The new SAL symbolic model checker

Yang Zhao

Department of Computer Science and Engineering

University of California at Riverside

June, 2010 $\sim$ September, 2010

Motivation

- Symbolic model checking is one of the state-of-the-art approaches to the verification of complex systems.
  - State-space generation (reachability analysis)
  - CTL and LTL model checking
  - $\cdots$
- Many symbolic model checkers have been developed, and most of them are based on binary decision diagrams (BDDs) manipulation. And CUDD is the most widely used BDD library.
  - NuSMV, VIS, SAL, $\cdots$
- Edge-valued multi-value decision diagrams (EVMDD) and saturation algorithm provide a more efficient approach to state-space generation and CTL model checking.
- Objectives: integrating the EVMDD-based algorithms in the existing model checker SAL, and comparing them with existing BDD algorithms.

Preliminary

Structured discrete-state models

A structured discrete-state model is specified by $\langle \widehat{\mathcal{S}}, \mathcal{S}_{init}, \mathcal{E} \rangle$:

- a potential state space $\widehat{\mathcal{S}} = \mathcal{S}_L \times \cdots \times \mathcal{S}_1$
  - the (global) state is of the form $\mathbf{i} = (i_L, ..., i_1)$
  - $\mathcal{S}_k$ is the (discrete) local state space for submodel k or local domain for state variable x_k
  - if $\mathcal{S}_k$ is finite, we can map it to $\{0, 1, \ldots, n_k - 1\}$ n_k is known after state-space generation
- a set of initial states $\mathcal{S}_{init} \subseteq \widehat{\mathcal{S}}$
  - often there is a single initial state $\mathbf{s}_{init}$
- a set of events $\mathcal{E}$ defining disjunctively-partitioned next-state functions or transition relations
  - $\mathcal{N}_\alpha : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ $\quad$ $\mathbf{j} \in \mathcal{N}_\alpha(\mathbf{i})$ iff state $\mathbf{j}$ can be reached by firing event α in state $\mathbf{i}$
  - $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ $\quad$ $\mathcal{N}(\mathbf{i}) = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha(\mathbf{i})$ $\quad$ image computation
  - naturally extended to sets of states $\mathcal{N}_\alpha(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}_\alpha(\mathbf{i})$ and $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$
  - α is enabled in $\mathbf{i}$ iff $\mathcal{N}_\alpha(\mathbf{i}) \neq \emptyset$, otherwise it is disabled

Locality

Locality in events:

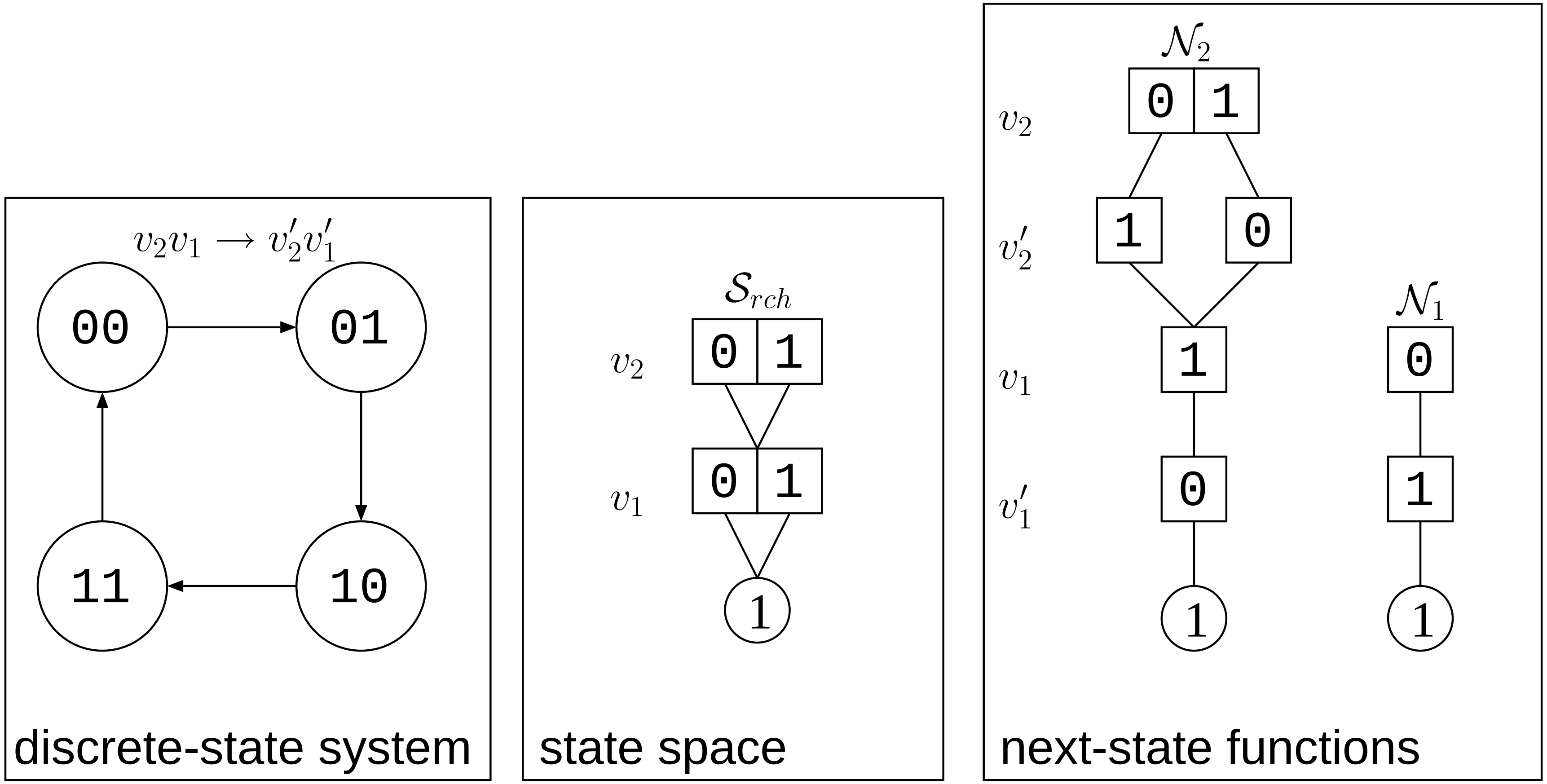
- α is independent of the k^{th} submodel if:
  - its enabling does not depend on i_k,
  - and its firing does not change the value of i_k.
- A level k belongs to $supp(\alpha)$, if α is not independent of k^{th} submodel.
- Let $Top(\alpha)$ be the highest-numbered level in $supp(\alpha)$.
- Let $\mathcal{E}_k$ be the set of events $\{\alpha \in \mathcal{E} : Top(\alpha) = k\}$.
- Let $\mathcal{N}_k$ be the next-state function corresponding to all events in $\mathcal{E}_k$:

$$\mathcal{N}_k = \bigcup_{\alpha \in \mathcal{E}_k} \mathcal{N}_\alpha$$

$\alpha :$ `z=0 ==> y'=x+1`

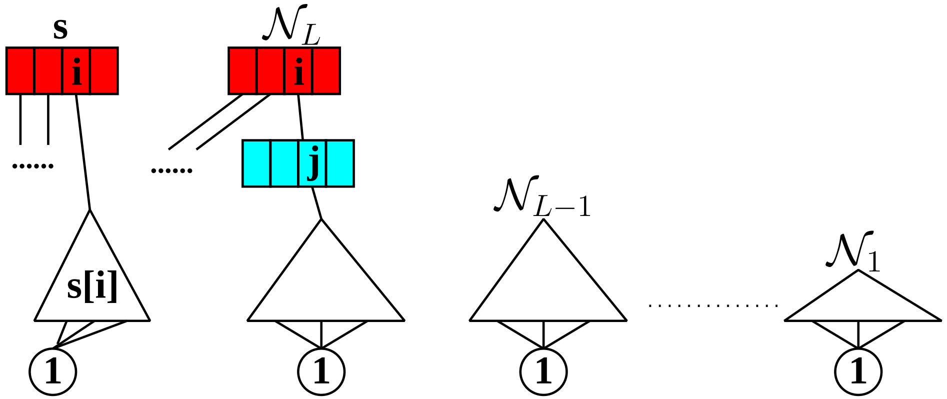
- $supp(\alpha) = \{x, y, z\}$
- $Top(\alpha) = Max(x.lvl, y.lvl, z.lvl)$

