

MoXI — A Model Checking Intermediate Language. (Draft)

Cesare Tinelli

2024-02-01

MoXI, which stands for Model Exchange Interlingua, is an intermediate language meant to be a common input and output standard for model checkers for finite- and infinite-state systems, with an initial focus is on finite-state ones.

General Design Philosophy

MoXI has been 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 meant to be simple enough to be easily compilable to lower level languages or be directly supported by model checking tools based on SAT/SMT technology.

For being an intermediate language, models expressed in MoXI are meant to be produced and processed by tools, hence it was designed to have

- simple, easily (machine) 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;
- simple translations to lower level modeling languages such as Btor2.

Based on these principles, MoXI provides no direct support for many of the features offered by current hardware modeling languages such as VHDL and Verilog or more general purpose system modeling languages such as SMV, TLA+, PROMELA, Simulink, SCADE, Lustre. However, it strives to offer enough capability so that problems expressed in those languages can be reduced to problems in MoXI.

MoXI is an extension the SMT-LIB language 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.

MoXI assumes a discrete and linear notion of time and hence has a standard trace-based semantics.

Each system definition:

- defines a *transition system* via the use of SMT formulas, imposing minimal syntactic restrictions on those formulas;
- is parametrized by a *state signature*, a sequence of typed variables;
- partitions state variables into input, output and local variables;
- can be expressed as the synchronous or asynchronous composition of other systems
- is *hierarchical*, i.e., may include (instances of) previously defined systems as subsystems;
- can encode both synchronous and asynchronous system composition.

The current focus on *finite-state* systems. However, the language has been designed to support the specification of infinite-state system as well.

Technical 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.

Notation

If F is a formula and $x = (x_1, \dots, x_n)$ a tuple of distinct variables, we write $F[x]$ or $F[x_1, \dots, x_n]$ to express the fact that every variable in x is free in F (although F may have additional free variables). We write $x \circ y$ to denote the concatenation of tuple x with tuple y . When it is clear from the context, given a formula $F[x]$ and a tuple $t = (t_1, \dots, t_n)$ of terms of the same type as $x = (x_1, \dots, x_n)$, we write $F[t]$ or $F[t_1, \dots, t_n]$ to denote the formula obtained from F by simultaneously replacing each occurrence of x_i by t_i for all $i = 1, \dots, n$.

A formula may contain *uninterpreted* constant and function symbols, that is, symbols with no constraints on their interpretation. For most purposes, we treat uninterpreted constants as free variables and treat uninterpreted function symbols as *second-order* free variables.

Transition systems

Formally, a transition system S is a pair of predicates of the form

$$S = (I_S[i, o, s], T_S[i, o, s, i', o', s'])$$

where

- i and i' are two tuples of *input variables* with the same length and type;
- o and o' are two tuples of *output variables* with the same length and type;
- s and s' are two tuples of *local variables* with the same length and type;
- I_S is the system's *initial state condition*, expressed as a formula with no (free) variables from i' , o' , and s' ;
- T_S is the system's *transition condition*, expressed as a formula over the variables $i, o, s, i', o',$ and s' .

Note: For convenience, but differently from other formalizations, a (full) state of system S is expressed by a valuation of the variables i, o, s . In other words, input and output variables are automatically *stateful* since the transition relation formula can access old values of inputs and outputs in addition to the old values of the local state. This means that, technically, S is a closed system. The designation of some state variables as input or output is, however, important when combining systems together, to capture which of the state values are shared between two systems being combined, and how.

Note: Similarly to Mealy machines, the initial state condition is also meant to specify the initial system's output, based on the initial state and input. Correspondingly, the transition relation is also meant to specify the system's output in every later state of the computation.

Note: The input and output values corresponding to the transition to the *new* state are those in the variables i' and o' . The values of i, o are the *old* input and output values.

Trace Semantics

A transition system in the sense above 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 (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 `Int`, `Real`, `Array(Int, Real)`),

BitVec(3), ...) is fixed by the theory \mathcal{T} in question. Once a theory \mathcal{T} has been fixed then, 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 .

More precisely, let us fix a tuple $x = (x_1, \dots, x_n)$ of distinct *state* variables, meant to represent the state of a computation system. We will denote by x' the tuple (x'_1, \dots, x'_n) and write formulas of the form $F[f, x, x']$ where f is a tuple of uninterpreted symbols. If F has free occurrences of variables from x but not from x' we call it a *one-state* formula; otherwise, we call it a *two-state* formula.

