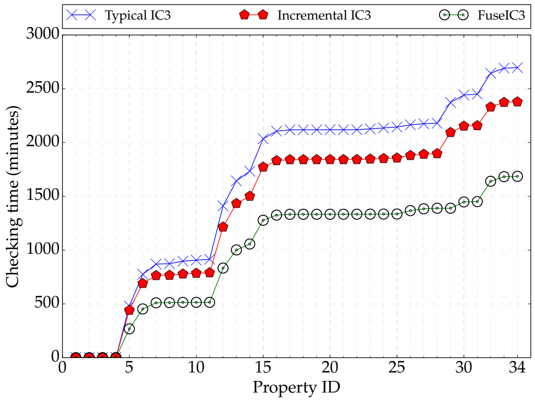

Fig. 1: FuseIC3 [22]: model checking **34 formulas** over **1,620 models** is **5.48x faster.**

Continuing with our example of symbolic model checking in NASA's next-generation transportation system, the next step of the process was to narrow down the design space (system designers reduced the possible configurations from 20,250 to 1,620), add details, and continue design-space model checking to refine the (safe) design set. To improve performance (per barrier (2), the state space explosion problem), we designed a new algorithm for model checking large design spaces, FuseIC3 [22]. However, to compare FuseIC3 with the state-of-the-art symbolic model-checking algorithms (namely, those claimed to be used in nuXmv [24] and in industry [4]), we had to **re-implement both of these existing model checkers from scratch** — our results appear in Figure 1. In other words, there was no easy way to compare state-of-the-art model checking back-end algorithms because the existing implementations were closed source, and did not check models in the same modeling languages. No tool accepted our SMV-language models and implemented the best back-end model-checking algorithm for the industrial verification task at hand, and there was no way to add additional algorithms to nuXmv because it is closed source. (It would also have been useful to check how closely nuXmv implemented the state-of-the-art algorithms described in the literature, and to be able to modify the existing implementation.) In addition to being a time-consuming engineering task, this hurdle also curtailed the use of model checking in this NASA project. The lack of standardization in symbolic model checking meant that it was no longer the case that the benefits of model checking outweighed the cost. While we successfully ameliorated barriers (1) and (2), at least in the context of the task at hand, we could not apply our advances efficiently, or in the context of existing, and trusted, model-checking infrastructures.

This is not a unique story. The Hardware Model Checking Competition (HWMCC) [6] regularly spurs back-end algorithmic advances in symbolic model checking. However, the competing tools historically only accepted models in the bit-level input language Aiger [1, 2]; HWMCC only recently advanced to the word-level, yet still machine-oriented, language Btor2 [46]. Such languages do not support direct modeling of modern complex systems — the way SMV does — and hinder adoption; it is very hard to convince industrial system designers that Aiger or Btor2 models correctly capture their higher-level systems. Most existing pathways for translating from high-level models to Aiger or Btor2 focus on hardware designs, and do not provide a natural means to describe realistic systems. Even outside of HWMCC, many new, advanced, model-checking algorithms get developed for open-source, award-winning, model-checking engines, like ABC [10], which do not accept models in popular languages that would benefit from model checking using those algorithms. Then, we also face the open problem of translating counterexamples produced, e.g., by ABC, back into meaningful counterexamples for a non-hardware-centric higher-level language model, such as a model in SMV.

For another example, the PANDA algorithm and tool achieved up to exponential performance improvement using the same symbolic model-checking back-end, just by encoding LTL input specifications differently [50]. Yet PANDA's impact is limited since the implementation is only compatible with tools that accept models written in SMV. If, instead, PANDA translated LTL specifications into a common intermediate language, we could use it to improve model-checking performance broadly.

To alleviate the language issues above we introduce **Model Exchange Interlingua (MoXI)**, an intermediate language meant to be a common input and output standard for model checkers for finite- and infinite-state systems. MoXI was designed to be general enough to be an intermediate target language for a variety of user-facing specification languages for model checking. At the same time, MoXI is simple enough to be easily comparable to lower-level languages or be directly supported by model-checking tools based on SAT/SMT technology.

Models expressed in MoXI are expected to be produced and processed by tools. Thus MoXI provides: a simple, easily-machine-parsable syntax; a rich set of data types; minimal syntactic sugar (at least initially); a well-understood formal semantics; a small but comprehensive set of commands; and simple translations to lower-level modeling languages, such as Btor2 [46]. Based on these principles, MoXI does not provide direct support for many of the expressive features offered by current hardware modeling languages such as VHDL [33,35] and Verilog [34], or more general-purpose system modeling languages such as SMV [42,48], TLA+ [39], PROMELA [32], Simulink [19], SCADE [17,20], and Lustre [11,36]. However, it is sufficiently expressive that problems defined in reasonably large fragments of those languages can be reduced to problems in MoXI. MoXI closely resembles the SMT-LIB language [3,53], but with new commands to define and verify systems. It allows the definition of multi-component synchronous or asynchronous reactive systems. It also allows the specification and checking of reachability conditions (or, indirectly, state and transition invariants) and deadlocks, possibly under fairness conditions on input values.

With MoXI, it is now possible to introduce or refine expressive modeling languages, model check them using existing and future state-of-the-art back-end algorithms, and return counterexamples in the original, high-level modeling language, simply by creating a translation between the modeling language and MoXI. Advances in back-end model checking algorithms can now be leveraged for models in any number of high-level modeling languages, simply by creating a translation between their low-level representations and MoXI. Most importantly, innovations mitigating the specification bottleneck and the state-space explosion problem will now push forward symbolic model checking as a whole, rather than advancing a single tool.

We introduce the notation and trace semantics underlying MoXI in Section 2. Section 3 defines MoXI, including the semantics for system definition and system checking. We demonstrate a prototype implementation translating the SMV modeling language through MoXI for model checking with the top tools from the last HWMCC in Section 4. Section 5 concludes with a concrete list of future research directions for the international symbolic model checking community, given the standardization provided by MoXI.

## 2 Preliminaries

The base logic of MoXI is the same as that of SMT-LIB: many-sorted first-order logic with equality and `let` binders. We refer to this logic simply as FOL. When we say *formula*, with no further qualifications, we refer to an arbitrary formula of FOL (possibly with quantifiers and `let` binders).

We say that a formula is *quantifier-free* if it contains no occurrences of the quantifiers $\forall$ and $\exists$. We say that it is *binder-free* if it is quantifier-free and also contains no occurrences of the `let` binder. The *scope* of binders and the notion of *free* and *bound* (occurrences of) variables in a formula are defined as usual.

