The new SAL symbolic model checker

Yang Zhao

Department of Computer Science and Engineering

University of California at Riverside

, 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 decision diagrams (EVMDDs) 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

ructured *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*

- $\mathcal{N}_\alpha : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ $\mathbf{j} \in \mathcal{N}_\alpha(\mathbf{i})$ iff state $\mathbf{j}$ can be reached by *firing* event α in state $\mathbf{i}$
- $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ $\mathcal{N}(\mathbf{i}) = \bigcup_{\alpha \in \mathcal{E}} \mathcal{N}_\alpha(\mathbf{i})$ *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

ality 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 $\mathcal{E}_k$:

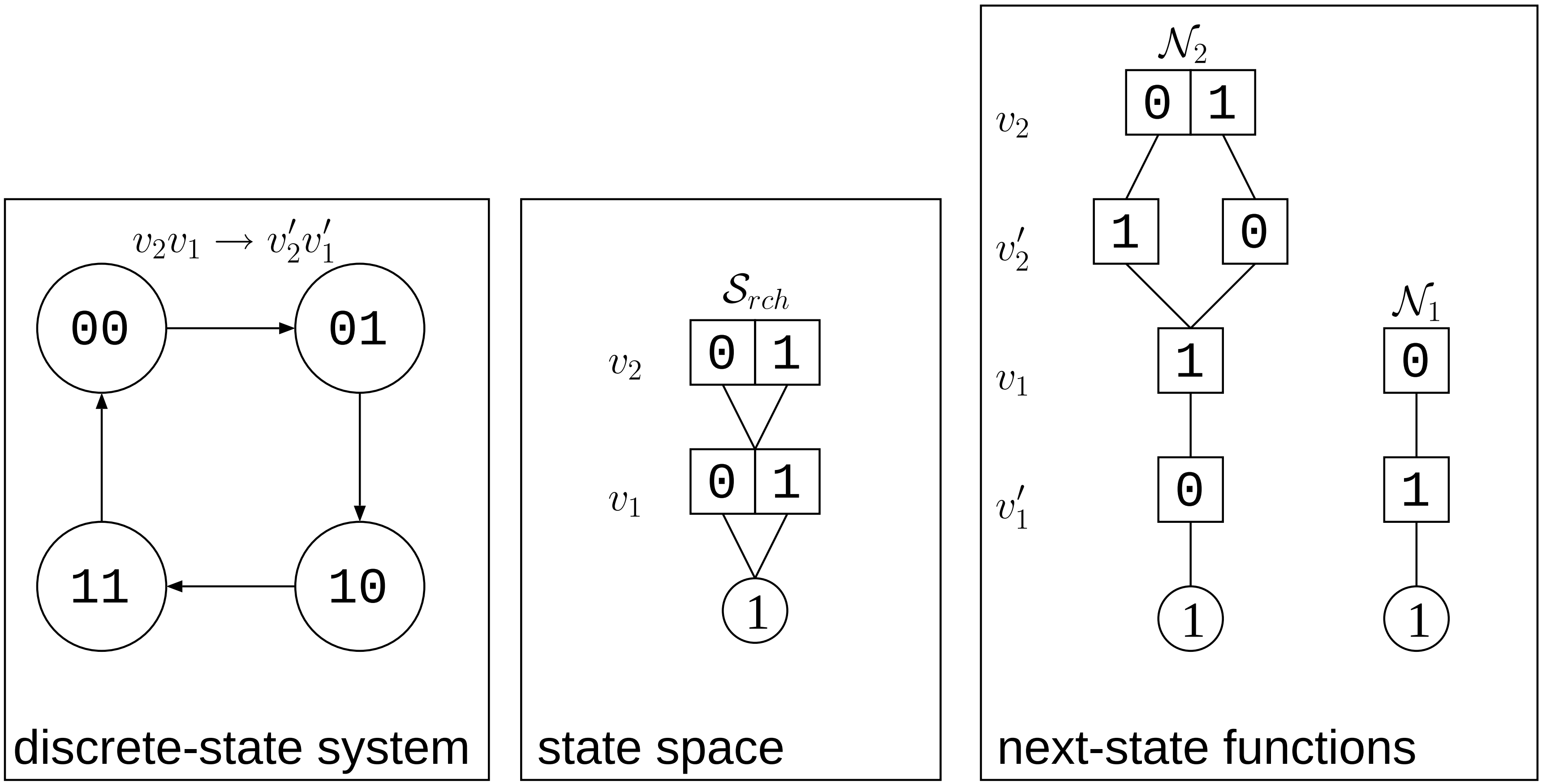
$$\mathcal{N}_k = \bigcup_{\alpha \in \mathcal{E}_k} \mathcal{N}_\alpha$$