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

Infinite-Trace Semantics

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

- F is atomic and $\mathcal{J}[x \mapsto \pi[0](x), x' \mapsto \pi[1](x)]$ satisfies F as in FOL;
- $F = \neg G$ and $(\mathcal{J}, \pi) \not\models G$;
- $F = G_1 \wedge G_2$ and $(\mathcal{J}, \pi) \models G_i$ for $i = 1, 2$;
- $F = \exists x G$ and $(\mathcal{J}[z \mapsto v], \pi) \models G$ for some value v for x ;
- $F = \textbf{eventually } G$ and $(\mathcal{J}, \pi^i) \models G$ for some $i \geq 0$;
- $F = \textbf{always } G$ and $(\mathcal{J}, \pi^i) \models G$ 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 is a *static*, or *rigid*, quantifier: the meaning of the variable it quantifies does not change over time, that is, from state to state in π . The same is true for uninterpreted symbols. They are *rigid* in the same sense: their meaning does not change over time. 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.

Now let

$$S = (I_S[i, o, s], T_S[i, o, s, i', o', s'])$$

be a transition system with state variables i, o, s .

The *infinite trace semantics* of S is the set of all pairs (\mathcal{J}, π) of interpretations \mathcal{J} in \mathcal{T} and infinite traces π such that

$$(\mathcal{J}, \pi) \models I_S \wedge \text{always } T_S$$

We call any such pair an *execution* of S .

Note: [We focus on reactive systems]

Finite-Trace Semantics

Let $F[f, x, x']$, \mathcal{J} , \mathcal{T} and π be defined as in the subsection above. For every $n \geq 0$, (\mathcal{J}, π) *n-satisfies* F , written $(\mathcal{J}, \pi) \models_n F$, iff

- F is atomic and $\mathcal{J}[x \mapsto \pi[0](x), x' \mapsto \pi[1](x)]$ satisfies F as in FOL;
- $F = \neg G$ and $(\mathcal{J}, \pi) \not\models_n G$
- $F = G_1 \wedge G_2$ and $(\mathcal{J}, \pi) \models_n G_i$ for $i = 1, 2$;
- $F = \exists x G$ and $(\mathcal{J}[z \mapsto v], \pi) \models_n G$ for some value v for x ;
- $F = \text{eventually } G$ and $(\mathcal{J}, \pi^i) \models_{n-i} G$ for some $i = 0, \dots, n$;
- $F = \text{always } G$ and $(\mathcal{J}, \pi^i) \models_{n-i} G$ for all $i = 0, \dots, n$;

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

Intuitively, n -satisfiability specifies when a formula is true over the first n states of a trace. Note that this notion is well defined even when $n = 0$ regardless of whether F has free occurrences of variables from x' or not. In the atomic case, this is true because π , for being an *infinite* trace, does contain the state $\pi[1]$. In the general case, the claim can be shown by a simple inductive argument.

The notion of n -satisfiability is useful when one is interested, as we are, in state reachability. The reason is that 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, however, that the converse does not hold. That is, it is possible for R to be reachable in a system S without being n -satisfied by an execution of S . See Section [todo](#) for more details.

Supported SMT-LIB commands

SMT-LIB is a command-based language with LISP-like syntax (s-expressions, in prefix notation) designed to be a common input/output language for SMT solvers.

MoXI adopts the following SMT-LIB commands:

- (declare-sort s n)
Declares s to be a sort symbol (i.e., type constructor) of arity n . Examples:

- ```
(declare-sort A 0)
(declare-sort Set 1)
; possible sorts: A, (Set A), (Set (Set A)), (Array Int Real), ...
```
- (define-sort  $s$  (  $u_1 \dots u_n$  )  $\tau$ )

Defines  $S$  as synonym of a parametric type  $\tau$  with parameters  $u_1 \dots u_n$ .  
Examples:

```
(declare-sort NestedSet (X) (Set (Set X)))
; possible sorts: (NestedSet A), ...
(declare-sort Array2 (X Y) (Array X (Array X Y)))
; possible sorts: (Array2 Int Bool), ...
```
- (declare-const  $c$   $\sigma$ )

Declares a constant  $c$  of sort  $\sigma$ . Examples:

```
(declare-const a A)
(declare-const n Int)
```
- (define-fun  $f$  ( (  $x_1 \sigma_1$  )  $\dots$  (  $x_n \sigma_n$  ) )  $\sigma$   $t$ )

Defines a (non-recursive) function  $f$  with inputs  $x_1, \dots, x_n$  (of respective sort  $\sigma_1, \dots, \sigma_n$ ), output sort  $\sigma$ , and body  $t$ . Examples:

```
(declare-fun sq ((n Int)) Int (* n n))
(declare-fun isSqRoot ((m Int) (n Int)) Bool (= n (sq m)))
(declare-fun max ((m Int) (n Int)) Int (ite (> m n) m n))
```
- (set-logic  $L$ )

Defines the model's *data logic*, that is, 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.).

```
(set-logic QF_BV)
```

One addition to SMT-LIB is the binary symbol `!=` for disequality. For each term  $s$  and  $t$  of the same sort, `(!= s t)` has the same meaning as `(not (= s t))` or, equivalently, `(distinct s t)`.