### 2.1 Notation

If $F$ is a formula and $\boldsymbol{x} = (x_1, \ldots, x_n)$ a tuple of distinct variables, we write $F[\boldsymbol{x}]$ or $F[x_1, \ldots, x_n]$ to express the fact that every variable in $\boldsymbol{x}$ occurs free in $F$ (although $F$ may have additional free variables). Let $\sigma(x_i)$ denote the type or *sort* of variable $x_i$ in $\boldsymbol{x}$. We denote by $\boldsymbol{x'}$ the tuple $(x'_1, \ldots, x'_n)$ such that $\sigma(x_i) = \sigma(x'_i)$. We write $\boldsymbol{x}, \boldsymbol{y}$ to denote the concatenation of tuple $\boldsymbol{x}$ with tuple $\boldsymbol{y}$. When it is clear from the context, given a formula $F[\boldsymbol{x}]$ and a tuple $\boldsymbol{t} = (t_1, \ldots, t_n)$ of terms of the same type as $\boldsymbol{x} = (x_1, \ldots, x_n)$, we write $F[\boldsymbol{t}]$ or $F[t_1, \ldots, t_n]$ to denote the formula obtained from $F$ by simultaneously replacing each occurrence of $x_i$ by $t_i$ for all $i = 1, \ldots, n$. A formula may contain *uninterpreted* constant and function symbols, that is, symbols with no constraints on their interpretation. For most purposes, we treat uninterpreted constant and function symbols as free (rigid) variables respectively of first and second order.

**Definition 1 (Transition system).** *A transition system $S$ is a pair of predicates of the form $S = (I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}],\ T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}, \boldsymbol{i'}, \boldsymbol{o'}, \boldsymbol{l'}])$ where*

1. *$\boldsymbol{i}$ and $\boldsymbol{i'}$ are tuples of* input variables*;*
2. *$\boldsymbol{o}$ and $\boldsymbol{o'}$ are tuples of* output variables*;*
3. *$\boldsymbol{l}$ and $\boldsymbol{l'}$ are tuples of* local variables*;*
4. *$I_S$ is a formula representing the* initial state condition*; and*
5. *$T_S$ is a formula representing the* transition condition*.*

### 2.2 Trace Semantics

A transition system implicitly defines a model (i.e., a Kripke structure) of First-Order Linear Temporal Logic (FO-LTL). The language of FO-LTL extends that of FOL with the same modal operators of time as in standard (propositional) LTL: **always**, **eventually**, **next**, **until**, **release**. For our purposes of defining the semantics of transition systems, it is enough to consider just the **always** and **eventually** operators.

The set of non-temporal operators depends on the particular theory, in the sense of SMT, considered (e.g., linear integer/real arithmetic, bit vectors, strings, and so on, and their combinations). The meaning of theory symbols (such as arithmetic operators) and theory sorts (such as $\mathsf{Int}$, $\mathsf{Real}$, $\mathsf{Array}(\mathsf{Int}, \mathsf{Real})$, $\mathsf{BitVec}(3)$, $\ldots$) is fixed by the theory $\mathcal{T}$. With a fixed theory $\mathcal{T}$, the meaning of a FO-LTL formula $F$ is provided by an interpretation of the uninterpreted (constant and function) symbols of $F$, if any, as well as an infinite sequence of valuations for the free variables of $F$.

Let tuple $\boldsymbol{x} = (x_1, \ldots, x_n)$ denote distinct *state* variables, meant to represent the state of a computation system. We write formulas of the form $F[\boldsymbol{f}, \boldsymbol{x}, \boldsymbol{x'}]$ where $\boldsymbol{f}$ is a tuple of uninterpreted (constant and function) symbols. If $F$ has free occurrences of variables from $\boldsymbol{x}$ but not from $\boldsymbol{x'}$ we call it a *one-state* formula; otherwise, we call it a *two-state* formula.

A *valuation* of $\boldsymbol{x}$, or a *state over* $\boldsymbol{x}$, is a function mapping each variable $x$ in $\boldsymbol{x}$ to a value of $x$'s sort. Let $\kappa$ be a positive ordinal up to $\omega$, the cardinality of the natural numbers. A *trace* (of length $\kappa$ over $\boldsymbol{x}$) is any state sequence $\pi = (s_j \mid 0 \leq j < \kappa)$. Note that $\pi$ is the finite sequence $s_0, \ldots, s_{\kappa-1}$ when $\kappa < \omega$, and is the infinite sequence $s_0, s_1, \ldots$ otherwise. For all $i$ such that $0 \leq i < \kappa$, we denote by $\pi[i]$ the state $s_i$ and by $\pi^i$ the subsequence $(s_j \mid i \leq j < \kappa)$.

**Infinite trace semantics** Let $F[\boldsymbol{f}, \boldsymbol{x}, \boldsymbol{x'}]$ be a formula as above. If $\mathcal{I}$ is an interpretation of $\boldsymbol{f}$ in the theory $\mathcal{T}$ and $\pi$ is an infinite trace, then $(\mathcal{I}, \pi)$ *satisfies* $F$, written $(\mathcal{I}, \pi) \models F$, *iff* one of the following holds:

1. $F$ is atomic and $\mathcal{I}[\boldsymbol{x} \mapsto \pi[0](\boldsymbol{x}), \boldsymbol{x'} \mapsto \pi[1](\boldsymbol{x})]$ satisfies $F$ as in FOL;
2. $F = \neg G$ and $(\mathcal{I}, \pi) \not\models G$;
3. $F = G_1 \wedge G_2$ and $(\mathcal{I}, \pi) \models G_i$ for $i = 1, 2$;
4. $F = \exists z\, G$ and $(\mathcal{I}[z \mapsto v], \pi) \models G$ for some value $v$ for $x$;
5. $F =$ **eventually** $G$ and $(\mathcal{I}, \pi^i) \models G$ for some $i \geq 0$;
6. $F =$ **always** $G$ and $(\mathcal{I}, \pi^i) \models G$ for all $i \geq 0$.

The semantics of the propositional connectives $\vee, \rightarrow, \leftrightarrow$, and the quantifier $\forall$ can be defined by reduction to the connectives above (e.g., by defining $G_1 \vee G_2$ as $\neg(\neg G_1 \wedge \neg G_2)$, and so on). Note that $\exists z$ is a *static*, or *rigid*, quantifier: the meaning of the variable it quantifies does not change over time, i.e., from state to state in $\pi$. Uninterpreted symbols are rigid in the same sense: their meaning does not change over time.[6] Given a transition system $S = (I_S, T_S)$, the infinite trace semantics of $S$ is the set of all pairs $(\mathcal{I}, \pi)$ of interpretations $\mathcal{I}$ in $\mathcal{T}$ and infinite traces $\pi$ such that $(\mathcal{I}, \pi) \models I_S \wedge$ **always** $T_S$. We call any such pair an *execution of* $S$.

