A Model Checking Intermediate Language

An Overview

The NSF:CCRI Team

FMCAD 2023

Model Checking Intermediate Language (IL) goals

The IL has been designed to

- be a general enough intermediate target language for MC
- support a variety of user-facing modeling languages
- be directly supported by tools or compiled to lower level languages
- leverage SAT/SMT technology

General design principles

IL models are meant to be produced and processed mostly by tools

So the IL was designed to have

- simple, easily parsable syntax
- a rich set of data types
- little syntactic sugar, at least initially
- well-understood semantics
- a small but comprehensive set of commands
- simple translations to lower level languages such as Btor2 and Aiger

General design principles

IL models are meant to be produced and processed mostly by tools

So the IL was designed to have

- simple, easily parsable syntax
- a rich set of data types
- little syntactic sugar, at least initially
- well-understood semantics
- a small but comprehensive set of commands
- simple translations to lower level languages such as Btor2 and Aiger

General design principles

IL models are meant to be produced and processed mostly by tools

So the IL was designed to have

- simple, easily parsable syntax
- a rich set of data types
- little syntactic sugar, at least initially
- well-understood semantics
- a small but comprehensive set of commands
- simple translations to lower level languages such as Btor2 and Aiger

General design principles

IL models are meant to be produced and processed mostly by tools

So the IL was designed to have

- simple, easily parsable syntax
- a rich set of data types
- little syntactic sugar, at least initially
- well-understood semantics
- a small but comprehensive set of commands
- simple translations to lower level languages such as Btor2 and Aiger

General design principles

IL models are meant to be produced and processed mostly by tools

So the IL was designed to have

- simple, easily parsable syntax
- a rich set of data types
- little syntactic sugar, at least initially
- well-understood semantics
- a small but comprehensive set of commands
- simple translations to lower level languages such as Btor2 and Aiger

General design principles

IL models are meant to be produced and processed mostly by tools

So the IL was designed to have

- simple, easily parsable syntax
- a rich set of data types
- little syntactic sugar, at least initially
- well-understood semantics
- a small but comprehensive set of commands
- simple translations to lower level languages such as Btor2 and Aiger

General design principles

IL models are meant to be produced and processed mostly by tools

So the IL was designed to have

- simple, easily parsable syntax
- a rich set of data types
- little syntactic sugar, at least initially
- well-understood semantics
- a small but comprehensive set of commands
- simple translations to lower level languages such as Btor2 and Aiger

General design principles

IL models are meant to be produced and processed mostly by tools

So the IL was designed to have

- simple, easily parsable syntax
- a rich set of data types
- little syntactic sugar, at least initially
- well-understood semantics
- a small but comprehensive set of commands
- simple translations to lower level languages such as Btor2 and Aiger

Design principles — implications

1. Little direct support for many of the features offered by
   - hardware modeling languages such as VHDL and Verilog or
   - system modeling languages such as SMV, TLA+, PROMELA, UNITY, Lustre

2. However, enough capability to support the reduction of problems in those languages to problems in the IL

Design principles — implications

1. Little direct support for many of the features offered by
   - hardware modeling languages such as VHDL and Verilog or
   - system modeling languages such as SMV, TLA+, PROMELA, UNITY, Lustre

2. However, enough capability to support the reduction of problems in those languages to problems in the IL

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each system definition

- defines a transition system via the use of SMT formulas
- generally imposes 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
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each system definition

- defines a transition system via the use of SMT formulas
- generally imposes 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
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each system definition

- defines a transition system via the use of SMT formulas
- generally imposes 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
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each system definition

- defines a transition system via the use of SMT formulas
- generally imposes 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
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each system definition

- defines a transition system via the use of SMT formulas
- generally imposes 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
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each system definition

- defines a transition system via the use of SMT formulas
- generally imposes 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
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each system definition

- defines a transition system via the use of SMT formulas
- generally imposes 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
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each verification command

- refers to a previously defined system
- queries the *reachability* of a one-state formula
- can impose environmental assumptions on input
- can specifie fairness condition on queries
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each verification command

- refers to a previously defined system
- queries the *reachability* of a one-state formula
- can impose environmental assumptions on input
- can specifie fairness condition on queries
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each verification command

- refers to a previously defined system
- queries the *reachability* of a one-state formula
- can impose environmental assumptions on input
- can specifie fairness condition on queries
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each verification command

- refers to a previously defined system
- queries the *reachability* of a one-state formula
- can impose environmental assumptions on input
- can specifie fairness condition on queries
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each verification command

- refers to a previously defined system
- queries the *reachability* of a one-state formula
- can impose environmental assumptions on input
- can specifie fairness condition on queries
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each verification command

- refers to a previously defined system
- queries the *reachability* of a one-state formula
- can impose environmental assumptions on input
- can specifie fairness condition on queries
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each verification command

- refers to a previously defined system
- queries the *reachability* of a one-state formula
- can impose environmental assumptions on input
- can specifie fairness condition on queries
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

IL in a Nutshell

Extension the SMT-LIB language with new commands to define and check systems

Each verification command

- refers to a previously defined system
- queries the *reachability* of a one-state formula
- can impose environmental assumptions on input
- can specifie fairness condition on queries
- is hierarchical, i.e., may include (instances of) previously defined systems as subsystems
- can encode synchronous and (later) asynchronous system composition

Current focus

Finite-state systems

but with an eye to infinite-state systems too

Current focus

Finite-state systems

but with an eye to infinite-state systems too

Technical preliminaries

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

$$S \;=\; (\, I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\;\; T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}'] \,)$$

where

- $\boldsymbol{i}$ and $\boldsymbol{i}'$ are two tuples of *input variables* with the same length and type
- $\boldsymbol{o}$ and $\boldsymbol{o}'$ are two tuples of *output variables* with the same length and type
- $\boldsymbol{s}$ and $\boldsymbol{s}'$ are two tuples of *local variables* with the same length and type
- I_S, the *initial state condition* is a formula with free vars from $[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}]$
- T_S, the *transition condition* is a formula with free vars from $[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}']$

Note: A (full) state of S is a valuation of $(\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s})$

Technical preliminaries

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

$$S = (\, I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\; T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}'] \,)$$

where

- $\boldsymbol{i}$ and $\boldsymbol{i}'$ are two tuples of *input variables* with the same length and type
- $\boldsymbol{o}$ and $\boldsymbol{o}'$ are two tuples of *output variables* with the same length and type
- $\boldsymbol{s}$ and $\boldsymbol{s}'$ are two tuples of *local variables* with the same length and type
- I_S, the *initial state condition* is a formula with free vars from $[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}]$
- T_S, the *transition condition* is a formula with free vars from $[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}']$

Note: A (full) state of S is a valuation of $(\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s})$