## MoXI-specific commands

### Enumeration declaration

```
(declare-enum-sort s ($c_1 \dots c_n$))
```

Declares  $s$  to be an enumerative type with (distinct) values  $c_1, \dots, c_n$ .

### System definition command

**Atomic systems** An atomic transition system is defined by a command of the form:

(define-system  $S$  :input ( (  $i_1 \delta_1$  )  $\cdots$  (  $i_m \delta_m$  ) ) :output ( (  $o_1 \tau_1$  )  $\cdots$  (  $o_n \tau_n$  ) ) :local ( (  $s_1 \sigma_1$  )  $\cdots$  (  $s_p \sigma_p$  ) ) :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  $s_j$  is a *local* variable of sort  $\sigma_j$ ;
- all variables above
- each  $i_j, o_j, s_j$  denote *current-state* values
- *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  $s_j$ ;
- $I$ , 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  $S$ ;
- all attributes are optional but can occur at most once;
- 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.

**Discussion:** We could allow multiple occurrences of the :inv attribute with conjunctive semantics. Should we allow multiple occurrences of the :trans attribute but with *disjunctive* semantics? The rationale would be to facilitate the recognition of systems defined by alternative sets of transitions.

**Atomic System Semantics** Let  $i = (i_1, \dots, i_m)$ ,  $o = (o_1, \dots, o_n)$ ,  $s = (s_1, \dots, s_p)$ , and  $v = i, o, s$ .

Formally, an atomic system  $S$  introduced by the define-system command above is a transition system whose behavior consists of all the (infinite) executions  $(\mathcal{J}, \pi)$  over  $v$  such that

$$(\mathcal{J}, \pi) \models I[v] \wedge \mathbf{always} (P[v] \wedge T[v, v']) .$$

We call  $I_S = I[v]$  the *initial state predicate* of  $S$  and  $T_S = P[v] \wedge T[v, v']$  the *transition predicate* of  $S$ .

**Note:** The relation expressed by the formula  $T$  is not required to be functional over  $i, o, s, i'$ , thus allowing the modeling of non-deterministic systems.

**Note:** The `:inv` attribute is not strictly necessary since a system with a declaration of the form

```
(define-system S :input ((i1 σ1) ... (im σm)) :output ((o1 τ1)
... (on τn)) :local ((s1 σ1) ... (sp σp)) :init I :trans T :inv
P)
```

can be equivalently expressed with a declaration of the form

```
(define-system S :input ((i1 σ1) ... (im σm)) :output ((o1 τ1)
... (on τn)) :local ((s1 σ1) ... (sp σp)) :init I :trans (and P
T))
```

**Note:** Systems are meant to be progressive: every reachable state has a successor with respect  $T_S$ . However, they may not be because of the generality of  $T$  and  $P$ . In other words, it is possible to define deadlocking systems. (See later for more details on deadlocked states.)

**Examples** (Adapted from “Principles of Cyber-Physical Systems” by R. Alur, 2015)

When reading these examples, it is helpful to keep in mind that, intuitively, in the `:init` formulas the input values are given and the local and output values are to be defined with respect to them. In contrast, in the `:trans` formulas the new input values, and old input, output and local values are given, and the new local and output values are to be defined.

The output of system `Delay` below is initially 0 and then is the previous input. No local variables are needed.

```
(define-system Delay :input ((i Int)) :output ((o Int))
:init (= o 0)
:trans (= o' i) ; the new output is the old input
)
```

A variant of `Delay` where the output is initially any number in  $[0,10]$ .

```
(define-system Delay :input ((i Int)) :output ((o Int))
:init (<= 0 o 10) ; more than one possible initial output
:trans (= o' i)
)
```



A clocked lossless channel, stuttering when the clock is not ticking. The clock is represented by a Boolean input variable clock.

```
(define-system StutteringClockedCopy
 :input ((clock Bool) (i Int))
 :output ((o Int))
 :init (=> clock (= o i)) ; o is arbitrary when clock is false
 :trans (ite clock (= o' i') (= o' o)) ; ite is if-then-else
)
```

Events carrying data can be modeled as instances of the polymorphic algebraic datatype (Event X) where X is the type of the data carried by the event.

```
(declare-datatype Event (par (X) (
 (absent)
 (present (val X))
)))
```

An event-triggered channel that arbitrarily loses its input data.

```
(define-system LossyIntChannel
 :input ((i (Event Int)))
 :output ((o (Event Int)))
 :inv (or (= o i) (= o absent))
)
```

Equivalent formulation using unconstrained local state.

```
(define-system LossyIntChannel
 :input ((i (Event Int)))
 :output ((o (Event Int)))
 :local ((s Bool))
 ; at all times, whether the input event is transmitted
 ; or not depends on value of s
 :inv (= o (ite s i absent))
)
```