**Finite trace semantics** Given formula $F[\boldsymbol{f}, \boldsymbol{x}, \boldsymbol{x'}]$, interpretation $\mathcal{I}$ in theory $\mathcal{T}$, infinite trace $\pi$, and $n \geq 0$, $(\mathcal{I}, \pi)$ *n-satisfies* $F$, written $(\mathcal{I}, \pi) \models_n F$, *iff*

1. $F$ is atomic and $\mathcal{I}[\boldsymbol{x} \mapsto \pi[0](\boldsymbol{x}), \boldsymbol{x'} \mapsto \pi[1](\boldsymbol{x})]$ satisfies $F$ as in FOL;
2. $F = \neg G$ and $(\mathcal{I}, \pi) \not\models_n G$;
3. $F = G_1 \wedge G_2$ and $(\mathcal{I}, \pi) \models_n G_i$ for $i = 1, 2$;
4. $F = \exists z\, G$ and $(\mathcal{I}[z \mapsto v], \pi) \models_n G$ for some value $v$ for $z$;
5. $F =$ **eventually** $G$ and $(\mathcal{I}, \pi^i) \models_{n-i} G$ for some $i = 0, \ldots, n$; or
6. $F =$ **always** $G$ and $(\mathcal{I}, \pi^i) \models_{n-i} G$ for all $i = 0, \ldots, n$.

The semantics of the propositional connectives $\vee, \rightarrow, \leftrightarrow$, and the quantifier $\forall$ is defined by reduction to the connectives above. Intuitively, $n$-satisfiability

[6] Another way to understand the difference between rigid and non-rigid symbols is that state variables are mutable over time, whereas quantified variables, theory symbols, and uninterpreted symbols are all immutable.

specifies when a formula is true over a trace's first $n$ states. Note that this notion is well defined even when $n = 0$ regardless of whether $F$ has free occurrences of variables from $\boldsymbol{x'}$ or not. This is true in the atomic case because the infinite trace $\pi$ contains the state $\pi[1]$. The claim can be shown in the general case by a simple inductive argument.

The notion of $n$-satisfiability is useful in state reachability. A state satisfying a (non-temporal) state property $R$ is reachable in a system $S$ only if the temporal formula **eventually** $R$ is $n$-satisfied by an execution of $S$ for some $n$. Note that the converse does not hold; $R$ can be reachable in a system $S$ without being $n$-satisfied by an execution of $S$.

## 3 The MoXI Intermediate Language

MoXI assumes a discrete and linear notion of time and adopts the trace-based semantics defined in the previous section. It builds on the SMT-LIB language, extending it with commands to represent transition systems and to specify properties or *queries*. It also standardizes a format for witnesses generated by back-end algorithms. Table 1 shows the supported SMT-LIB commands in MoXI. Since enumerated sorts are useful in modeling real-world systems, MoXI introduces a `declare-enum-sort` command. For example, `(declare-enum-sort` $s$ `(`$c_1 \cdots c_n$ `))` declares $s$ to be an enumerative type with (distinct) values $c_1, \ldots, c_n$.

Table 1: Supported SMT-LIB commands in MoXI

| | |
|---|---|
| `(declare-sort` $s$ $n$`)` | Declares $s$ to be a sort symbol (i.e., type constructor) of arity $n$. |
| `(define-sort` $s$ `(`$u_1 \cdots u_n$`)` $\tau$`)` | Defines $S$ as a synonym of a parametric type $\tau$ with parameters $u_1 \cdots u_n$. |
| `(declare-const` $c$ $\sigma$`)` | Declares a constant $c$ of sort $\sigma$. |
| `(define-fun` $f$ `( (` $x_1$ $\sigma_1$ `)` $\cdots$ `(` $x_n$ $\sigma_n$ `) )` $\sigma$ $t$`)` | Defines a function $f$ with inputs $x_1, \ldots, x_n$ (of respective sort $\sigma_1, \ldots, \sigma_n$), output sort $\sigma$, and body $t$. |
| `(set-logic` $L$`)` | Defines the model's *data logic*, i.e., the background theories of relevant data types (e.g., integers, reals, bit vectors, and so on) as well as the language of allowed logical constraints (e.g., quantifier-free, linear, etc.). |

### 3.1 System Definition

MoXI allows the definition of a model as the composition of one or more *systems*. MoXI 's system definition commands follow the SMT-LIB syntax for attribute-value pairs. Each system definition:

1. defines a *transition system* via the use of SMT formulas, imposing minimal syntactic restrictions on those formulas;
2. is parameterized by a *state signature*, a sequence of typed variables;
3. partitions *state* variables into *input*, *output*, and *local* variables;
4. can be expressed as the (a)synchronous composition of other systems.

**Atomic systems** MoXI defines an *atomic transition system* that has $m$ inputs, $n$ outputs, and $p$ local variables via the command:

(`define-system` $S$<br>
`:input` ( $(i_1\ \delta_1) \cdots (i_m\ \delta_m)$ ) `:output` ( $(o_1\ \tau_1) \cdots (o_n\ \tau_n)$ )<br>
`:local` ( $(l_1\ \sigma_1) \cdots (l_p\ \sigma_p)$ ) `:init` $I$ `:trans` $T$ `:inv` $P$), where

- $S$ is the system's identifier;
- each $i_j$ is an *input* variable of sort $\delta_j$;
- each $o_j$ is an *output* variable of sort $\tau_j$;
- each $l_j$ is a *local* variable of sort $\sigma_j$;
- each $i_j$, $o_j$, $l_j$ denote *current-state* values;
- $I$, *the initial condition*, is a one-state formula over the unprimed system's variables (input, output, and local state variables) that expresses a constraint on the initial states of $S$;
- $T$, *the transition condition,* is a two-state formula over all of the system's variables (primed and unprimed) that expresses a constraint on the state transitions of $S$;
- $P$, *the invariance condition*, is a one-state formula over all of the *unprimed* system's variables that expresses a constraint on all reachable states of $S$.

Next-state variables are not provided explicitly but are denoted by convention by appending $'$ to the names of the current-state variables $i_j$, $o_j$, and $l_j$. Note that all attributes are optional but can occur at most once, and the order of the attributes is immaterial except that `:input`, `:output`, and `:local` must occur before `:init`, `:trans`, and `:inv`. The default value for a missing attribute is the empty list `()` for `:input`, `:output`, and `:local`; and `true` for `:init`, `:trans`, and `:inv`. Syntactically, the system identifier, the input, output, and local variables are SMT-LIB symbols. In contrast, the sorts $\delta_j$, $\tau_j$, $\sigma_j$ are SMT-LIB sorts, while the formulas $I$, $T$, and $P$ are SMT-LIB terms of type `Bool`.