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

$supp(\alpha) = \{x, y, z\}$

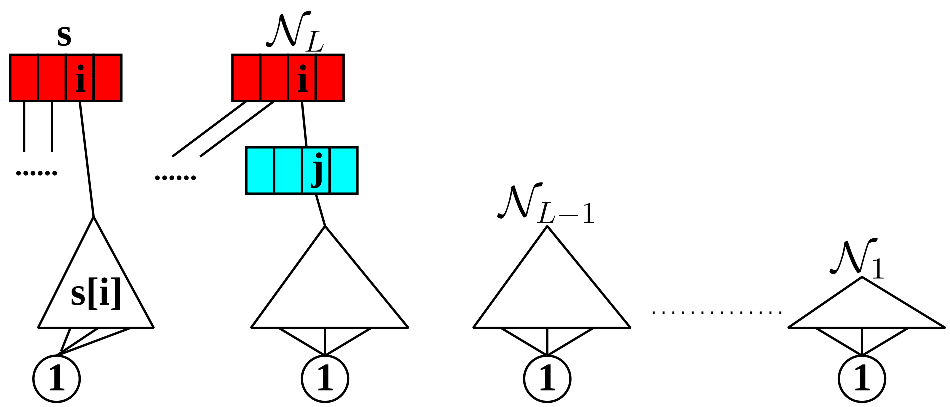
$Top(\alpha) = Max(x.lvl, y.lvl, z.lvl)$

Example: 2-bit counter



aturation: an iteration strategy based on the model structure

D 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)

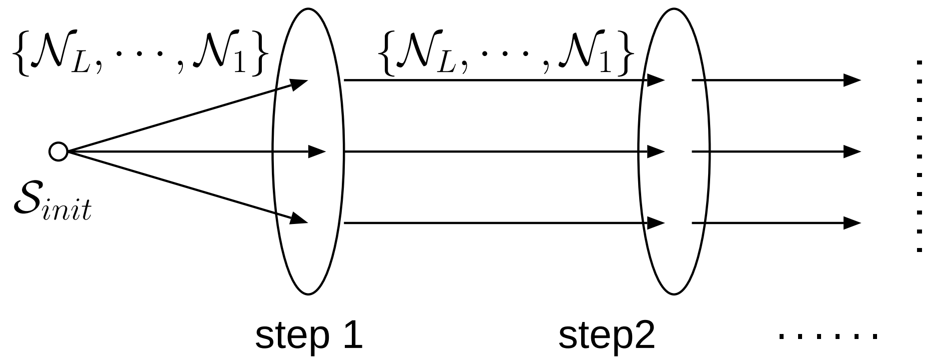
saturate each node at level 3: fire in them all events α s.t. $Top(\alpha) = 3$
(if this creates nodes at levels 2 or 1, saturate them immediately upon creation)

$\ldots$

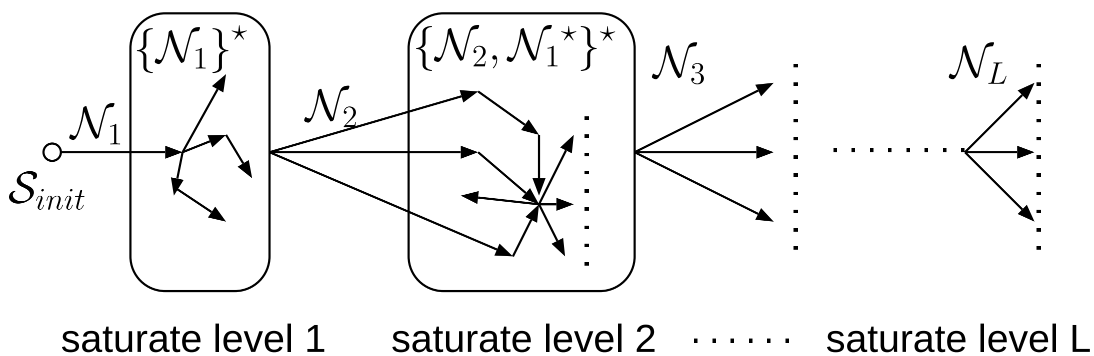
saturate the root node at level L: fire in it all events α s.t. $Top(\alpha) = L$
(if this creates nodes at levels $L{-}1$, $L{-}2$, $\ldots$, 1, saturate them immediately upon creation)