TimedSwitch models a timed light switch where the light stays on for at most 10 steps unless it is switched off before. The input Boolean is interpreted as an on/off toggle. The transition predicate is formulated as a set of transitions.

```
(declare-enum-sort LightStatus (on off))

(define-system TimedSwitch1
 :input ((press Bool))
 :output ((sig Bool))
 :local ((s LightStatus) (n Int))
 :inv (= sig (= s on))
 :init (and
 (= n 0)
```

```

 (ite press (= s on) (= s off))
)
:trans (let
 (; transitions
 (stay-off (and (= s off) (not press') (= s' off) (= n' n)))
 (turn-on (and (= s off) press' (= s' on) (= n' n)))
 (stay-on (and (= s on) (not press') (< n 10) (= s' on) (= n' (+ n 1))))
 (turn-off (and (= s on) (or press' (>= n 10)) (= s' off) (= n' 0)))
)
 (or stay-off turn-on turn-off stay-on)
)
)

```

A variant of the system above where the transition predicate is formulated guarded-transitions style.

```

(define-system TimedSwitch2
:input ((press Bool))
:output ((sig Bool))
:local ((s LightStatus) (n Int))
:inv (= sig (= s on))
:init (and
 (= n 0)
 (ite press (= s on) (= s off))
)
:trans (and
 (=> (and (= s off) (not press'))
 (and (= s' off) (= n' n))) ; off --> off
 (=> (and (= s off) press')
 (and (= s' on) (= n' n))) ; off --> on
 (=> (and (= s on) (not press') (< 10 n))
 (and (= s' on) (= n' (+ n 1)))) ; on --> on
 (=> (and (= s on) (or press' (>= n 10)))
 (and (= s' off) (= n' 0))) ; on --> off
)
)

```

Another variant but in equational style.

```

(define-fun flip ((s LightStatus)) LightStatus
 (ite (= s off) on off)
)

```

```

(define-system TimedSwitch3
: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 (and
 (= s' (ite press' (flip s)
 (ite (or (= s off) (>= n 10)) off
 on)))
 (= n' (ite (or (= s off) (= s' off)) 0
 (+ n 1)))
)
)

```

The non-deterministic arbiter below grants input requests expressed by the Boolean inputs r1 and r2. A request is always granted, expressed by the Boolean outputs g1 or grant2, if it is the only request. When both inputs contain a request, one of the two request is granted, with a non-deterministic choice. Since the output depends only on the current values of input and local variables, the systems can be specified simply by an invariant.

```

(define-system NonDetArbiter
:input ((r1 Bool) (r2 Bool))
:output ((g1 Bool) (g2 Bool))
:local ((s Bool))
:inv (and
 (=> (and (not r1) (not r2))
 (and (not g1) (not g2)))
 (=> (and r1 (not r2))
 (and g1 (not g2)))
 (=> (and (not r1) r2)
 (and (not g1) g2))
 (=> (and r1 r2)
 ; the unconstrained value of `s` is used as non-deterministic choice
 (ite s (and g1 (not g2))
 (and (not g1) g2)))
)
)

```

The next arbiter is similar to NonDetArbiter but grants requests a cycle later and does not use a local variable for the non-deterministic choice.

```

(define-system DelayedArbiter
:input ((r1 Bool) (r2 Bool))
:output ((g1 Bool) (g2 Bool))
:local ((s Bool))
:init (and (not g1) (not g2)) ; nothing is granted initially
:trans (and
 (=> (and (not r1) (not r2))

```

```

 (and (not g1') (not g2'))))
(=> (and r1 (not r2))
 (and g1' (not g2'))))
(=> (and (not r1) r2)
 (and (not g1') g2'))
(=> (and r1 r2)
 (!= g1' g2'))
)
)

```

Similar to DelayedArbiter but for requests expressed as integer events.

```

(define-system IntNonDetArbiter
 :input ((r1 (Event Int)) (r2 (Event Int)))
 :output ((g1 (Event Int)) (g2 (Event Int)))
 :local ((s Bool))
 :init (= g1 g2 absent)
 :trans (and
 (=> (= r1' r2' absent)
 (= g1' g2' absent))
 (=> (and (!= r1' absent) (= r2' absent))
 (and (= g1' r1) (= g2' absent)))
 (=> (and (= r1' absent) (!= r2' absent))
 (and (= g1' absent) (= g2' r2')))
 (=> (and (!= r1' absent) (!= r2' absent))
 (or (and (= g1' r1') (= g2' absent))
 (and (= g1' absent) (= g2' r2')))))
)
)