*Semantics* Let $\boldsymbol{i} = (i_1, \ldots, i_m)$, $\boldsymbol{o} = (o_1, \ldots, o_n)$, $\boldsymbol{l} = (l_1, \ldots, l_p)$, and $\boldsymbol{x} = \boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}$. A system $S$ introduced by the `define-system` command above is a transition system whose behavior consists of all the (infinite) executions $(\mathcal{I}, \pi)$ over $\boldsymbol{x}$ such that $(\mathcal{I}, \pi) \models I[\boldsymbol{x}] \wedge \textbf{always}\ (P[\boldsymbol{x}] \wedge T[\boldsymbol{x}, \boldsymbol{x'}])$. We call $I_S = I[\boldsymbol{x}]$ *the initial state predicate* of $S$ and $T_S = P[\boldsymbol{x}] \wedge T[\boldsymbol{x}, \boldsymbol{x'}]$ *the transition predicate* of $S$.

**Composite systems** Transition systems can be defined as the synchronous[7] composition of other systems using the command:

(`define-system` $S$ `:input` ( $(i_1\ \delta_1) \cdots (i_m\ \delta_m)$ )<br>
`:output` ( $(o_1\ \tau_1) \cdots (o_n\ \tau_n)$ ) `:local` ( $(l_1\ \sigma_1) \cdots (l_p\ \sigma_p)$ )<br>
`:subsys` $(N_1\ (S_1\ \boldsymbol{x}_1\ \boldsymbol{y}_1)\ ) \cdots$ `:subsys` $(N_q\ (S_q\ \boldsymbol{x}_q\ \boldsymbol{y}_q)\ )$<br>
`:init` $I$ `:trans` $T$ `:inv` $P$), where

- `:input`, `:output`, `:local`, `:init`, `:trans`, and `:inv` are as in atomic system definitions;

[7] The asynchronous composition of systems is planned for a later version of MoXI.

- $q > 0$ and each $S_i$ is the name of a system other than $S$;
- the names $S_1 \ldots, S_q$ need not be all distinct;
- each $N_i$ is a local synonym for $S_i$, with $N_1, \ldots, N_q$ distinct;
- each $\boldsymbol{x}_i$ consists of $S$'s variables of the same sort as $S_i$'s input;
- each $\boldsymbol{y}_i$ consists of $S$'s local/output variables of the same sort as $S_i$'s output;
- the directed subsystem graph rooted at $S$ is acyclic.

*Semantics* For $k = 1, \ldots, q$, let $S_k = (I_k[\boldsymbol{i}_k, \boldsymbol{o}_k, \boldsymbol{l}_k], T_k[\boldsymbol{i}_k, \boldsymbol{o}_k, \boldsymbol{l}_k, \boldsymbol{i'}_k, \boldsymbol{o'}_k, \boldsymbol{l'}_k])$, with the elements of $\boldsymbol{l}_1, \ldots, \boldsymbol{l}_q$ all mutually distinct. Let $\boldsymbol{i} = (i_1, \ldots, i_m)$, $\boldsymbol{o} = (o_1, \ldots, o_n)$, $\boldsymbol{l} = (l_1, \ldots, l_p), \boldsymbol{l}_1, \ldots, \boldsymbol{l}_q$, and $\boldsymbol{x} = \boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}$. A composite system $S$ introduced by the `define-system` command above is a transition system whose behavior consists of all the (infinite) executions $(\mathcal{I}, \pi)$ over $\boldsymbol{x}$ such that $(\mathcal{I}, \pi) \models I_S[\boldsymbol{x}] \wedge$ **always** $T_S[\boldsymbol{x}, \boldsymbol{x'}]$, where

- $I_S[\boldsymbol{x}] = I[\boldsymbol{x}] \wedge \bigwedge_{k=1,\ldots,q} I_k[\boldsymbol{x}_k, \boldsymbol{y}_k, \boldsymbol{l}_k]$ and
- $T_S[\boldsymbol{x}, \boldsymbol{x'}] = P[\boldsymbol{x}] \wedge T[\boldsymbol{x}, \boldsymbol{x'}] \wedge \bigwedge_{k=1,\ldots,q} T_k[\boldsymbol{x}_k, \boldsymbol{y}_k, \boldsymbol{l}_k, \boldsymbol{x'}_k, \boldsymbol{y'}_k, \boldsymbol{l'}_k]$.

**Sanity Requirements on $\boldsymbol{I_S}$ and $\boldsymbol{T_S}$** Every system defined in MoXI is expected to execute forever. This is not a limitation in practice because systems meant to reach a final state can be modeled with states that cycle back to themselves and produce stuttering outputs. In such semantics, the reachability of a *deadlocked state* (i.e., a state with no successors in the transition relation) indicates the presence of an error in the system's definition. For a system definition to define a deadlock-free system, the following must hold for the variables $\boldsymbol{x} = \boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}$ and their primed versions.

(1) Every assignment of values to the input variables $\boldsymbol{i}$ can be extended to an assignment to $\boldsymbol{x}$ that satisfies $I_S[\boldsymbol{x}]$.
(2) For every reachable state $s$ (i.e., assignment to the variables $\boldsymbol{x}$), every assignment to the primed input variables $\boldsymbol{i'}$ can be extended to an assignment $s'$ to $\boldsymbol{x'}$ so that $s, s'$ satisfies $T_S[\boldsymbol{x}, \boldsymbol{x'}]$.

The first restriction guarantees that the system can start. The second ensures that from any reachable state and for any new input, the system can move to another state (*and* also produce output). Given a specified background theory,

- A sufficient condition for (1) is the validity of the formula

$$\forall \boldsymbol{i}\, \exists \boldsymbol{o}\, \exists \boldsymbol{l}\; I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}]$$

- A sufficient condition for (2) is the validity of the formula

$$\forall \boldsymbol{i}\; \forall \boldsymbol{o}\; \forall \boldsymbol{l}\; \forall \boldsymbol{i'}\; \exists \boldsymbol{o'}\; \exists \boldsymbol{l'}\; T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}, \boldsymbol{i'}, \boldsymbol{o'}, \boldsymbol{l'}]$$

  Note that this is not a necessary condition as it needs not apply to unreachable states. Let Reachable$[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}]$ denote the (possibly higher-order) formula satisfied exactly by the reachable states of $S$. Then, a more accurate sufficient condition for (2) above would be the validity of the formula

$$\forall \boldsymbol{i}\; \forall \boldsymbol{o}\; \forall \boldsymbol{l}\; \forall \boldsymbol{i'}\; \exists \boldsymbol{o'}\; \exists \boldsymbol{l'}\; \text{Reachable}[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}] \Rightarrow T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{l}, \boldsymbol{i'}, \boldsymbol{o'}, \boldsymbol{l'}]$$

### 3.2 System Checking

The properties to check for a (possibly composite) defined system are specified using the following command for defining *queries* on the system's behavior.