Example: 2-bit counter



Saturation: an iteration strategy based on the model structure

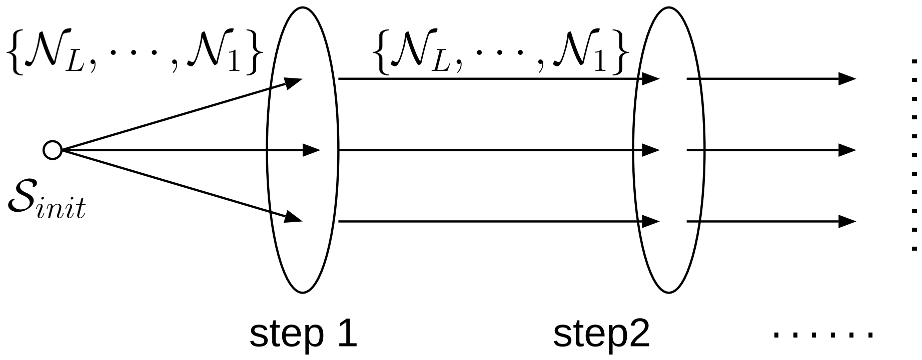
MDD node p at level k is ***saturated*** if it encodes a fixpoint w.r.t. any event α s.t. $Top(\alpha) \leq k$



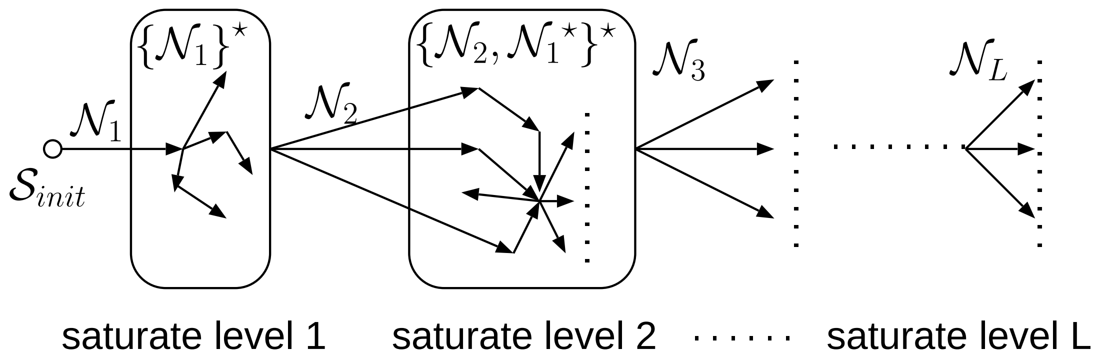
- build the L-level MDD encoding of $\mathcal{S}_{init}$ — if $|\mathcal{S}_{init}| = 1$, there is one node per level
- saturate each node at level 1: fire in them all events α s.t. $Top(\alpha) = 1$
- saturate each node at level 2: fire in them all events α s.t. $Top(\alpha) = 2$
  (if this creates nodes at level 1, saturate them immediately upon creation)
- saturate each node at level 3: fire in them all events α s.t. $Top(\alpha) = 3$
  (if this creates nodes at levels 2 or 1, saturate them immediately upon creation)
- ...
- saturate the root node at level L: fire in it all events α s.t. $Top(\alpha) = L$
  (if this creates nodes at levels $L-1$, $L-2$, ..., 1, saturate them immediately upon creation)

Saturation vs. BFS

Breadth-first search (BFS):



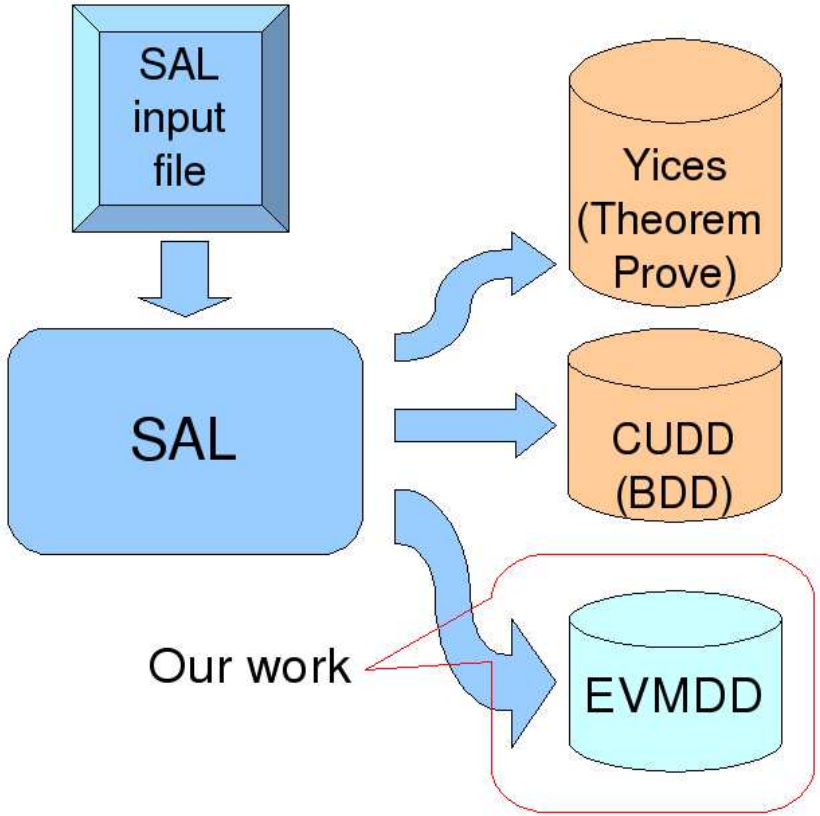
Saturation:



- states are **not** discovered in breadth-first order
- enormous time and memory savings for asynchronous systems

SAL overview

SAL model checker



SAL input language

- Expression language:
  - Types
  - Operations
- Basic module:
  - Initialization
  - Transition
- Module Composition:

```
peterson: CONTEXT =
BEGIN
PC: TYPE = {sleeping, trying, critical};
process [tval : BOOLEAN]: MODULE =
BEGIN
INPUT pc2 : PC
INPUT x2 : BOOLEAN
OUTPUT pc1 : PC
OUTPUT x1 : BOOLEAN
INITIALIZATION pc1 = sleeping
TRANSITION
[
pc1 = sleeping --> pc1' = trying; x1' = x2 = tval
[]
pc1 = trying AND (pc2 = sleeping OR x1 = (x2 /= tval))
--> pc1' = critical
[]
pc1 = critical --> pc1' = sleeping; x1' = x2 = tval
]
END;

system: MODULE =
process[FALSE]
[]
RENAME pc2 TO pc1, pc1 TO pc2,
x2 TO x1, x1 TO x2
IN process[TRUE];
```

SAL input language: expression [a]

- Type:
  - Basic types: BOOLEAN, INTEGER (within a given range), REAL
  - ENUMERATION :

    `FORKSTATE : TYPE = {available, not_available};`
  - ARRAY :

    `ARRAY R OF FORKSTATE`

- Operation:
  - Boolean operations: AND, OR, NOT
  - Arithmetic operations: $>, >=, <, <=, ==, +, -$
  - Array selection: $[\,]$
  - Undeterministic statement: IN