```

**Composite Systems - synchronous composition** A transition systems can be defined as the synchronous composition of other systems by a command of the form:

```

(define-system S :input ((i1 σ1) ... (im σm)) :output ((o1 τ1) ... (on τn))
 :local ((s1 σ1) ... (sp σp)) :subsys (N1 (S1 x1 y1)) ... :subsys (
 Nq (Sq xq yq)) :init I :trans T :inv P)

```

where

- :input, :output, :local :init, :trans, and :inv are as in atomic system definitions;
- $q > 0$  and each  $S_i$  is the name of a system other than  $S$ ;
- the names  $S_1 \dots, S_q$  need not be all distinct;
- each  $N_i$  is a local synonym for  $S_i$ , with  $N_1, \dots, N_q$  distinct;
- each  $x_i$  consists of  $S$ 's variables of the same type as  $S_i$ 's input;
- each  $y_i$  consists of  $S$ 's local/output variables of the same type as  $S_i$ 's output;

- the directed subsystem graph rooted at  $S$  is acyclic.

**Composite System Semantics** For  $k = 1, \dots, q$ , let  $S_k = (I_k[i_k, o_k, s_k], T_k[i_k, o_k, s_k, i'_k, o'_k, s'_k])$ , with the elements of  $s_1, \dots, s_q$  all mutually distinct.

Let  $i = (i_1, \dots, i_m)$ ,  $o = (o_1, \dots, o_n)$ ,  $s = (s_1, \dots, s_p) \circ s_1 \circ \dots \circ s_q$ , and  $v = i \circ o \circ s$ .

Formally, a composite system  $S$  introduced by the define-system command above is a transition system whose behavior consists of all the (infinite) executions  $(\mathcal{J}, \pi)$  over  $v$  such that

$$(\mathcal{J}, \pi) \models I_S[v] \wedge \mathbf{always} T_S[v, v']$$

where

- $I_S[v] = I[v] \wedge \bigwedge_{k=1, \dots, q} I_k[x_k, y_k, s_k]$  and
- $T_S[v, v'] = P[v] \wedge T[v, v'] \wedge \bigwedge_{k=1, \dots, q} T_k[x_k, y_k, s_k, x'_k, y'_k, s'_k]$

### Examples, composite systems

```
;-----
; Two-step delay
;-----
```

```
; +-----+
; | DoubleDelay |
; | +-----+ +-----+ |
; in | | | temp | | | out
; ---+-->| Delay |---->| Delay |-->
; | | | | | |
; | +-----+ +-----+ |
; +-----+
```

```
; One-step delay
(define-system Delay :input ((i Int)) :output ((o Int))
 :local ((s Int))
 :inv (= s i)
 :init (= o 0)
 :trans (= o' s)
)
```

```
; Two-step delay
(define-system DoubleDelay :input ((in Int)) :output ((out Int))
 :local ((temp Int))
 :subsys (D1 (Delay in temp))
 :subsys (D2 (Delay temp out))
)
```

```

)

;; DoubleDelay expanded
(define-system DoubleDelay
 :input ((in Int))
 :output ((out Int))
 :local (
 (temp Int)
 (s1 Int) ; from `(Delay in temp)`
 (s2 Int) ; from `(Delay temp out)`
)
 :inv (and
 (= s1 in) ; from `(Delay in temp)`
 (= s2 temp) ; from `(Delay temp out)`
)
 :init (and
 (= temp 0) ; from `(Delay in temp)`
 (= out 0) ; from `(Delay temp out)`
)
 :trans (and
 (= temp' s1) ; from `(Delay in temp)`
 (= out' s2) ; from `(Delay temp out)`
)
)

; Example trace:
; in = 1, 2, 3, 4, 5, 6, 7, ...
; s1 = 1, 2, 3, 4, 5, 6, 7, ...
; temp = 0, 1, 2, 3, 4, 5, 6, ...
; s2 = 0, 1, 2, 3, 4, 5, 6, ...
; out = 0, 0, 1, 2, 3, 4, 5, ...

```

The next example defines a three-bit counter in terms of three identical one-bit counters. The one-bit counter uses a latch. The latch has a Boolean state represented by state variable *s* with arbitrary initial value. The value of output *out* is always the previous value of *s*. A set request (represented by input *set* being true) sets the new value of *s* to true unless there is a concurrent reset request (represented by input *reset* being true). In that case, the choice between the two requests is resolved arbitrarily using the value of the unconstrained local variable *b*. In the absence of either a set or a reset, the value of *out* is unchanged.