Technical preliminaries

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

$$S = (\, I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\ T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}']\,)$$

where

- $\boldsymbol{i}$ and $\boldsymbol{i}'$ are two tuples of *input variables* with the same length and type
- $\boldsymbol{o}$ and $\boldsymbol{o}'$ are two tuples of *output variables* with the same length and type
- $\boldsymbol{s}$ and $\boldsymbol{s}'$ are two tuples of *local variables* with the same length and type
- I_S, the *initial state condition* is a formula with free vars from $[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}]$
- T_S, the *transition condition* is a formula with free vars from $[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}']$

Note: A (full) state of S is a valuation of $(\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s})$

SMT-LIB commands

As in SMT-LIB

```
(set-logic L)
(declare-sort s n)
(define-sort s (u₁ ··· uₙ) τ)
(declare-fun f ((x₁ σ₁) ··· (xₙ σₙ)) σ)
(define-fun f ((x₁ σ₁) ··· (xₙ σₙ)) σ t)
(declare-datatype d (···))
(assert F)
```

(perhaps a few more)

SMT-LIB commands

New

`(define-system` S $\cdots$`)`

`(check-system` S $\cdots$`)`

`(declare-enum-sort` S `(`c_1 $\cdots$ c_n`))`

`(declare-range-sort` S `(`m n`))`

SMT-LIB commands

New

```
(define-system S ···)

(check-system S ···)

(declare-enum-sort S (c1 ··· cn))

(declare-range-sort S (m n))
```

Note: IL inherits SMT-LIB's concrete syntax, based on s-expressions
(only prefix operators)

Logical semantics

A **`define-system`** command implicitly defines a *model* (i.e., a Kripke structure) of First-Order Linear Temporal Logic (FO-LTL)

An FO-LTL formula $F[\boldsymbol{f}, \boldsymbol{x}, \boldsymbol{x}']$ with

- free (immutable) constants/functions (aka, uninterpreted symbols) from $\boldsymbol{f}$
- free (mutable) variables from $\boldsymbol{x}, \boldsymbol{x}'$

is *satisfiable* in an SMT theory $\mathcal{T}$ if there is

1. a $\mathcal{T}$-interpretation $\mathcal{I}$ of $\boldsymbol{f}$ and
2. an infinite trace π over $\boldsymbol{x}$ in $\mathcal{I}$

that satisfy F

Logical semantics

A **`define-system`** command implicitly defines a *model* (i.e., a Kripke structure) of First-Order Linear Temporal Logic (FO-LTL)

An FO-LTL formula $F[\boldsymbol{f}, \boldsymbol{x}, \boldsymbol{x}']$ with

- free (immutable) constants/functions (aka, uninterpreted symbols) from $\boldsymbol{f}$
- free (mutable) variables from $\boldsymbol{x}, \boldsymbol{x}'$

is *satisfiable* in an SMT theory $\mathcal{T}$ if there is

1. a $\mathcal{T}$-interpretation $\mathcal{I}$ of $\boldsymbol{f}$ and
2. an infinite trace π over $\boldsymbol{x}$ in $\mathcal{I}$

that satisfy F

Logical semantics

A `define-system` command implicitly defines a *model* (i.e., a Kripke structure) of First-Order Linear Temporal Logic (FO-LTL)

An FO-LTL formula $F[\boldsymbol{f}, \boldsymbol{x}, \boldsymbol{x}']$ with

- free (immutable) constants/functions (aka, uninterpreted symbols) from $\boldsymbol{f}$
- free (mutable) variables from $\boldsymbol{x}, \boldsymbol{x}'$

is *satisfiable* in an SMT theory $\mathcal{T}$ if there is

1. a $\mathcal{T}$-interpretation $\mathcal{I}$ of $\boldsymbol{f}$ and
2. an infinite trace π over $\boldsymbol{x}$ in $\mathcal{I}$

that satisfy F

Trace semantics

Fix

- an FOL-LTL formula $F[\boldsymbol{f}, \boldsymbol{x}, \boldsymbol{x}']$ over a theory $\mathcal{T}$
- a $\mathcal{T}$-interpretation $\mathcal{I}$ of $\boldsymbol{f}$
- an infinite trace $\pi = s_0, s_1, \ldots$ where s_i is an assignment of $\boldsymbol{x}$ into $\mathcal{I}$ for all $i \geq 0$

Let $\pi^i = s_i, s_{i+1}, \ldots$ for all $i \geq 0$

$(\mathcal{I}, \pi)$ *satisfies F*, written $(\mathcal{I}, \pi) \models F$, iff

1. $\mathcal{I}[\boldsymbol{x} \mapsto s_0(\boldsymbol{x}), \boldsymbol{x}' \mapsto s_1(\boldsymbol{x})]$ satisfies F — when F is atomic
2. $(\mathcal{I}, \pi) \not\models G$ — when F is $\neg G$
3. $(\mathcal{I}, \pi) \models G_j$ for $j = 1, 2$ — when F is $G_1 \wedge G_2$
4. $(\mathcal{I}, \pi^1) \models G$ — when F is **next** G
5. $(\mathcal{I}, \pi^i) \models G$ for all $i = 0, \ldots,$ — when F is **always** G
6. $(\mathcal{I}, \pi^i) \models G$ for some $i = 0, \ldots,$ — when F is **eventually** G
7. …

Trace semantics

Fix

- an FOL-LTL formula $F[\boldsymbol{f}, \boldsymbol{x}, \boldsymbol{x}']$ over a theory $\mathcal{T}$
- a $\mathcal{T}$-interpretation $\mathcal{I}$ of $\boldsymbol{f}$
- an infinite trace $\pi = s_0, s_1, \ldots$ where s_i is an assignment of $\boldsymbol{x}$ into $\mathcal{I}$ for all $i \geq 0$

Let $\pi^i = s_i, s_{i+1}, \ldots$ for all $i \geq 0$

$(\mathcal{I}, \pi)$ *satisfies* F , written $(\mathcal{I}, \pi) \models F$, iff

1.	$\mathcal{I}[\boldsymbol{x} \mapsto s_0(\boldsymbol{x}), \boldsymbol{x}' \mapsto s_1(\boldsymbol{x})]$ satisfies F	when F is atomic
2.	$(\mathcal{I}, \pi) \not\models G$	when F is $\neg G$
3.	$(\mathcal{I}, \pi) \models G_j$ for $j = 1, 2$	when F is $G_1 \wedge G_2$
4.	$(\mathcal{I}, \pi^1) \models G$	when F is **next** G
5.	$(\mathcal{I}, \pi^i) \models G$ for all $i = 0, \ldots,$	when F is **always** G
6.	$(\mathcal{I}, \pi^i) \models G$ for some $i = 0, \ldots,$	when F is **eventually** G
7.	$\ldots$	