(`check-system` $S$
 `:input` ( $(i_1\ \delta_1) \cdots (i_m\ \delta_m)$ ) `:output` ( $(o_1\ \tau_1) \cdots (o_n\ \tau_n)$ )
 `:local` ( $(l_1\ \sigma_1) \cdots (l_p\ \sigma_p)$ ) `:assumption` ($a$ A ) `:fairness` $(f\ F)$
 `:reachable` $(r\ R)$ `:current` $(c\ C)$ `:query` ($q$ $(g_1 \cdots g_q)$ )
 `:queries` ( ($q_1$ $(g_{1,1} \cdots g_{1,n_1})$ ) $\cdots$ ($q_t$ $(g_{t,1} \cdots g_{t,n_t})$ ) )), where

- $S$ is the identifier of a system with $m$ inputs, $n$ outputs, and $p$ local variables;
- $\boldsymbol{i} = (i_1, \ldots, i_m)$ is a renaming of input variables in $S$ of sort $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_m)$;
- $\boldsymbol{o} = (o_1, \ldots, o_n)$ is a renaming of output variables in $S$ of sort $\boldsymbol{\tau} = (\tau_1, \ldots, \tau_n)$;
- $\boldsymbol{l} = (l_1, \ldots, l_p)$ is a renaming of local variables in $S$ of sort $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_p)$;
- $a$, $r$, $f$, $c$, $q$, $q_1, \ldots, q_k$ are identifiers;
- $A$ is a formula over $\boldsymbol{i}$, $\boldsymbol{o}$, $\boldsymbol{l}$, $\boldsymbol{i'}$ expressing an *assumption* on $\boldsymbol{i'}$;
- $F$ is a formula over $\boldsymbol{i}$, $\boldsymbol{o}$, $\boldsymbol{l}$, $\boldsymbol{i'}$ expressing a *fairness condition* on $\boldsymbol{i'}$;
- $R$ is a formula over $\boldsymbol{i}$, $\boldsymbol{o}$, $\boldsymbol{l}$, $\boldsymbol{i'}$, $\boldsymbol{o'}$, $\boldsymbol{l'}$ expressing a state *reachability condition*;
- $C$ is a formula over $\boldsymbol{i}$, $\boldsymbol{o}$, $\boldsymbol{l}$ expressing a state *initiality condition*;
- each $g_j$ and $g_{j,k}$ ranges over the $a$, $r$, $f$, $c$ identifiers;
- $(q\ \ (g_1 \cdots g_q))$ defines a query $q$ as consisting of the formulas named by $g_1, \ldots, g_q$; the same holds for each $(q_j\ \ (g_{j,1} \cdots g_{j,n_j}))$.

Note that $A$, $F$, $R$, and $C$ are all non-temporal formulas. Each of the attributes `:assumption`, `:reachable`, `:query`, and `:queries` can occur zero or more times. Moreover, a query can contain more than one assumption, fairness condition, and reachability condition but at most one initiality condition.

*Semantics* Each query ($q$ and each $q_j$) in the `check-system` command asks for the existence of a trace. The query is to be evaluated with infinite-state semantics if it includes at least one fairness condition, and finite-state semantics otherwise. Specifically, for a system $S$, let $I_S$ and $T_S$ be the initial state and transition predicates of $S$ *modulo* the variable renamings in the `check-system` command. Let $t, u, v \geq 0$. The semantics of a query are defined as follows:

(1) A query of the form $(a_1\ \cdots\ a_t\ r_1\ \cdots\ r_u)$, where each $a_j$ and $r_j$ identify an assumption $A_j$ and reachability condition $R_j$, respectively, is *satisfiable* iff

$$\begin{aligned} I_S \wedge\ & \textbf{always}\ T_S \\ & \wedge\ \textbf{always}\ (A_1 \wedge \cdots \wedge A_t) \\ & \wedge\ \textbf{eventually}\ R_1 \wedge \cdots \wedge \textbf{eventually}\ R_u \end{aligned}$$

is $n$-**satisfiable** in LTL for some $n \geq 0$.

(2) A query of the form $(c\ a_1\ \cdots\ a_t\ r_1\ \cdots\ r_u)$, where $c$ is the initiality condition $C$, and each $a_j$ and $r_j$ identify an assumption $A_j$ and reachability condition $R_j$, respectively, is *satisfiable* iff

$$\begin{aligned}&C \land \textbf{always } T_S\\ &\quad\land \textbf{always } (A_1 \land \cdots \land A_t)\\ &\quad\land \textbf{eventually } R_1 \land \cdots \land \textbf{eventually } R_u\end{aligned}$$

is $n$-**satisfiable** in LTL for some $n \geq 0$.

(3) A query of the form $(a_1 \cdots a_t\ r_1 \cdots r_u\ f_1 \cdots f_v)$, where each $a_j$, $r_j$, and $f_j$ identify an assumption $A_j$, a reachability condition $R_j$, and a fairness condition $F_j$, respectively, is *satisfiable* iff

$$\begin{aligned}&I_S \land \textbf{always } T_S\\ &\quad\land \textbf{always } (A_1 \land \cdots \land A_t)\\ &\quad\land \textbf{always eventually } F_1 \land \cdots \land \textbf{always eventually } F_v\\ &\quad\land \textbf{eventually } R_1 \land \cdots \land \textbf{eventually } R_u\end{aligned}$$

is **satisfiable** in LTL.

(4) A query of the form $(c\ a_1 \cdots a_t\ r_1 \cdots r_u\ f_1 \cdots f_v)$, where $c$ is the initiality condition $C$ and each $a_j$, $r_j$, and $f_j$ identify an assumption $A_j$, a reachability condition $R_j$, and a fairness condition $F_j$, respectively, is *satisfiable* iff

$$\begin{aligned}&C \land \textbf{always } T_S\\ &\quad\land \textbf{always } (A_1 \land \cdots \land A_t)\\ &\quad\land \textbf{always eventually } F_1 \land \cdots \land \textbf{always eventually } F_v\\ &\quad\land \textbf{eventually } R_1 \land \cdots \land \textbf{eventually } R_u\end{aligned}$$

is **satisfiable** in LTL.