Saturation vs. BFS

adth-first search (BFS):



uration:

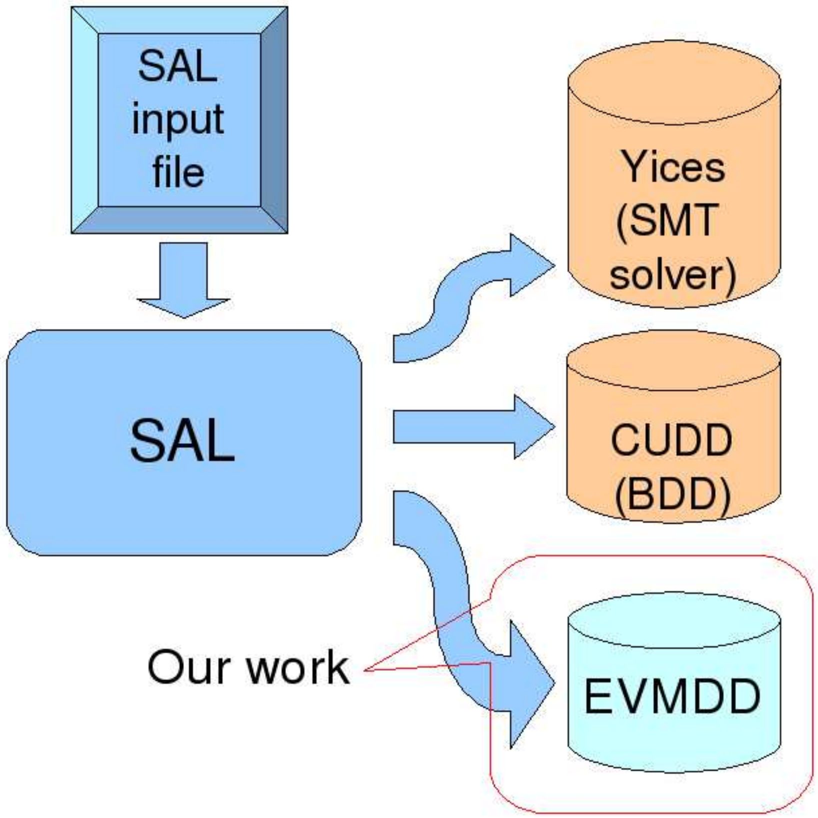


states are **not** discovered in breadth-first order

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]

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

Types:

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

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

  `ARRAY R OF FORKSTATE`

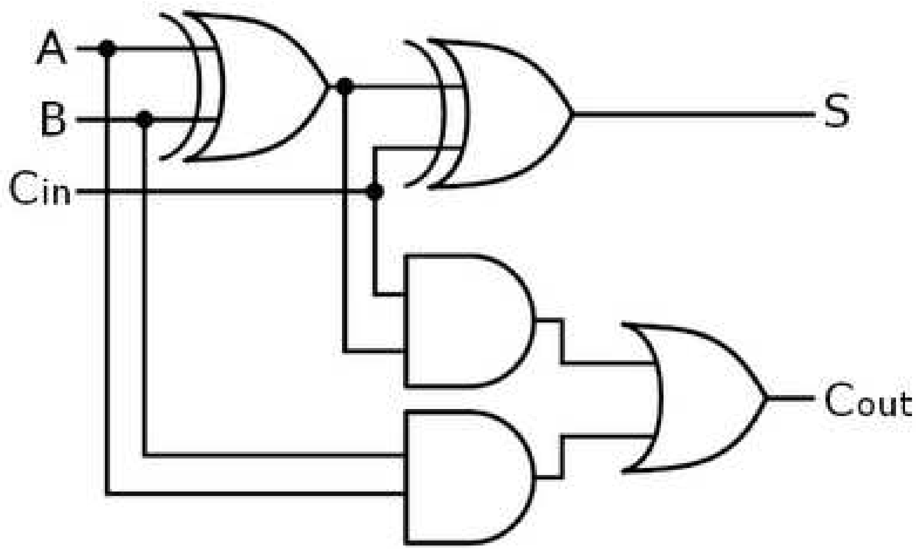
Operations:

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

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

Arithmetic operations in BDD

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



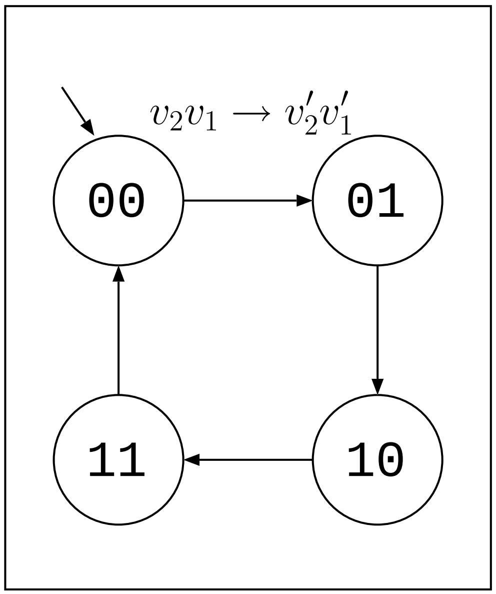
All the operations are synthesized into binary logic.