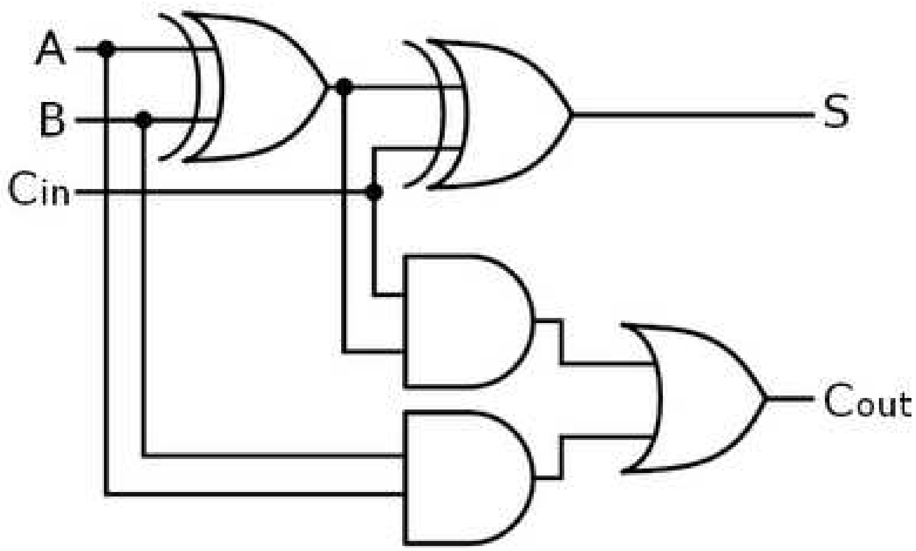
    $x \;\; \mathbf{IN} \;\; \{0..5\}$

---

[a] Yellow: supported feature; Red: unsupported feature

Arithmetic operations in BDD

Full adder: $A + B + C_{in}$

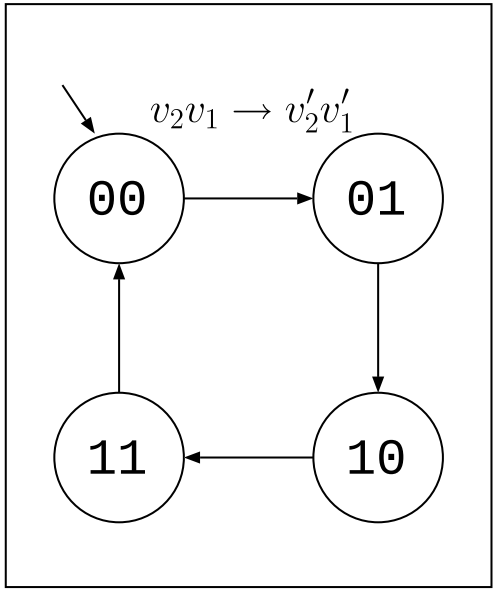


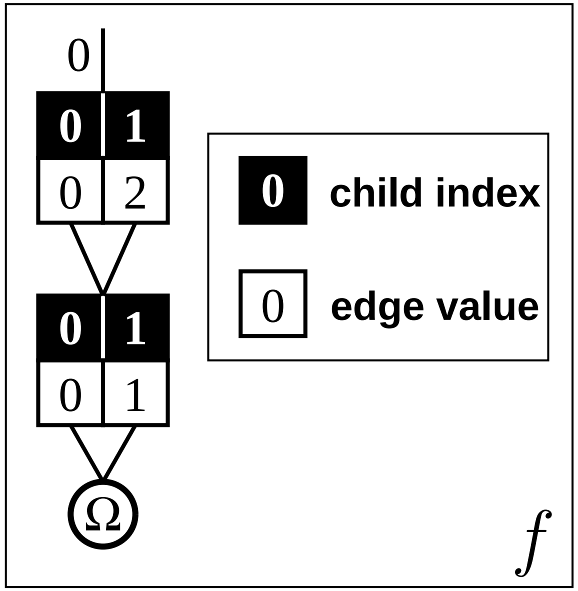
- All the operations are synthesized into binary logic.
- According to the previous experience, BDDs perform good in random logic (control-flow), but not compact in arithmetic operations (data-flow), especially multiplication.

EVMDD

Using Edge-valued multi-value decision diagrams (EVMDD) to encode a function:

$$f : \mathcal{S} \mapsto \mathbb{Z}$$



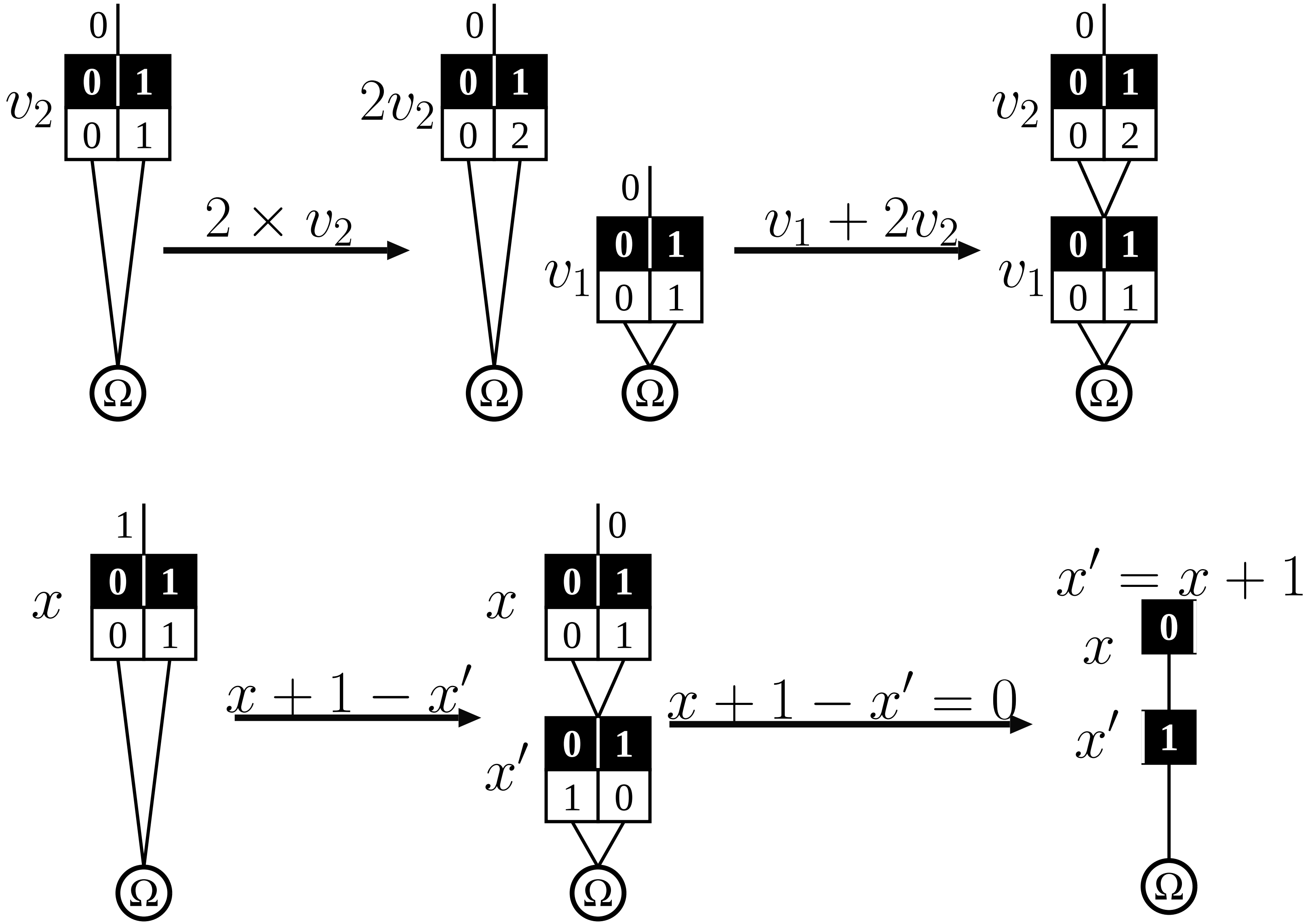


$$f(\langle 0,1\rangle) = 0 + 1 = 1$$

$$f(\langle 1,0\rangle) = 2 + 0 = 2$$

$$f(\langle 1,1\rangle) = 2 + 1 = 3$$

EVMDD-based arithmetic operations



EVMDD-based arithmetic operations

- $\langle MDD \rangle$ and $\langle EVMDD \rangle$ typing:
  - Base:
    Boolean variable := $\langle MDD \rangle$ Int variable := $\langle EVMDD \rangle$
  - MDD operations: $\wedge, \vee, /$
    $\langle MDD \rangle\ MDDOper\ \langle MDD \rangle := \langle MDD \rangle$
  - Arithematic operations: $+, -, \times, /$
    $\langle EVMDD \rangle\ ArithOper\ \langle EVMDD \rangle := \langle EVMDD \rangle$
  - Predicate: $==, >, >=, <, <=$
    $Predicate(\langle EVMDD \rangle, ...) := \langle MDD \rangle$

- Theoretical results:
  - Space complexity: For any function f , the number of nodes of the EVMDD representing f is at most the number of nodes of the MTMDD representing the same function f.
  - Time complexity: The number of recursive calls of the generic apply algorithm for MTMDDs is equal to that for EVMDDs representing the same function.

SAL input language: basic module [a]

- Initialization:
  - INITIALIZATION
- Definition:
  - DEFINITION
- Transitions:

  \- Statement:
  - Next-state variable: `x'`
  - Assignment: `x'=x+1`
  - Guarded Commands: $Guard \rightarrow Assignment$
  - IF statement

  \- Statement composition:
  - `stat1 [] stat2`

    $$stat1 \vee stat2$$
  - `stat1 || stat2`

    $$stat1 \wedge stat2$$

[a] Yellow: supported feature; Red: unsupported feature

SAL input language: module compposition [a]

- Asynchronous composition : `M1[]M2`
  - Initialization: combining the initializations in different modules.

$$\mathit{init\_M1} \wedge \mathit{init\_M2}$$

  - Transition: the union of transition definitions.

$$\mathit{trans\_M1\_stat1} \vee \cdots \vee \mathit{trans\_M2\_stat1} \vee \cdots$$

- Synchronous composition : `M1||M2`
  - Difficulty: applying saturation on

$$(\mathit{trans\_M1\_stat1} \vee \cdots) \wedge (\mathit{trans\_M2\_stat1} \vee \cdots)$$