Finite-Trace semantics

Fix

- an FOL-LTL formula $F[\boldsymbol{f}, \boldsymbol{x}, \boldsymbol{x}']$ over a theory $\mathcal{T}$
- a $\mathcal{T}$-interpretation $\mathcal{I}$ of $\boldsymbol{f}$
- an infinite trace $\pi = s_0, s_1, \ldots$ where s_i is an assignment of $\boldsymbol{x}$ into $\mathcal{I}$ for all $i \geq 0$

Let $\pi^i = s_i, s_{i+1}, \ldots$ for all $i \geq 0$

$(\mathcal{I}, \pi)$ *n-satisfies* F for some $n > 0$, written $(\mathcal{I}, \pi) \models_n F$, iff

1. $\mathcal{I}[\boldsymbol{x} \mapsto s_0(\boldsymbol{x}), \boldsymbol{x}' \mapsto s_1(\boldsymbol{x})]$ satisfies F when F is atomic
2. $(\mathcal{I}, \pi) \not\models_n G$ when F is $\neg G$
3. $(\mathcal{I}, \pi) \models_n G_j$ for $j = 1, 2$ when F is $G_1 \wedge G_2$
4. $(\mathcal{I}, \pi^1) \models_{n-1} G$ and $n - 1 > 0$ when F is **next** G
5. $(\mathcal{I}, \pi^i) \models_{n-i} G$ for all $i = 0, \ldots, n-1$ when F is **always** G
6. $(\mathcal{I}, \pi^i) \models_{n-i} G$ for some $i = 0, \ldots, n-1$ when F is **eventually** G
7. $\ldots$

Model Specification

Atomic system definition

(**define-system** S
 :input ((i_1 δ_1) $\cdots$ (i_m δ_m)) ; input vars
 :output ((o_1 τ_1) $\cdots$ (o_n τ_n)) ; output vars
 :local ((s_1 σ_1) $\cdots$ (s_p δ_p)) ; local vars
 :init I ; initial state formula
 :trans T ; transition formula
 :inv P ; invariant formula
)

Atomic system **definition**

```
(define-system S
 :input  ((i_1 δ_1) ··· (i_m δ_m)) ; input vars
 :output ((o_1 τ_1) ··· (o_n τ_n)) ; output vars
 :local  ((s_1 σ_1) ··· (s_p δ_p)) ; local vars
 :init   I                         ; initial state formula
 :trans  T                         ; transition formula
 :inv    P                         ; invariant formula
)
```

where

- each var gets a primed copy: $i'_1, \ldots, o'_1, \ldots, s'_1, \ldots$
- I and P are one-state formulas (over unprimed vars only)
- T is a two-state formula (over unprimed and primed vars)
- all attributes are optional and their order is immaterial
  - however, **:input**, **:output**, **:local** must occur before **:init**, **:trans**, **:inv**

Default values for missing attributes

attribute	default
`:input`	`()`
`:output`	`()`
`:local`	`()`
`:init`	`true`
`:trans`	`true`
`:inv`	`true`

Examples

```
; The output of Delay is initially in [0,10] and
; then is the previous input
;
(define-system Delay :input ((in Int)) :output ((out Int))
  :init (<= 0 out 10)
  :trans (= out’ in)
)
```

Examples

```
; The output of Delay is initially in [0,10] and
; then is the previous input
;
(define-system Delay :input ((in Int)) :output ((out Int))
  :init (<= 0 out 10)
  :trans (= out’ in)
)
```

Example trace:

step	0	1	2	3	4	5	6	...
in	1	2	3	4	3	2	1	...
out	9	1	2	3	4	3	2	...

Examples

```
; A clocked lossless channel that stutters when clock is false
;
(define-system ClockedChannel
  :input ((clock Bool) (in Int))
  :output ((out Int))
        ; out is unconstrained when clock is false
  :init (=> clock (= out in))
  :trans (ite clock’ (= out’ in’) (= out’ out))
)
```

Examples

```
; A clocked lossless channel that stutters when clock is false
;
(define-system ClockedChannel
  :input ((clock Bool) (in Int))
  :output ((out Int))
        ; out is unconstrained when clock is false
  :init (=> clock (= out in))
  :trans (ite clock' (= out' in') (= out' out))
)
```

Example trace:

step	0	1	2	3	4	5	6	...
clock	F	T	T	F	F	T	F	...
in	1	2	3	4	5	6	7	...
out	-3	2	3	3	3	6	6	...

Example: timed light switch

`TimedSwitch` models a timed light switch

where, once on, the light stays on for 10 steps unless it is switched off before

A Boolean input is provided as a toggle signal

Example: timed light switch

```
(define-enum-sort LightStatus (On Off))

; Guarded-transitions-style definition
(define-system TimedSwitch :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'))             ; Off ->
        (and (= s' Off) (= n' n)))                ; Off
    (=> (and (= s  Off) press')                   ; Off ->
        (and (= s' On)  (= n' n)))                ; On
    (=> (and (= s  On)  (not press') (< 10 n))    ; On ->
        (and (= s' On)  (= n' (+ n 1))))          ; On
    (=> (and (= s  On)  (or press' (>= n 10))     ; On ->
        (and (= s' Off) (= n' 0)))                ; Off
  )
)
```

Example: timed light switch

```
(define-enum-sort LightStatus (On Off))

; Set-of-transitions-style definition
(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
    (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)
    )
)
```

Example: timed light switch

```
(define-enum-sort LightStatus (On Off))

; Equational-style definition
(define-system TimedSwitch3 :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
    (= s’ (ite press’ (flip s)
            (ite (or (= s Off) (>= n 10)) Off
              On)))
    (= n’ (ite (or (= s Off) (s’ Off)) 0
            (+ n 1)))
  )
)

(define-fun flip ((s LightStatus)) LightStatus
  (ite (= s Off) On Off)
)
```

Examples

```
(declare-datatype Event (par (X) (Abs) (Pres (val X))))
```

Values of sort `(Event Int)`: `Abs`, `(Pres 12)`, `(Pres -23)`, ...
Values of sort `(Event Bool)`: `Abs`, `(Pres true)`, `(Pres false)`

Examples

```
(declare-datatype Event (par (X) (Abs) (Pres (val X))))
```

Values of sort `(Event Int)`: `Abs`, `(Pres 12)`, `(Pres -23)`, ...
Values of sort `(Event Bool)`: `Abs`, `(Pres true)`, `(Pres false)`

```
; An event-triggered channel that arbitrarily loses its input data
(define-system LossyIntChannel
  :input ((in (Event Int)))
  :output ((out (Event Int)))
  :inv (or (= out in) (= out Abs))
)
```