Let $\mathcal{T}$ be the background theory specified for a MoXI model. For each satisfiable query in the `check-system` command, the back-end model checking algorithm is expected to produce (1) a $\mathcal{T}$-interpretation $\mathcal{I}$ of the (global) free symbols in the script; (2) a witnessing trace in $\mathcal{I}$. For each unsatisfiable query, the model checker may return a *proof certificate* for that query's unsatisfiability.

The interpretation $\mathcal{I}$ *must be the same* for all queries in the same `:queries` attribute. In contrast, queries in different attributes may each interpret the free symbols differently. Regardless of where it occurs, each query may have its own witnessing trace.

### 3.3 System Checking Response

MoXI also defines the content and format of possible responses (from the back-end model checker) to a `check-system` command. Witness traces returned by the model checker are currently limited to *lasso* traces, that is, traces of the form $pl^\omega$, where $p$ and $l$ are finite sequences of state, or *trails*. Each witness is then represented by two trails: (1) a *prefix trail* $p$, and (2) a *lasso trail* $l$. In contrast, a proof certificate for a trace represents a proof of the unsatisfiability of the query. Currently, MoXI does not specify the format of proof certificates except for requiring them to be SMT-LIB S-expressions. Figure 2 shows the format for a `check-system` command with input `i`, output `o`, local variable `s`, and queries `q1`, `q2`, `q3`, where `q1` has a reachability condition `r` and fairness condition `f`.

```
(check-system-response
 :verbosity full
 :query (q1 :result sat :model m :trace t)
 :query (q2 :result unsat :certificate c)
 :query (q3 :result unknown)   ; for timeouts and other cases
 :trace (t :prefix p :lasso l) ; t = pl^w
 :model (m M)                  ; M is an interpr. in SMT-LIB format
 :trail (p ((0 (i i0) (o o0) (s s0) (r r0) (f f0)); first state in p
             ...
            (j (i ij) (o oj) (s sj) (r rj) (f fj)); last state in p
           )
        )
 :trail (l ( ( ... ) ... ( ... ) )) ; similar to p
 :certificate (c ... )
)
```

Fig. 2: Format of a model checker's response to a `check-system` command.

### 3.4 Example

As an illustrative example of a MoXI model, consider a timed switch with a single Boolean input and output where the output *switches* from its current value if the next input is true or the output has been true for at least 10 consecutive steps. Figure 3 shows a full definition of such a system in MoXI.

```
  (set-logic QF_LIA)
  (declare-enum-sort LightStatus (on off))

  (define-system TimedSwitch :input ( (press Bool) )
   :output ( (sig Bool) )
   :local ( (s LightStatus) (n Int) )
   :inv (= sig (= s on))
   :init (and (= n 0) (= s (ite press on off)))
   :trans (let (; transitions
      (turn-on  (and (= s off) press' (= s' on) (= n' n)))
      (stay-on  (and (= s on) (< n 10) (not press')
                     (= s' on) (= n' (+ n 1))))
      (turn-off (and (= s on) (or (>= n 10) press')
                     (= s' off) (= n' 0)))
      (stay-off (and (= s off) (not press') (= s' off) (= n' n)))
     )
     (or turn-on stay-on turn-off stay-off)
   )
  )

  (check-system TimedSwitch :input ( (press Bool) )
   :output ( (sig Bool) )
   :local ( (s LightStatus) (n Int) )
   :reachable (r1 (and press (not sig) (= s off)))
   :query (q1 (r1))
  )
```

Fig. 3: Example MoXI model of a timed switch with a 10-step timeout.

To start, we select the SMT-LIB logic `QF-LIA`, which restricts the language of formulas to quantifier-free formulas over linear integer arithmetic. Then, we declare an enumeration sort to represent the internal state of the switch. Next,

```
(check-system-response TimedSwitch
  :query (q1 :result sat :trace w1)
  :trace (w1 :prefix t1)
  :trail (t1 (0 (n 0) (s on) (sig true) (press true))
             (1 (n 0) (s off) (sig false) (press true)))
)
```

Fig. 4: A possible response to the `check-system` command from Figure 3.

we define the timed switch itself. This begins by declaring its input variable `press`, output variable `sig`, and local variables `s` and `n` to track, respectively, the switch's current state (on or off) and the number of steps the switch has been continuously set to `on`. In the same definition, on line 8, we define an invariant stating that `sig` is true exactly when the value of `s` is `on`. The initial condition, on line 9, sets `n` to 0 and `s` to the enumeration value corresponding to the initial value of `press`. We finish up the definition of `TimedSwitch` with the transition relation on lines 9-19, provided in *named transition* style,[8] where we define each possible transition separately and one of the transitions gets non-deterministically chosen each time. In our case, we have four possibilities based on the pre- and post-state of the transition as follows.

`turn-on`: when `s` is off in the pre-state and `press` is true in the post-state.
`stay-on`: when `s` is on and `n` $<$ 10 in the pre-state, and `press` is false in the post-state.
`turn-off`: when `s` is on in the pre-state and either `n` is at least 10 in the pre-state or `press` is false in the post-state.
`stay-off`: when `s` is off in the pre-state and `press` is false in the post-state.

Finally, on lines 21-26, we issue a check request on the system, asking whether it can reach a state where `press` is true, `sig` is false, and `s` is off. Figure 4 provides a possible response to this request. It shows a finite trace, represented as a lasso trace with an empty lasso, where the transition `turn-off` is taken immediately from an initial state (state 0) where `s` is on.

## 4 Translator Toolchain

A core application for MoXI is as an intermediate language in a toolchain integrating a high-level modeling language and a low-level representation for a model-checking back end. As Figure 5 shows, this allows users to write models in an intuitive, high-level language while also leveraging state-of-the-art algorithms provided by model checkers that accept a low-level language as input.

**Prototype Implementation** We provide a prototype Python implementation of such a toolchain, using SMV and Btor2 as the high-level and low-level languages, respectively [37]. We selected SMV due to its ubiquity in modeling symbolic transition systems [7, 8, 13, 21, 22, 27, 28, 31, 38, 41, 44, 45, 49, 54–57] and

[8] MoXI's syntax also supports other styles, such as equational or condition-action.