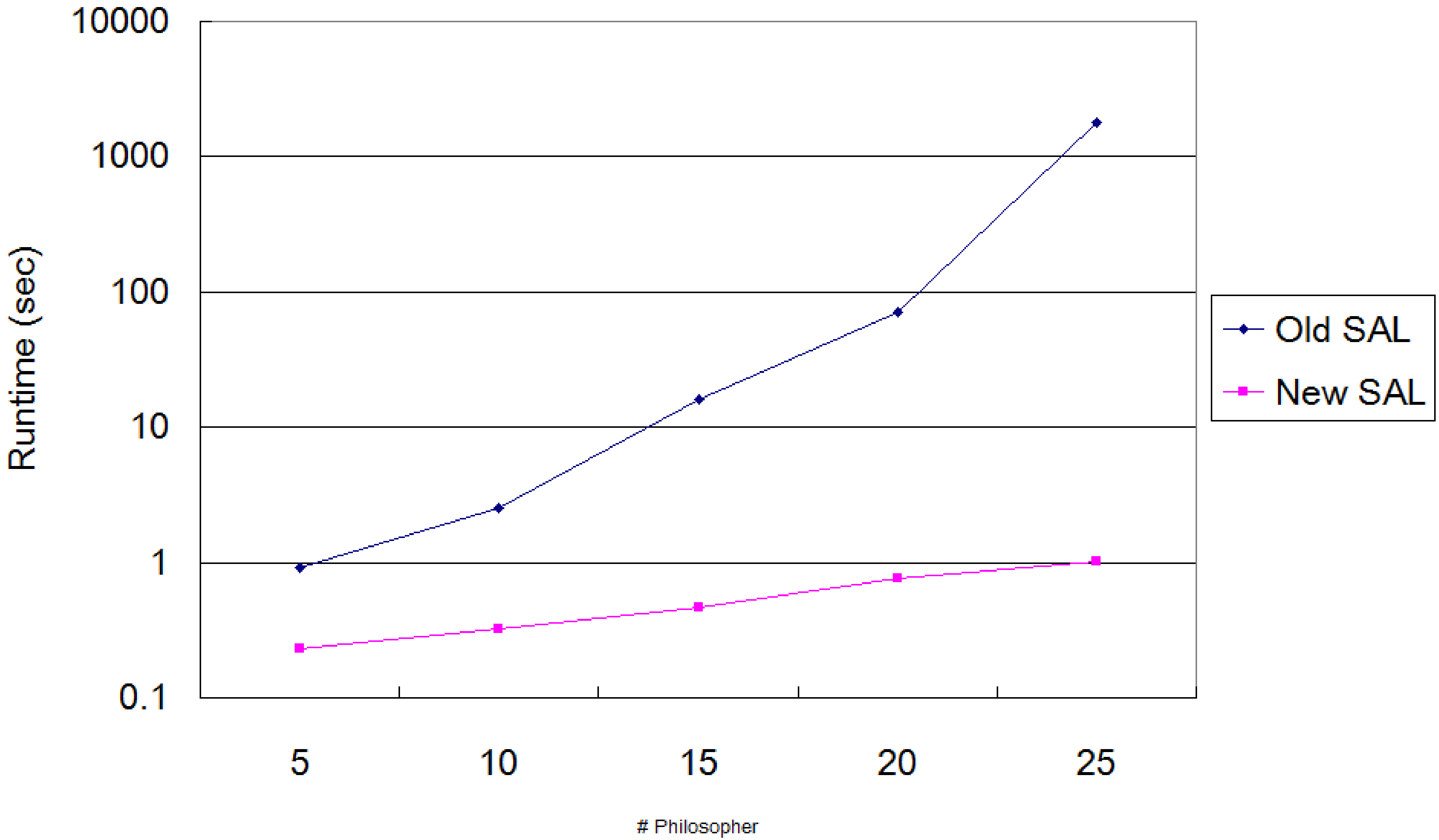
[a] Yellow: supported feature; Red: unsupported feature