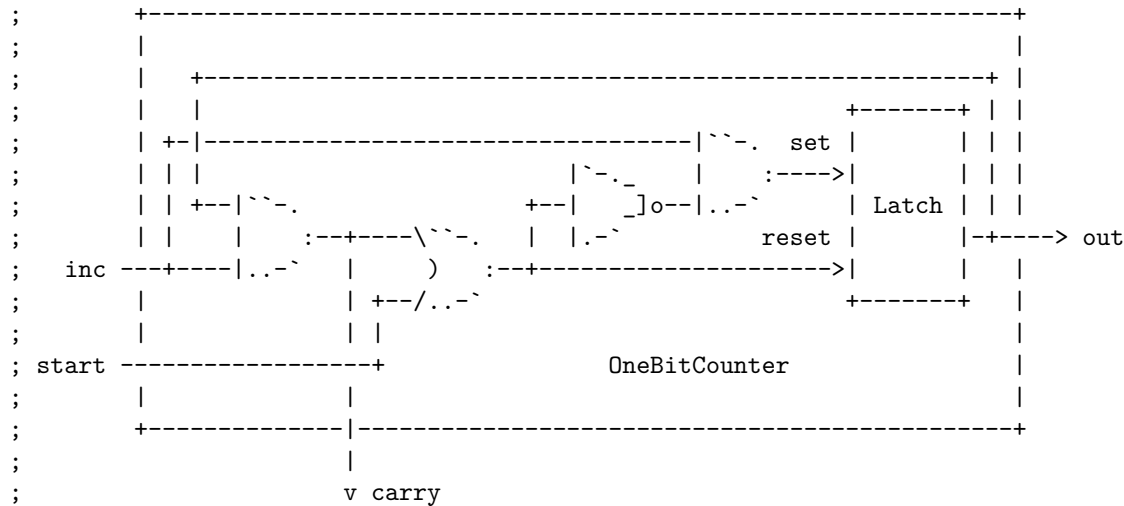
```

(define-system Latch :input ((set Bool) (reset Bool)) :output ((out Bool))
 :local ((s Bool) (b Bool))
 :init (and
 (= out b)
)
 :trans (and
 (= out' s)
)
)

```

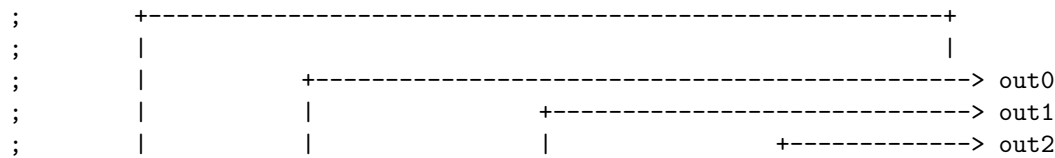
```
(= s' (or (and set (or (not reset) b))
 (and (not set) (not reset) out)))
)
```

The one-bit counter is implemented using the latch component modeled by **Latch**. The counter goes from 0 (represented as false) to 1 (true) with a carry value of 0, or from 1 to 0 with a carry value of 1 when the increment signal `inc` is true. It is reset to 0 (false) when the start signal is true. The initial value of the counter is arbitrary.



```
(define-system OneBitCounter :input ((inc Bool) (start Bool))
:output ((out Bool) (carry Bool))
:local ((set Bool) (reset Bool))
:subsys (L (Latch set reset out))
:inv (and
 (= set (and inc (not reset)))
 (= reset (or carry start))
 (= carry (and inc out))
)
)
```

The three-bit counter is a resettable counter obtained by cascading three 1-bit counters. The output is three Boolean values standing for the three bits, with out0 being the least significant one.



```

; | | | |
; | +-----+ +-----+ +-----+ |
; | | car0 | | car1 | | car2 | |
; inc ----->| OneBit |----->| OneBit |----->| OneBit |----->|
; | | Counter | | Counter | | Counter | |
; start ----->| | +-->| | +-->| | +-->| |
; | | +-----+ | +-----+ | +-----+ |
; | +-----+-----+
;
;
;
(define-system ThreeBitCounter :input ((inc Bool) (start Bool))
:output ((out0 Bool) (out1 Bool) (out2 Bool))
:local ((car0 Bool) (car1 Bool) (car2 Bool))
:subsys (C1 (OneBitCounter inc start out0 car0))
:subsys (C2 (OneBitCounter car0 start out1 car1))
:subsys (C3 (OneBitCounter car1 start out2 car2))
)

```

**Sanity requirements on  $I_S$  and  $T_S$**  Because of the infinite trace semantics every system defined in MoXI is expected to execute forever. This is not a limitation in practice because systems that are meant to reach a final state can be always 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.

Intuitively, for a system definition to define a system with no deadlocks:

1. Any assignment of values to the input variables can be extended to a total assignment (to all the unprimed variables) that satisfies  $I_S$ .
2. For any reachable state  $S$ , any assignment  $s$  to the primed input variables can be extended to a total assignment  $s'$  so that  $s, s'$  satisfies  $T_S$ .

The first restriction above guarantees that the system can start at all. The second ensures that from any reachable state and for any new input the system can move to another state (*and so* also produce output).