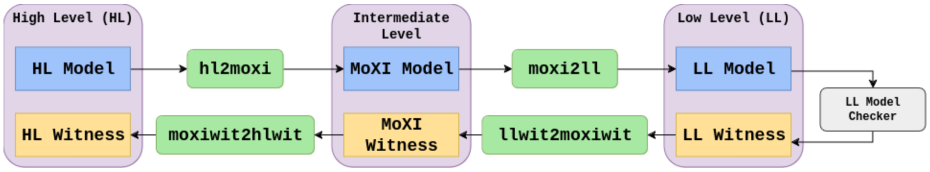

Fig. 5: The abstract translator toolchain provides translators from a high-level model to low-level model via MoXI, and back for generated witnesses. Blue denotes models and their queries, green denotes a translator, and yellow denotes a witness. Examples of high-level languages include SMV and Verilog; low-level languages include Btor2 and Aiger. All blue boxes are behaviorally equivalent, as are all yellow boxes.

Btor2 due to it being the input language for the most-recent 2020 Hardware Model Checking Competition [6]. The toolchain translates SMV models to behaviorally equivalent MoXI models, and subsequently to Btor2 models. Then, the tool translates responses from a Btor2 model checker back to SMV-style responses via MoXI responses.

**Experimental Evaluation** To evaluate the effectiveness of our toolchain, we ran it over the set of 960 `QF_BV` and `QF_ABV` benchmarks provided with the nuXmv public release and compared the performance of a variety of back-end model checkers: AVR [30] (`hwmcc20` GitHub branch), nuXmv [12] (`version 2.0` — latest public release), and Pono [40] (`commit #b243cef` — latest development head).

We ran two experiments: the first ran each solver using IC3-based [9] algorithms and the second ran each solver using a portfolio approach of BMC, K-Induction, and IC3-based algorithms. In the latter case, we collected the best time among the three techniques. For each experiment, we ran nuXmv directly on the SMV benchmarks, i.e., SMV → nuXmv, and ran the other two solvers on Btor2 generated from the toolchain presented above, i.e., SMV → MoXI → Btor2 → AVR/Pono and back. Here are some details about each tool.

- AVR (Yices2 [23] as the back-end SMT solver): BMC [5], K-induction based on [52], IC3 based on [29].
- nuXmv (MathSAT5 [16] as the back-end SMT solver): BMC [5], K-induction based on [25], and IC3 based on [15].
- Pono (Boolector [46] and MathSAT5 [16] as back-end SMT solvers): BMC [5], K-induction based on [25], IC3 based on [15].

In all cases, no discrepancies were found, i.e., no two model checkers returned conflicting *safe* and *unsafe* results, and all generated Btor2 was well-formed according to the reference checker catbtor [46].

As the top of Figure 6 shows, the three model checkers nuXmv, AVR, and Pono complement each other when using their IC3-based algorithms; each solver performs similarly, but the virtual-best outperforms each individual by a noticeable margin. Similarly, the bottom of Figure 6 shows that while the model

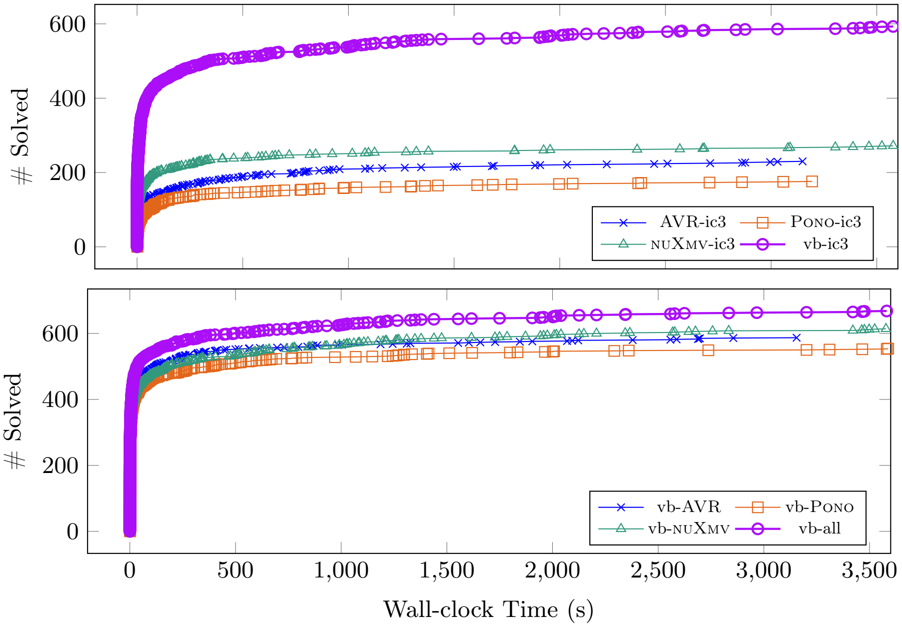

Fig. 6: Performance comparison on SMV-language benchmark queries using IC3 (Top) and portfolio approach (Bottom) across three different model checkers: SMV benchmarks → nuXmv, a translation of SMV benchmarks → MoXI → Btor2 → AVR (and back), and a translation of SMV benchmarks → MoXI → Btor2 → Pono (and back). The vb-* represents the virtual-best back-end solver for each model checker. The portfolio approach represents the best time using BMC, k-induction, and IC3 algorithms. Wall-clock time for the non-nuXmv plots include translation time.

checkers perform similarly using a portfolio approach, they again complement each other, as shown by the virtual-best outperforming each model checker individually. Importantly, our initial, non-optimized translation of SMV-language benchmarks through MoXI does not inhibit the model checking performance of either AVR or Pono, compared to using nuXmv directly on the same SMV-language benchmarks distributed with that tool.

## 5 Future Directions for Symbolic Model Checking

Creating an international standard for symbolic model checking changes the landscape of research next steps, from low-hanging fruit to future challenges. There is even more reason to continue and expand on current research directions aimed at breaking down the barriers to the wider adoption of symbolic model checking by addressing (1) the specification bottleneck; and (2) the state-space explosion problem. Having laid a foundation for standardization, we hope that the community can converge on MoXI as a common intermediate language and leverage this standard in pursing research directions that make advances in areas (1) and (2).

*High-level language translation.* In addition to creating translations for existing high-level modeling languages to/from MoXI, there is now an opportunity to create new languages that take advantage of the access to back-end algorithms that MoXI provides. Current model-checking high-level languages were designed to be as general as possible, to represent a reasonably broad class of systems since usually each model-checking tool accepts only one modeling language. Now, there is more room for highly-specialized, system-specific languages, further mitigating the specification bottleneck.

*Low-level language translation.* To loop in current and future model-checking back-end algorithms, we need translations between MoXI and the low-level representations used by model-checking tools. Future algorithms may be designed with such a translation in mind. Though it was designed as an intermediate language, MoXI may be sufficiently low-level to serve as the input representation for future back-end tools; this is another avenue worth investigating.