Examples

```
(declare-datatype Event (par (X) (Abs) (Pres (val X))))
```

Values of sort `(Event Int)`: `Abs`, `(Pres 12)`, `(Pres -23)`, ...
Values of sort `(Event Bool)`: `Abs`, `(Pres true)`, `(Pres false)`

```
; An event-triggered channel that arbitrarily loses its input data
(define-system LossyIntChannel
  :input ((in (Event Int)))
  :output ((out (Event Int)))
  :inv (or (= out in) (= out Abs))
)

; Equivalent formulation using unconstrained local state
(define-system LossyIntChannel
  :input ((in (Event Int)))
  :output ((out (Event Int)))
  :local ((s Bool))
  ; input event is relayed or not depending on value of s
  :inv (= out (ite s in Abs))
)
```

Atomic system definition — Semantics

```
(define-system S
 :input  ((i₁ δ₁) ··· (iₘ δₘ)) ; input vars
 :output ((o₁ τ₁) ··· (oₙ τₙ)) ; output vars
 :local  ((s₁ σ₁) ··· (sₚ δₚ)) ; local vars
 :init   I                     ; initial state formula
 :trans  T                     ; transition formula
 :inv    P                     ; invariant formula
)
```

Atomic system definition — Semantics

```
(define-system S
 :input  ((i_1 δ_1) ··· (i_m δ_m)) ; input vars
 :output ((o_1 τ_1) ··· (o_n τ_n)) ; output vars
 :local  ((s_1 σ_1) ··· (s_p δ_p)) ; local vars
 :init   I                         ; initial state formula
 :trans  T                         ; transition formula
 :inv    P                         ; invariant formula
)
```

$$S = (I_S, T_S) = (I[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\ P[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}] \wedge T[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}'])$$

where $\boldsymbol{i} = (i_1, \ldots, i_m)$, $\boldsymbol{o} = (o_1, \ldots, o_n)$, $\boldsymbol{s} = (s_1, \ldots, s_n)$

Atomic system definition — Semantics

```
(define-system S
 :input  ((i₁ δ₁) ··· (iₘ δₘ)) ; input vars
 :output ((o₁ τ₁) ··· (oₙ τₙ)) ; output vars
 :local  ((s₁ σ₁) ··· (sₚ δₚ)) ; local vars
 :init   I                     ; initial state formula
 :trans  T                     ; transition formula
 :inv    P                     ; invariant formula
)
```

$$S = (I_S, T_S) = (I[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\ P[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}] \wedge T[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}'])$$

where $\boldsymbol{i} = (i_1, \ldots, i_m)$, $\boldsymbol{o} = (o_1, \ldots, o_n)$, $\boldsymbol{s} = (s_1, \ldots, s_n)$

S denotes the set of all infinite traces that satisfy the FO-LTL formula

$$I_S \ \wedge\ \textbf{always}\ T_S$$

Atomic system definition — Semantics

```
(define-system S
 :input  ((i₁ δ₁) ··· (iₘ δₘ)) ; input vars
 :output ((o₁ τ₁) ··· (oₙ τₙ)) ; output vars
 :local  ((s₁ σ₁) ··· (sₚ δₚ)) ; local vars
 :init   I                     ; initial state formula
 :trans  T                     ; transition formula
 :inv    P                     ; invariant formula
)
```

Note:

Systems are meant to be *progressive*: every reachable state has a successor wrt T_S

However, they may not be because of the generality of T and P

(It is possible to define deadlocking systems)

Composite system definition

```
(define-system S
 :input  ((i₁ δ₁) ··· (iₘ δₘ)) ; input vars
 :output ((o₁ τ₁) ··· (oₙ τₙ)) ; output vars
 :local  ((s₁ σ₁) ··· (sₚ δₚ)) ; local vars
 :subsys (N₁ (S₁ x₁ y₁))       ; component subsystem
  ...      ...                    ...
 :subsys (N_q (S_q x_q y_q))   ; component subsystem
)
```

Composite system definition

```
(define-system S
 :input  ((i₁ δ₁) ··· (iₘ δₘ)) ; input vars
 :output ((o₁ τ₁) ··· (oₙ τₙ)) ; output vars
 :local  ((s₁ σ₁) ··· (sₚ δₚ)) ; local vars
 :subsys (N₁ (S₁ x₁ y₁))       ; component subsystem
   ...      ...                   ...
 :subsys (N_q (S_q x_q y_q))   ; component subsystem
)
```

where

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

Composite system definition extended

```
(define-system S
 :input  ((i₁ δ₁) ··· (iₘ δₘ)) ; input vars
 :output ((o₁ τ₁) ··· (oₙ τₙ)) ; output vars
 :local  ((s₁ σ₁) ··· (sₚ δₚ)) ; local vars
 :subsys (N₁ (S₁ x₁ y₁))       ; component subsystem
   ···      ···                  ···
 :subsys (N_q (S_q x_q y_q))   ; component subsystem
 :init   I                     ; initial state formula
 :trans  T                     ; transition formula
 :inv    P                     ; invariant formula
)
```

Composite system definition extended

```
(define-system S
 :input  ((i₁ δ₁) ··· (iₘ δₘ)) ; input vars
 :output ((o₁ τ₁) ··· (oₙ τₙ)) ; output vars
 :local  ((s₁ σ₁) ··· (sₚ δₚ)) ; local vars
 :subsys (N₁ (S₁ x₁ y₁))       ; component subsystem
  ...     ...                    ...
 :subsys (N_q (S_q x_q y_q))   ; component subsystem
 :init   I                     ; initial state formula
 :trans  T                     ; transition formula
 :inv    P                     ; invariant formula
)
```

Composition is synchronous by default

Examples

```
; 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 Delay2 :input ((in Int)) :output ((out Int))
  :local ((temp Int))
  :subsys (D1 (Delay in temp))
  :subsys (D2 (Delay temp out))
)
```

Examples

```
; 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 Delay2 :input ((in Int)) :output ((out Int))
  :local ((temp Int))
  :subsys (D1 (Delay in temp))
  :subsys (D2 (Delay temp out))
)
```

Example trace:

step	0	1	2	3	4	5	6	7	...
in	2	3	4	5	6	7	8	9	...
temp	0	2	3	4	5	6	7	8	...
out	0	0	2	3	4	5	6	7	...