- A sufficient condition for (1) is that the following formula is valid in the (previously specified) background theory:

$$\forall i \exists o \exists s I_S$$

- A sufficient condition for (2) is that the following formula is valid in the background theory:

$$\forall i \forall o \forall s \forall i' \exists o' \exists s' T_S$$



This condition is not necessary, however, since it need not apply to unreachable states. Let  $\text{Reachable}[i, o, s]$  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:

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

**Note:** In general, checking the two sufficient conditions above automatically can be impossible or very expensive because of the quantifier alternations in the conditions.

### System checking command

(check-system  $S$  :input ( (  $i_1 \delta_1$  )  $\cdots$  (  $i_m \delta_m$  ) ) :output ( (  $o_1 \tau_1$  )  $\cdots$  (  $o_n \tau_n$  ) )  
:local ( (  $s_1 \sigma_1$  )  $\cdots$  (  $s_p \sigma_p$  ) ) :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 previously defined system with  $m$  inputs,  $n$  outputs, and  $p$  local variables;
- all attributes are optional and their order is immaterial except that :input, :output, :local, have to come before the other attributes [may not need this restriction];
- all attributes except for :input, :output and :local are repeatable.
- $i = (i_1, \dots, i_m)$  is a renaming of the corresponding input variables of  $S$  of sort  $\delta = (\delta_1, \dots, \delta_m)$ ;
- $o = (o_1, \dots, o_n)$  is a renaming of the corresponding output variables of  $S$  of sort  $\tau = (\tau_1, \dots, \tau_n)$ ;
- $s = (s_1, \dots, s_p)$  is a renaming of the corresponding local variables of  $S$  of sort  $\sigma = (\sigma_1, \dots, \sigma_p)$ ;
- $a, r, f, c, q, q_1, \dots, q_k$  are identifiers;
- $A$  is (non-temporal) formula over the system variables  $i, o, s$ , and  $i'$  expressing an *environmental assumption* on  $i'$ ;
- $F$  is (non-temporal) formula over the system variables  $i, o, s$ , and  $i'$  expressing a *fairness condition* on  $i'$ ;
- $R$  is (non-temporal) formula over all the system variables, primed and unprimed, expressing a state *reachability condition*;
- $C$  is (non-temporal) formula over all the unprimed system variables 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  $g_1, \dots, g_q$ ; the same holds for each  $(q_j (g_{j,1} \cdots g_{j,n_j}))$ ;
- a query can contain more than one assumption, fairness condition and reachability condition but at most one initiality condition.

**Note:** When the command contains more than one instance of the attributes :assumption, :reachable, :fairness and :current, the list of elements of a query  $q$  can refer to any of the identifiers in those attributes.

**Note:** The order of the formula names in a query is immaterial.

**Semantics** Each query  $q$  ( $q_j$ ) 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 the finite-state semantics otherwise.

Given a check command like the above 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 command. The meaning of the query depends on its components.

Specifically, let  $t, u, v \geq 0$  :

- (1) A  $q$  query of the form (  $a_1 \dots a_t r_1 \dots r_u$  ), with each  $a_j$  naming an assumption  $A_j$  and each  $r_j$  naming a reachability condition  $R_j$ , is *satisfiable* iff the formula

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

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

- (2) A  $q$  query of the form (  $c a_1 \dots a_t r_1 \dots r_u$  ), with  $c$  naming an initiality condition  $C$ , each  $a_j$  naming an assumption  $A_j$ , and each  $r_j$  naming a reachability condition  $R_j$ , is *satisfiable* iff the formula

$$\begin{aligned} C &\wedge \text{always } T_S \\ &\wedge \text{always } (A_1 \wedge \dots \wedge A_t) \\ &\wedge \text{eventually } R_1 \wedge \dots \wedge \text{eventually } R_u \end{aligned}$$

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

- (3) A  $q$  query of the form (  $a_1 \dots a_t r_1 \dots r_u f_1 \dots f_v$  ), with each  $a_j$  naming an assumption  $A_j$ , each  $r_j$  naming a reachability condition  $R_j$ , and each  $f_j$  naming a fairness condition  $F_j$ , *satisfiable* iff the formula

$$\begin{aligned} I_S &\wedge \text{always } T_S \\ &\wedge \text{always } (A_1 \wedge \dots \wedge A_t) \\ &\wedge \text{always eventually } F_1 \wedge \dots \wedge \text{always eventually } F_v \\ &\wedge \text{eventually } R_1 \wedge \dots \wedge \text{eventually } R_u \end{aligned}$$

is *satisfiable* in LTL.