*Translation optimizations.* We provide initial translations between the high-level SMV language and MoXI, and between MoXI and the low-level representation Btor2 [37]. However, these are just proofs of concept. They demonstrate that a translation is possible and that the design of MoXI maintains the expressiveness we intended. These translations were not designed to be optimal, or even efficient, only correct and, hopefully, transparent. Our initial toolchain encodes LTL specifications by using PANDA [50] to translate them into SMV, then translating the PANDA-generated SMV models to MoXI; exploring direct LTL-to-MoXI translations could improve model-checking performance. Section 4 demonstrates that model checking through MoXI does not exacerbate the state-space explosion problem, but there is certainly ample room for improvement. We expect a progression of future papers creating increasingly performant translations, improving upon our translations and those contributed by others (see above).

*Proofs and benchmarks.* While we have extensively investigated the correctness of our initial MoXI translations, we have yet to prove them correct formally, for instance by using an interactive theorem prover. We believe it is possible, though challenging, to state the semantic equivalence of representations in a high- or low-level language and MoXI as theorems, prove them correct using a theorem prover like PVS, and then generate verified translators using a tool like PVS2C [18,26].[9] Lower-hanging fruit involves creating, packaging, and releasing benchmarks for MoXI translations that help others check their new translations and serve as performance checkpoints for translation tools.

[9] Thanks to Laura Gamboa Guzman, Katherine Kosaian, and Yi Lin for their initial investigations into this possibility.

*Extensions.* Though we have initially addressed hardware model checking of finite-state systems, MoXI is extensible by design. Future research directions include further extending MoXI representations to infinite-state model checking, investigating efficient representations for highly-expressive high-level modeling languages, and even exploring the uses of MoXI for applications in software model checking.[10]

## 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. 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)
4. Beer, I., Ben-David, S., Eisner, C., Landver, A.: Rulebase: An industry-oriented formal verification tool. In: Design Automation Conference. pp. 655–660. IEEE Computer Society (1996)
5. 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
6. Biere, A., Froleyks, N., Preiner, M.: Hardware Model Checking Competition (HWMCC). https://fmv.jku.at/hwmcc20/index.html (2020)
7. 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)
8. 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)
9. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: VMCAI. pp. 70–87 (2011)
10. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: International Conference on Computer Aided Verification. pp. 24–40. Springer (2010)
11. 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)
12. 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)

[10] Thanks to Dirk Beyer for initial ideas in this direction.

13. Choi, Y., Heimdahl, M.: Model checking software requirement specifications using domain reduction abstraction. In: IEEE ASE. pp. 314–317 (2003)
14. 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)
15. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Ic3 modulo theories via implicit predicate abstraction. In: Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20. pp. 46–61. Springer (2014)
16. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: TACAS. pp. 93–107 (2013)
17. Colaço, J.L., Pagano, B., Pouzet, M.: Scade 6: A formal language for embedded critical software development. In: 2017 International Symposium on Theoretical Aspects of Software Engineering (TASE). pp. 1–11. IEEE (2017)
18. Courant, N., Séré, A., Shankar, N.: The correctness of a code generator for a functional language. In: Beyer, D., Zufferey, D. (eds.) Verification, Model Checking, and Abstract Interpretation - 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16-21, 2020, Proceedings. Lecture Notes in Computer Science, vol. 11990, pp. 68–89. Springer (2020). https://doi.org/10.1007/978-3-030-39322-9_4, https://doi.org/10.1007/978-3-030-39322-9_4
19. Documentation, S.: Simulation and model-based design (2020), https://www.mathworks.com/products/simulink.html
20. Documentation, SCADE: Ansys SCADE Suite (2023), https://www.ansys.com/products/embedded-software/ansys-scade-suite
21. 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
22. 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)
23. Dutertre, B.: Yices 2.2. In: International Conference on Computer Aided Verification. pp. 737–744. Springer (2014)
24. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD. pp. 125–134 (2011)
25. Eén, N., Sörensson, N.: Temporal induction by incremental sat solving. Electronic Notes in Theoretical Computer Science **89**(4), 543–560 (2003)
26. Férey, G., Shankar, N.: Code generation using a formal model of reference counting. In: Rayadurgam, S., Tkachuk, O. (eds.) NASA Formal Methods: 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings. pp. 150–165. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_12, http://dx.doi.org/10.1007/978-3-319-40648-0_12
27. 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
28. 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

29. Goel, A., Sakallah, K.: Model checking of verilog rtl using ic3 with syntax-guided abstraction. In: NASA Formal Methods: 11th International Symposium, NFM 2019, Houston, TX, USA, May 7–9, 2019, Proceedings 11. pp. 166–185. Springer (2019)
30. 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)
31. 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)
32. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
33. IEEE: IEEE standard multivalue logic system for VHDL model interoperability (Std_logic_1164) (1993)
34. IEEE: IEEE standard for Verilog hardware description language (2005)
35. IEEE: IEEE standard for VHDL language reference manual (2019)
36. Jahier, E., Raymond, P., Halbwachs, N.: The lustre v6 reference manual. Verimag, Grenoble, Dec (2016)
37. 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: Proceedings of 36th International Conference on Computer Aided Verification (CAV). LNCS, Springer (July 2024)
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.: The SMV language. Tech. rep., Cadence Berkeley Lab (1999)
43. McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
44. 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
45. Miller, S.P., Tribble, A.C., Whalen, M.W., Per, M., Heimdahl, E.: Proving the shalls. STTT **8**(4-5), 303–319 (2006)
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. R. Cavada, A.C., Jochim, C., Keighren, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: NuSMV 2.4 user manual. Tech. rep., CMU/ITC-irst (2005)
49. Raimondi, F., Lomuscio, A., Sergot, M.J.: Towards model checking interpreted systems. In: FAABS 02, LNAI 2699. pp. 115–125. Springer (2002)
50. 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)
51. Rozier, K.Y.: Specification: The biggest bottleneck in formal methods and autonomy. In: Proceedings of 8th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2016). LNCS, vol. 9971, pp. 1–19. Springer-Verlag, Toronto, ON, Canada (July 2016). https://doi.org/10.1007/978-3-319-48869-1_2
52. 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)
53. SMTLib. https://smtlib.cs.uiowa.edu/
54. 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)
55. Yoo, J., Jee, E., Cha, S.: Formal modeling and verification of safety-critical software. Software, IEEE **26**(3), 42–49 (2009)
56. 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)
57. 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)