Examples

```
(define-system Latch :input ((s Bool) (r Bool)) :output ((o Bool))
  :local ((b Bool))
  :trans (= o’ (or (and s (or (not r) b))
                   (and (not s) (not r) o)))
)

(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)))
)

(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))
)
```

Composite system definition — Semantics

```
(define-system S
 :input  ((i₁ δ₁) ··· (iₘ δₘ)) ; input vars
 :output ((o₁ τ₁) ··· (oₙ τₙ)) ; output vars
 :local  ((s₁ σ₁) ··· (sₚ δₚ)) ; local vars
 :subsys (N₁ (S₁ x₁ y₁))       ; component subsystem
   ···      ···                  ···
 :subsys (N_q (S_q x_q y_q))   ; component subsystem
)
```

Composite system definition — Semantics

```
(define-system S
 :input  ((i₁ δ₁) ··· (iₘ δₘ)) ; input vars
 :output ((o₁ τ₁) ··· (oₙ τₙ)) ; output vars
 :local  ((s₁ σ₁) ··· (sₚ δₚ)) ; local vars
 :subsys (N₁ (S₁ x₁ y₁))       ; component subsystem
   ...     ...                   ...
 :subsys (N_q (S_q x_q y_q))   ; component subsystem
)
```

Let $S_k = (\, I_k[\boldsymbol{i}_k, \boldsymbol{o}_k, \boldsymbol{s}_k],\ T_k[\boldsymbol{i}_k, \boldsymbol{o}_k, \boldsymbol{s}_k, \boldsymbol{i}'_k, \boldsymbol{o}'_k, \boldsymbol{s}'_k]\,)$ for $k = 1, \ldots, q$, with $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_q$ all distinct

Let $\boldsymbol{i} = (i_1, \ldots, i_m)$, $\boldsymbol{o} = (o_1, \ldots, o_n)$, $\boldsymbol{s} = s_1, \ldots, s_q, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_q$

$$S = (I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}], T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}'])$$

with $\quad I_S \;=\; \bigwedge_{k=1}^{q} I_k[\boldsymbol{x}_k, \boldsymbol{y}_k, \boldsymbol{s}_k] \qquad T_S \;=\; \bigwedge_{k=1}^{q} T_k[\boldsymbol{x}_k, \boldsymbol{y}_k, \boldsymbol{s}_k, \boldsymbol{x}'_k, \boldsymbol{y}'_k, \boldsymbol{s}'_k]$

Composite system definition extended — Semantics

```
(define-system S
 :input  ((i_1 δ_1) ··· (i_m δ_m)) ; input vars
 :output ((o_1 τ_1) ··· (o_n τ_n)) ; output vars
 :local  ((s_1 σ_1) ··· (s_p δ_p)) ; local vars
 :subsys (N_1 (S_1 x_1 y_1))       ; component subsystem
  ···     ···                        ···
 :subsys (N_q (S_q x_q y_q))       ; component subsystem
 :init   I                         ; initial state formula
 :trans  T                         ; transition formula
 :inv    P                         ; invariant formula
)
```

$$S = (I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}], T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}'])$$

with $I_S = I \wedge \bigwedge_{k=1}^{q} I_k[\boldsymbol{x}_k, \boldsymbol{y}_k, \boldsymbol{s}_k]$ $\quad T_S = P \wedge T \wedge \bigwedge_{k=1}^{q} T_k[\boldsymbol{x}_k, \boldsymbol{y}_k, \boldsymbol{s}_k, \boldsymbol{x}'_k, \boldsymbol{y}'_k, \boldsymbol{s}'_k]$

Expressiveness

define-system + SMT-LIB commands and types appear sufficient to allow faithful reductions from (full or large fragment of)

- Moore and Mealy machines
- I/O automata
- SMV and nuXMV
- UNITY
- TLA+
- Reactive Modules
- Lustre
- SAL

Model Checking

System checking command

```
(check-system S
 :input       ((i₁ δ₁) ··· (iₘ δₘ)) ; renaming of S's input vars
 :output      ((o₁ τ₁) ··· (oₙ τₙ)) ; renaming of S's output vars
 :local       ((s₁ σ₁) ··· (sₚ δₚ)) ; renaming of S's local vars
 :assumption (a A)                  ; environmental assumption
 :reachable  (r R)                  ; reachability condition
 :fairness   (f F)                  ; fairness condition
 :current    (c C)                  ; initiality condition
 :query      (q (g₁ ··· g_q))       ; trace query to be checked
)
```

System checking command

```
(check-system S
 :input       ((i₁ δ₁)  ···  (iₘ δₘ)) ; renaming of S's input vars
 :output      ((o₁ τ₁)  ···  (oₙ τₙ)) ; renaming of S's output vars
 :local       ((s₁ σ₁)  ···  (sₚ δₚ)) ; renaming of S's local vars
 :assumption  (a A)                   ; environmental assumption
 :reachable   (r R)                   ; reachability condition
 :fairness    (f F)                   ; fairness condition
 :current     (c C)                   ; initiality condition
 :query       (q (g₁ ··· g_q))        ; trace query to be checked
)
```

where

- a, r, f, c, q are identifiers; each g_i ranges over $\{a, r, f, c\}$
- C is a one-state (non-temporal) formula over the given vars
- A, R, F are one- or two-state (non-temporal) formulas over the given vars
- all attributes are optional and their order is immaterial
- all attributes but the first three are repeatable

System checking command

```
(check-system S
 :input      ((i₁ δ₁) ··· (iₘ δₘ)) ; renaming of S's input vars
 :output     ((o₁ τ₁) ··· (oₙ τₙ)) ; renaming of S's output vars
 :local      ((s₁ σ₁) ··· (sₚ δₚ)) ; renaming of S's local vars
 :assumption (a A)                 ; environmental assumption
 :reachable  (r R)                 ; reachability condition
 :fairness   (f F)                 ; fairness condition
 :current    (c C)                 ; initiality condition
 :query      (q (g₁ ··· g_q))      ; trace query to be checked
)
```

System checking command — Semantics

```
(check-system S
 :input       ((i₁ δ₁) ··· (iₘ δₘ)) ; renaming of S's input vars
 :output      ((o₁ τ₁) ··· (oₙ τₙ)) ; renaming of S's output vars
 :local       ((s₁ σ₁) ··· (sₚ δₚ)) ; renaming of S's local vars
 :assumption (a A)                  ; environmental assumption
 :reachable  (r R)                  ; reachability condition
 :fairness   (f F)                  ; fairness condition
 :current    (c C)                  ; initiality condition
 :query      (q (a r))
)
```

Query q succeeds iff the formula below is n-satisfiable in LTL for some $n > 0$

$$I_S \wedge \textbf{always}\ T_S \wedge \textbf{always}\ A \wedge \textbf{eventually}\ R \wedge \textbf{always eventually}\ F$$

where I_S and T_S are the initial state and transition predicates of S modulo the renamings above

System checking command — Semantics

```
(check-system S
 :input       ((i₁ δ₁) ··· (iₘ δₘ)) ; renaming of S's input vars
 :output      ((o₁ τ₁) ··· (oₙ τₙ)) ; renaming of S's output vars
 :local       ((s₁ σ₁) ··· (sₚ δₚ)) ; renaming of S's local vars
 :assumption  (a A)                 ; environmental assumption
 :reachable   (r R)                 ; reachability condition
 :fairness    (f F)                 ; fairness condition
 :current     (c C)                 ; initiality condition
 :query       (q (a f r))
)
```