- (4) A  $q$  query of the form  $(c\ a_1 \dots a_t\ r_1 \dots r_u\ f_1 \dots f_v)$ , with  $c$  naming an initiality condition  $C$ , each  $a_j$  naming an assumption  $A_j$ , each  $r_j$  naming a reachability condition  $R_j$ , and each  $f_j$  naming a fairness condition  $F_j$ , *satisfiable* iff the formula

$$\begin{aligned} C &\wedge \text{always } T_S \\ &\wedge \text{always } (A_1 \wedge \dots \wedge A_t) \\ &\wedge \text{always eventually } F_1 \wedge \dots \wedge \text{always eventually } F_v \\ &\wedge \text{eventually } R_1 \wedge \dots \wedge \text{eventually } R_u \end{aligned}$$

is **satisfiable** in LTL.

Let  $\mathcal{T}$  be the background theory specified for an MoXI script. For each satisfiable query in a check-system command, the model checker is expected to produce

1. a  $\mathcal{T}$ -interpretation  $\mathcal{I}$  of the (global) free symbols in the script;
2. a witnessing trace in  $\mathcal{I}$ .

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

**Note:** To enforce the infinite state semantics for an query it is enough for it to contain any fairness condition, including the valid formula true.

**Note:** For queries with no fairness conditions, the witnessing trace may not be a trace of the system. [Elaborate]

**Note:** The witnessing trace for a query may satisfy each reachability condition in the query in a different state.

#### Check-system examples [Non-deterministic arbiter.]

```
(check-system NonDetArbiter
:input ((req1 Bool) (req2 Bool))
:output ((gr1 Bool) (gr2 Bool))
; There are never concurrent requests
:assumption (a1 (not (and req1 req2)))
; The same request is never issued twice in a row
:assumption (a2 (and (=> req1 (not req1')) (=> req2 (not req2'))))
; Neg of: every request is immediately granted
:reachable (r (not (and (=> req1 gr1) (=> req2 gr2))))
; check the reachability of r under assumptions a1 and a2
:query (q (a1 a2 r))
)
```

[Temporal queries.]

```

(define-system Historically :input ((b Bool)) :output ((hb Bool))
 :init (= hb b)
 :trans (= hb' (and b' hb))
)

(define-system Before :input ((b Bool)) :output ((bb Bool))
 :init (= bb' false)
 :trans (= bb' b)
)

(define-system Count :input ((b Bool)) :output ((c Int))
 :init (= c (ite b 1 0))
 :trans (= c' (+ c (ite b 0 1)))
)

(define-system Monitor :input ((r1 Bool) (r2 Bool)) :output ((g1 Bool) (g2 Bool))
 :local ((a1 Bool) (a2 Bool) (b Bool) (h1 Bool) (h2 Bool) (bf Bool) (c1 Int))
 :subsys (A (NonDetArbiter r1 r2 g1 g2))
 :subsys (H1 (Historically a1 h1))
 :subsys (H2 (Historically a2 h2))
 :subsys (C (Count g1 c1))
 :subsys (B (Before b bf))
 :inv (and
 ; a1 <=> no requests
 (= a1 (and (not r1) (not r2)))
 ; a2 <=> no grants
 (= a2 (and (not g1) (not g2)))
 ; b <=> c1 is 4
 (= b (= c1 4))
)
)

(check-system Monitor :input ((r1 Bool) (r2 Bool)) :output ((g1 Bool) (g2 Bool))
 :local ((a1 Bool) (a2 Bool) (b Bool) (h1 Bool) (h2 Bool) (bf Bool) (c1 Int))
 ; no concurrent requests
 :assumption (A (not (and r1 r2)))
 ; neg of: if there have been no requests, there have been no grants
 :reachable (P1 (not (=> h1 h2)))
 ; neg of: a request is granted at most 4 times
 :reachable (P2 (not (=> bf (not g1))))
 :query (Q1 (A P1))
 :query (Q2 (A P2))
)

```

**Check-system response** We define a trail each to be a finite sequence of states. A trace consists of two trails:

- a prefix trail and
- a lasso trail.

A trace represents an infinite counterexample and consists of a finite prefix of type trail followed by a lasso of type trail representing an infinite loop: there is a transition from last state of the lasso to its first state. Oppositely from a trace, a certificate represents a proof of correctness. We return one trail or certificate in response to each query.

**Verbose** response with input var **i**, output var **o**, local var **s**, reachability pred **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^
:model (m M) ; M is model in SMT-LIB format
:trail (p ((0 (i i) (o o) (s s) (r r) (f f)) ; numbered state/valuation
 ...
 (j (i i) (o o) (s s) (r r) (f f))
)
)
:trail (l ((...) ... (...)))
:certificate (c :inv F :k n)
)
```

The **compact** response format is identical to the verbose one except that each trail starts with a fully specified state and continues with states that list only the variables whose value differs from their value in the previous state.

[Complete examples]

```
(check-system-response
:verbosity compact
...
)
```