Supported syntax summary

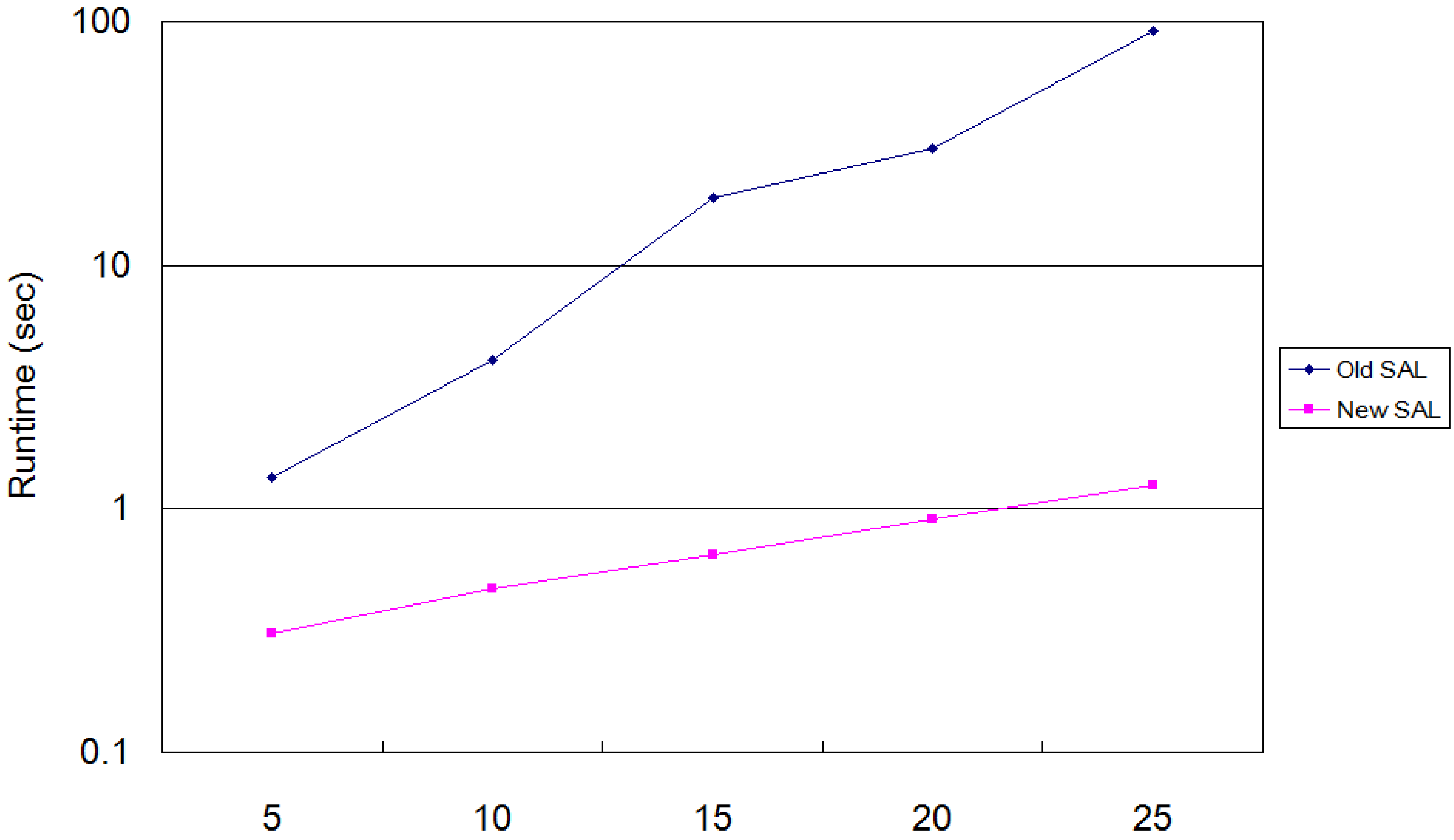
Syntax in SAL	New SAL	EVMDD-SMC
Variable Type:		
Basic type	√	√
Enumeration	√	×
Array	√	×
Record	×	×
Transition language:		
Assignment	√	√
Guarded command	√	√
Undeterministic assignment	√	×
IF statement	×	×
Module:		
Parameter	√	×
Module composition	Asynchronous only	×
Other feature:		
Variable ordering	Ongoing	×

Experimental results

Runtime: Dinning Philosopher



Runtime: Round Robin



Summary and future work

- The new SAL supports more expressive syntax than our prototype tool.
- EVMDD provides an elegant and efficient way of handling the arithmetic operations.
- The new SAL using saturation algorithm consistently improves on the old implementation in state-space generation for asynchronous systems.

Future work:

- Variable ordering
- More powerful preprocesses in SAL and its contribution to model checking
- State-space generation for synchronous systems
- Capturing and exploiting more locality in the asynchrous systems

Thank you!

Q & A