Query q succeeds iff the formula below is satisfiable in LTL

$$I_S \wedge \textbf{always}\ T_S \wedge \textbf{always}\ A \wedge \textbf{eventually}\ R \wedge \textbf{always eventually}\ F$$

where I_S and T_S are the initial state and transition predicates of S modulo the renamings above

System checking command — Semantics

```
(check-system S
 :input      ((i₁ δ₁) ··· (i_m δ_m)) ; renaming of S's input vars
 :output     ((o₁ τ₁) ··· (o_n τ_n)) ; renaming of S's output vars
 :local      ((s₁ σ₁) ··· (s_p δ_p)) ; renaming of S's local vars
 :assumption (a A)                   ; environmental assumption
 :reachable  (r R)                   ; reachability condition
 :fairness   (f F)                   ; fairness condition
 :current    (c C)                   ; initiality condition
 :query      (q (a c r))
)
```

Query q succeeds iff the formula below is n-satisfiable in LTL for some $n > 0$

$$C \wedge \textbf{always}\ T_S \wedge \textbf{always}\ A \wedge \textbf{eventually}\ R \wedge \textbf{always eventually}\ F$$

where I_S and T_S are the initial state and transition predicates of S modulo the renamings above

System **checking** command — Semantics

```
(check-system S
 :input       ((i₁ δ₁) ··· (i_m δ_m)) ; renaming of S's input vars
 :output      ((o₁ τ₁) ··· (o_n τ_n)) ; renaming of S's output vars
 :local       ((s₁ σ₁) ··· (s_p δ_p)) ; renaming of S's local vars
 :assumption  (a A)                   ; environmental assumption
 :reachable   (r R)                   ; reachability condition
 :fairness    (f F)                   ; fairness condition
 :current     (c C)                   ; initiality condition
 :query       (q (g₁ ··· g_q))        ; trace query to be checked
)
```

For each successful query, the model checker is expected to produce

- a $\mathcal{T}$-interpretation $\mathcal{I}$ (of the free immutable symbols) and
- a *witnessing* trace in $\mathcal{I}$

System **checking** command — Semantics

```
(check-system S
 :input      ((i_1 δ_1) ··· (i_m δ_m)) ; renaming of S's input vars
 :output     ((o_1 τ_1) ··· (o_n τ_n)) ; renaming of S's output vars
 :local      ((s_1 σ_1) ··· (s_p δ_p)) ; renaming of S's local vars
 :assumption (a A)                     ; environmental assumption
 :reachable  (r R)                     ; reachability condition
 :fairness   (f F)                     ; fairness condition
 :current    (c C)                     ; initiality condition
 :query      (q (g_1 ··· g_q))         ; trace query to be checked
)
```

For each successful query, the model checker is expected to produce

- a $\mathcal{T}$-interpretation $\mathcal{I}$ (of the free immutable symbols) and
- a *witnessing* trace in $\mathcal{I}$

Different queries may be given different interpretations and traces

Example — Non-deterministic arbiter

```
(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)
    ; unconstrained value of s used as non-deterministic choice
        (ite s (and g1       (not g2))
               (and (not g1) g2     )))
  )
)
```

Example — 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))
)
```

Example — 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) (b0 Bool) (b1 Bool) (b2 Bool)
          (h1 Bool) (h2 Bool) (c Int) (bf Bool))
  :subsys (NDA (NonDetArbiter r1 r2 g1 g2))
  :subsys (His1 (Historically a1 h1))
  :subsys (His2 (Historically a2 h2))
  :subsys (Cnt (Count g1 c))
  :subsys (Bf (Before b0 bf))
  :inv (and
     ; a1 = no concurrent requests       a2 = no concurrent grants
    (= a1 (and (not r1) (not r2)))    (= a2 (and (not g1) (not g2)))    (= b0 (= c 4))
    (= b1 (=> h1 h2)) ; b1 = if there have been no requests, there have been no grants
    (= b2 (=> bf (not g1))))) ; b2 = a request is granted at most 4 times

(check-system Monitor :input ((r1 Bool) (r2 Bool))
  :output ((g1 Bool) (g2 Bool))
  :local (_ _ _ (b1 Bool) (b2 Bool) _ _ _ _)
  :assumption (A (not (and r1 r2))) :reachable (R (not (and b1 b2))) :query (Q1 (A R))
)
```

Example — Multiple queries

```
(check-system NonDetArbiter :input ((r1 Bool) (r2 Bool))
  :output ((g1 Bool) (g2 Bool))
  :assumption (a (not (and r1 r2)))
  ; Neg of: Every request is (immediately) granted
  :reachable (p1 (not (and (=> r1 g1) (=> r2 g2))))
  ; Neg of: In the absence of other requests, every request is granted
  :reachable (p2 (not (=> (!= r1 r2) (and (=> r1 g1) (=> r2 g2)))))
  ; Neg of: A request is granted only if present
  :reachable (p3 (not (and (=> g1 r1) (=> g2 r2))))
  ; Neg of: At most one request is granted at any one time
  :reachable (p4 (not (not (and g1 g2))))
  ; Neg of: In case of concurrent requests, one of them is always granted
  :reachable (p5 (not (=> (and r1 r2) (or g1 g2))))
  :query (q1 (a p1)) :query (q2 (a p2)) :query (q3 (a p3))
  :query (q4 (a p4)) :query (q5 (a p5))
)
```

Each query can be witnessed by a different $\mathcal{T}$-interpretation and trace in it

Output format for check-system

(**define-system** A **:input** ((i σ_A)) **:output** ((o τ_A)) **:local** ((s θ_A)) ...)

(**define-system** B **:input** ((i σ_B)) **:output** ((o τ_B)) **:local** ((s θ_B))
:subsys ($\cdots$ (S (A $\cdots$)) $\cdots$) ...)

(**check-system** B $\cdots$ **:fairness** (f $\cdots$) **:reachable** (r $\cdots$) $\cdots$
:query (q (r f $\cdots$)) $\cdots$)

Output:

(**response**
:query (q :result sat :model m :trace t)
:model ($\cdots$) ; SMT-LIB interpretation of free symbols
:trail (p (; state sequence
((i i_0) (o o_0) (s s_0) (S::i $i_{S,0}$) (S::o $o_{S,0}$) (S::s $s_{S,0}$) (r r_0) (f f_0)$\cdots$)
...
((i i_k) (o o_k) (s s_k) (S::i $i_{S,k}$) (S::o $o_{S,k}$) (S::s $s_{S,k}$) (r r_k) (f f_k) $\cdots$)
)
)
:trail (l ($\cdots$))
...
:trace (q **:prefix** p **:lasso** l) ; witness trace for query q is pl^ω
...
)

Additional features

- Special predicate for frame conditions
- Special predicate for deadlock states
- Aggregate queries

Planned extensions

- Executable system definitions
- Parametric models

Resources

Available at `https://github.com/ModelChecker/FMCAD23-Tutorial`

- These slides
- Examples of systems and queries
- Syntax highlighting for VS Code
- Executables of an experimental version of Kind 2 model checker with MCIL front-end

Available at `https://github.com/ModelChecker/IL`

- Detailed document of IL definition

What’s intentionally missing (and why)

- Restrictions to just bit vector types

  Other types are useful!

- Stronger syntactic restrictions for **:init** and **:trans** formulas

  Should be enforced in the user-facing language

- Direct support for LTL, or your favorite temporal logic, in **check-system**

  Generality, mostly

- Global (mutable) variables *a la* SAL

  Tricky to get right

- Parametric components as in SMV or SAL

  Some support. The rest is better provided in the user-facing language

- Compositional reasoning features (i.e., assume-guarantee contracts)

  Too many different approaches out there

What's intentionally missing (and why)

- Restrictions to just bit vector types

  Other types are useful!

- Stronger syntactic restrictions for **:init** and **:trans** formulas

  Should be enforced in the user-facing language

- Direct support for LTL, or your favorite temporal logic, in **check-system**

  Generality, mostly

- Global (mutable) variables *a la* SAL

  Tricky to get right

- Parametric components as in SMV or SAL

  Some support. The rest is better provided in the user-facing language

- Compositional reasoning features (i.e., assume-guarantee contracts)

  Too many different approaches out there

What’s intentionally missing (and why)

- Restrictions to just bit vector types

  Other types are useful!

- Stronger syntactic restrictions for **:init** and **:trans** formulas

  Should be enforced in the user-facing language

- Direct support for LTL, or your favorite temporal logic, in **check-system**

  Generality, mostly

- Global (mutable) variables *a la* SAL

  Tricky to get right

- Parametric components as in SMV or SAL

  Some support. The rest is better provided in the user-facing language

- Compositional reasoning features (i.e., assume-guarantee contracts)

  Too many different approaches out there

What's intentionally missing (and why)

- Restrictions to just bit vector types

  Other types are useful!

- Stronger syntactic restrictions for **:init** and **:trans** formulas

  Should be enforced in the user-facing language

- Direct support for LTL, or your favorite temporal logic, in **check-system**

  Generality, mostly

- Global (mutable) variables *a la* SAL

  Tricky to get right

- Parametric components as in SMV or SAL

  Some support. The rest is better provided in the user-facing language

- Compositional reasoning features (i.e., assume-guarantee contracts)

  Too many different approaches out there

What's intentionally missing (and why)

- Restrictions to just bit vector types

  Other types are useful!

- Stronger syntactic restrictions for **:init** and **:trans** formulas

  Should be enforced in the user-facing language

- Direct support for LTL, or your favorite temporal logic, in **check-system**

  Generality, mostly

- Global (mutable) variables *a la* SAL

  Tricky to get right

- Parametric components as in SMV or SAL

  Some support. The rest is better provided in the user-facing language

- Compositional reasoning features (i.e., assume-guarantee contracts)

  Too many different approaches out there

What's intentionally missing (and why)

- Restrictions to just bit vector types
  Other types are useful!
- Stronger syntactic restrictions for **:init** and **:trans** formulas
  Should be enforced in the user-facing language
- Direct support for LTL, or your favorite temporal logic, in **check-system**
  Generality, mostly
- Global (mutable) variables *a la* SAL
  Tricky to get right
- Parametric components as in SMV or SAL
  Some support. The rest is better provided in the user-facing language
- Compositional reasoning features (i.e., assume-guarantee contracts)
  Too many different approaches out there

What's intentionally missing (and why)

- Restrictions to just bit vector types
  Other types are useful!
- Stronger syntactic restrictions for **:init** and **:trans** formulas
  Should be enforced in the user-facing language
- Direct support for LTL, or your favorite temporal logic, in **check-system**
  Generality, mostly
- Global (mutable) variables *a la* SAL
  Tricky to get right
- Parametric components as in SMV or SAL
  Some support. The rest is better provided in the user-facing language
- Compositional reasoning features (i.e., assume-guarantee contracts)
  Too many different approaches out there

What's intentionally missing (and why)

- Restrictions to just bit vector types
  Other types are useful!
- Stronger syntactic restrictions for **:init** and **:trans** formulas
  Should be enforced in the user-facing language
- Direct support for LTL, or your favorite temporal logic, in **check-system**
  Generality, mostly
- Global (mutable) variables *a la* SAL
  Tricky to get right
- Parametric components as in SMV or SAL
  Some support. The rest is better provided in the user-facing language
- Compositional reasoning features (i.e., assume-guarantee contracts)
  Too many different approaches out there

Additional Features

Special predicate: `Deadlock`

For every system $S = (\, I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\ T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}'] \,)$

`Deadlock` is a predicate (implicitly) over $\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}$

A state $\{\boldsymbol{i} \mapsto \boldsymbol{i}_0,\ \boldsymbol{o} \mapsto \boldsymbol{o}_0,\ \boldsymbol{s} \mapsto \boldsymbol{s}_0\}$ satisfies `Deadlock`, or
is deadlocked,
iff
it satisfies the formula $\exists \boldsymbol{i}'\, \forall \boldsymbol{o}'\, \forall \boldsymbol{s}'\, \neg T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}']$

Special predicate: `Deadlock`

For every system $S = (\, I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\ T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}'] \,)$

`Deadlock` is a predicate (implicitly) over $\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}$

A state $\{\boldsymbol{i} \mapsto \boldsymbol{i}_0,\ \boldsymbol{o} \mapsto \boldsymbol{o}_0,\ \boldsymbol{s} \mapsto \boldsymbol{s}_0\}$ satisfies `Deadlock`, or
is deadlocked,
iff
it satisfies the formula $\exists \boldsymbol{i}'\, \forall \boldsymbol{o}'\, \forall \boldsymbol{s}'\ \neg T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}']$

Uses of `Deadlock`

Examples

- ```
  (check-system S ···
    :assumption (a A) :current (d Deadlock) :query (a d))
  ```
  checks the existence of deadlocked states under assumption *A*

- ```
  (check-system S ···
    :assumption (a A) :reachable (d Deadlock) :query (a d))
  ```
  checks the reachability of deadlocked states under assumption *A*

- ```
  (check-system S ···
    :fairness (f true) :reachable (r R) :query (f r))
  ```
  checks the reachability of *R* on infinite (hence deadlock-free) traces
  ```

Uses of `Deadlock`

Examples

- `(check-system S ···`
  `:assumption (a A) :current (d Deadlock) :query (a d))`

  checks the existence of deadlocked states under assumption A

- `(check-system S ···`
  `:reachable (d Deadlock)` with `:assumption (a A) :query (a d))`

  checks the reachability of deadlocked states under assumption A

- `(check-system S ···`
  `:fairness (f true) :reachable (r R) :query (f r))`

  checks the reachability of R on infinite (hence deadlock-free) traces

Uses of `Deadlock`

Examples

- `(check-system` *S* ⋯
  `:assumption (`*a* *A*`) :current (`*d* `Deadlock) :query (`*a* *d*`))`

  checks the existence of deadlocked states under assumption *A*

- `(check-system` *S* ⋯
  `:assumption (`*a* *A*`) :reachable (`*d* `Deadlock) :query (`*a* *d*`))`

  checks the reachability of deadlocked states under assumption *A*

- `(check-system` *S* ⋯
  `:fairness (`*f* `true) :reachable (`*r* *R*`) :query (`*f* *r*`))`

  checks the reachability of *R* on infinite (hence deadlock-free) traces

Special predicate: `OnlyChange`

For every system $S = (\, I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\ T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}'] \,)$

`OnlyChange` is a *multi-arity* predicate over $\boldsymbol{o}, \boldsymbol{s}, \boldsymbol{o}', \boldsymbol{s}'$:

$$\texttt{OnlyChange}(x_1, \ldots, x_n) \quad \equiv \quad \bigwedge \{y' = y \mid y \in (\boldsymbol{o} \cup \boldsymbol{s} \cup \boldsymbol{o}' \cup \boldsymbol{s}') \setminus \{x_1, \ldots, x_n\}\}$$

Fixes the value of all output and local variables not in $(x_1, \ldots, x_n)$

It is a useful abbreviation in transition conditions to express transitions that leave many state variables unchanged

Note: $\texttt{OnlyChange}(x_1, \ldots, x_n)$ does not actually constrain the x_i's in any way

Special predicate: `OnlyChange`

For every system $S = (\, I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\ T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}']\,)$

`OnlyChange` is a *multi-arity* predicate over $\boldsymbol{o}, \boldsymbol{s}, \boldsymbol{o}', \boldsymbol{s}'$:

$$\texttt{OnlyChange}(x_1, \ldots, x_n) \quad \equiv \quad \bigwedge \{y' = y \mid y \in (\boldsymbol{o} \cup \boldsymbol{s} \cup \boldsymbol{o}' \cup \boldsymbol{s}') \setminus \{x_1, \ldots, x_n\}\}$$

Fixes the value of all output and local variables not in $(x_1, \ldots, x_n)$

It is a useful abbreviation in transition conditions to express transitions that leave many state variables unchanged

Note: $\texttt{OnlyChange}(x_1, \ldots, x_n)$ does not actually constrain the x_i's in any way

Special predicate: `OnlyChange`

For every system $S = (\, I_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}],\ T_S[\boldsymbol{i}, \boldsymbol{o}, \boldsymbol{s}, \boldsymbol{i}', \boldsymbol{o}', \boldsymbol{s}']\,)$

`OnlyChange` is a *multi-arity* predicate over $\boldsymbol{o}, \boldsymbol{s}, \boldsymbol{o}', \boldsymbol{s}'$:

$$\texttt{OnlyChange}(x_1, \ldots, x_n) \quad \equiv \quad \bigwedge \{y' = y \mid y \in (\boldsymbol{o} \cup \boldsymbol{s} \cup \boldsymbol{o}' \cup \boldsymbol{s}') \setminus \{x_1, \ldots, x_n\}\}$$

Fixes the value of all output and local variables not in $(x_1, \ldots, x_n)$

It is a useful abbreviation in transition conditions to express transitions that leave many state variables unchanged

Note: $\texttt{OnlyChange}(x_1, \ldots, x_n)$ does not actually constrain the x_i's in any way

Example

```
; increment n_i iff n = i; n_i is 0 initially if not incremented
(define-system Increment :input ((i Int))
  :output ((inc Bool) (n1 Int) (n2 Int) ··· (n5 Int))
  :inv (= inc (<= 1 i 5))
  :init (and
    (=> (= n 1) (and (= n1 1) (= n2 n3 n4 n5 0)))
      ⋮
    (=> (= n 5) (and (= n5 1) (= n1 n2 n3 n4 0)))
    (=> (not (<= 1 n 5)) (= n1 n2 n3 n4 n5 0))
  )
  :trans (and
    (=> (= n' 1) (and (= n1' (+ n1 1)) (OnlyChange inc n1)))
      ⋮
    (=> (= n' 5) (and (= n5' (+ n5 1)) (OnlyChange inc n5)))
    (=> (not (<= 1 n' 5)) (OnlyChange inc))
  )
)
```

Aggregate queries

```
(check-system S
 :input     ((i_1 δ_1) ··· (i_m δ_m))
 :output    ((o_1 τ_1) ··· (o_n τ_n))
 ⋮
 :queries   ((q_1 (g_{1,1} ··· g_{1,n_1})) ··· (q_k (g_{k,1} ··· g_{k,n_k})))
)
```

- Each query q_i can be witnessed by a different trace
- However, each free immutable symbol has the same interpretation across all queries

Possible Extensions

Executable system definitions

Local and output variables are defined exclusively equationally

```
(define-system TimedSwitch :input ((press Bool)) :output ((sig Bool))
  :local ((s LightStatus) (n Int))
  :inv-def (
    (sig (= s On))
  )
  :init-def (
    (n 0)
    (s (ite press On Off))
  )
  :next-def (
    (s’ (ite press’ (ite (= s Off) On Off))
          (ite (= s Off) Off (ite (< n 10) On Off))))
    (n’ (ite (or (= s Off) (s’ Off)) 0 (+ n 1)))
  ))
```

Restrictions: (guaranteeing progressiveness and executability)

- Each local or output variable must be listed in **`:inv-def`** or in both **`:init-def`** and **`:next-def`**
- No definitional cycles
- No uninterpreted symbols

Parametric definitions

```
(define-system Delay :param ((V Sort) (d V) (n Int)) :input ((in V))
  :output ((out V))
  :local ((a (Array Int V)))
  :inv (and
    (= in (select a 0))
    (= out (select a n))
  )
  :init (forall ((i Int)) (=> (<= 1 i n)
           (= (select a i) d))
        )
  :trans (forall ((i Int)) (=> (<= 1 i n))
            (= (select a' i) (select a (- i 1))))
         )
)

(check-system Delay :param ((V String) (d "") (n 4)) :input ((in String))
  :output ((out String))
  :local ((a (Array Int String)))
  ...
)
```

Restrictions: parameters are immutable (rigid)