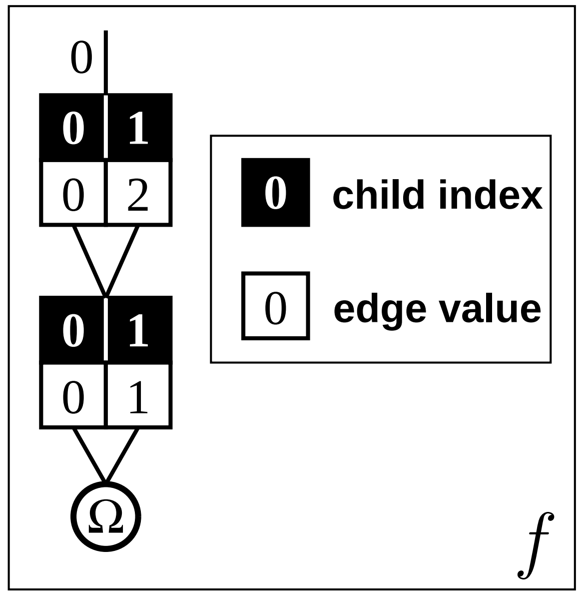
BDDs perform good in random logic (control-flow), but not compact in arithmetic operations (data-flow), especially multiplication.

EVMDD

ng Edge-valued multi-valued decision diagrams (EV+MDD) to encode a function:

$$f : \mathcal{S} \mapsto \mathbb{Z} \cup \{+\infty\}$$



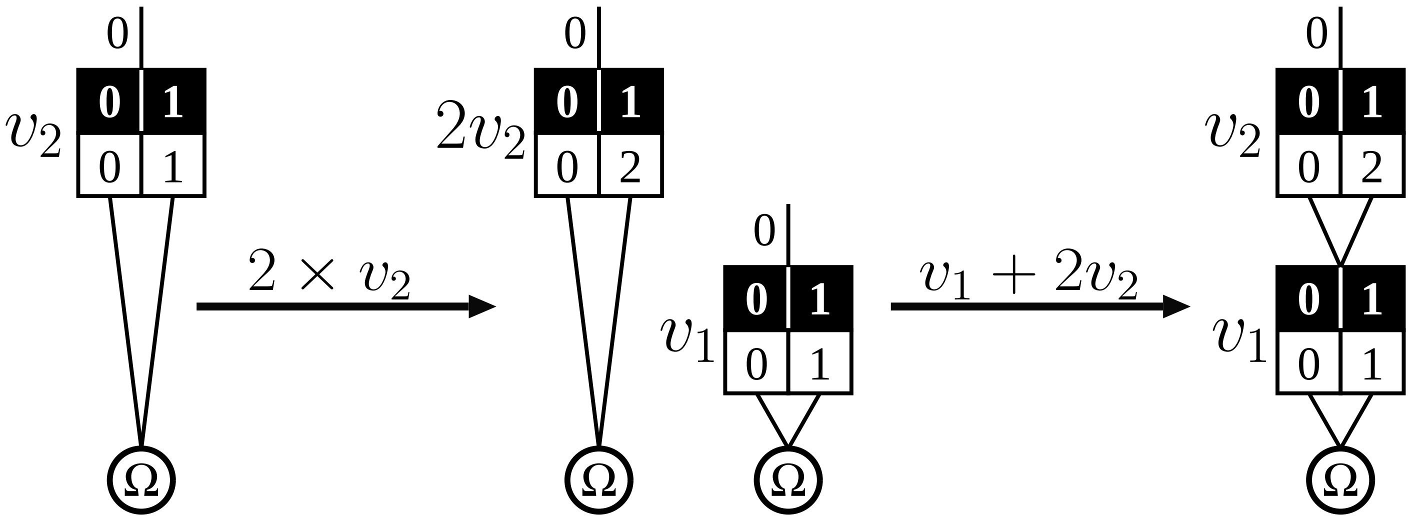


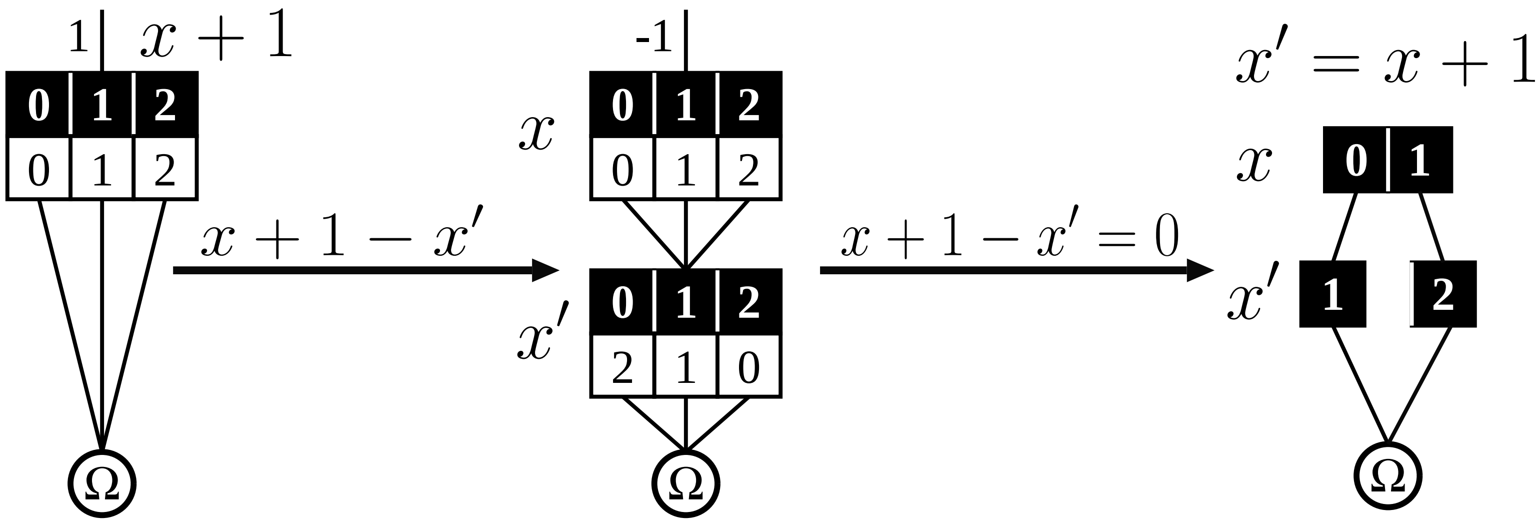
$$f(\langle 0,1 \rangle) = 0 + 1 = 1$$

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

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

EVMDD-based arithmetic operations





$$v_1, v_2 \in \{0, 1\}, x \in \{0, 1, 2\}$$

EVMDD-based arithmetic operations

$\langle MDD \rangle$ and $\langle EVMDD \rangle$ typing:

- Base:
  TRUE, FALSE $:= \langle MDD \rangle$ $\quad \mathbb{Z} := \langle EV\,MDD \rangle$
  Boolean variable $:= \langle MDD \rangle$ $\quad$ Int variable $:= \langle EV\,MDD \rangle$
- MDD operations: $\cap, \cup, \setminus$
  $MDDOper : \ (\langle MDD \rangle, \cdots) \rightarrow \langle MDD \rangle$
- Arithmetic operations: $+, -, \times, /$
  $ArithOper : \ (\langle EVMDD \rangle, \langle EVMDD \rangle) \rightarrow \langle EVMDD \rangle$
- Relational operations: $==, >, >=, <, <=$
  $RelOper : \ (\langle EVMDD \rangle, \langle EVMDD \rangle) \rightarrow \langle MDD \rangle$

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 MT-MDDs is equal to that for EVMDDs representing the same function.

SAL input language: basic module [a]

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

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`

$$stat1 \wedge stat2$$

SAL input language: module compposition [a]

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

Asynchronous composition : `M1[]M2`

- Initialization: combining the initializations in different modules.

$$init\_M1 \wedge init\_M2$$

- Transition: the union of transition definitions.

$$trans\_M1\_stat1 \vee \cdots \vee trans\_M2\_stat1 \vee \cdots$$

Synchronous composition : `M1||M2`

- Difficulty: applying saturation on

$$(trans\_M1\_stat1 \vee \cdots) \wedge (trans\_M2\_stat1 \vee \cdots)$$

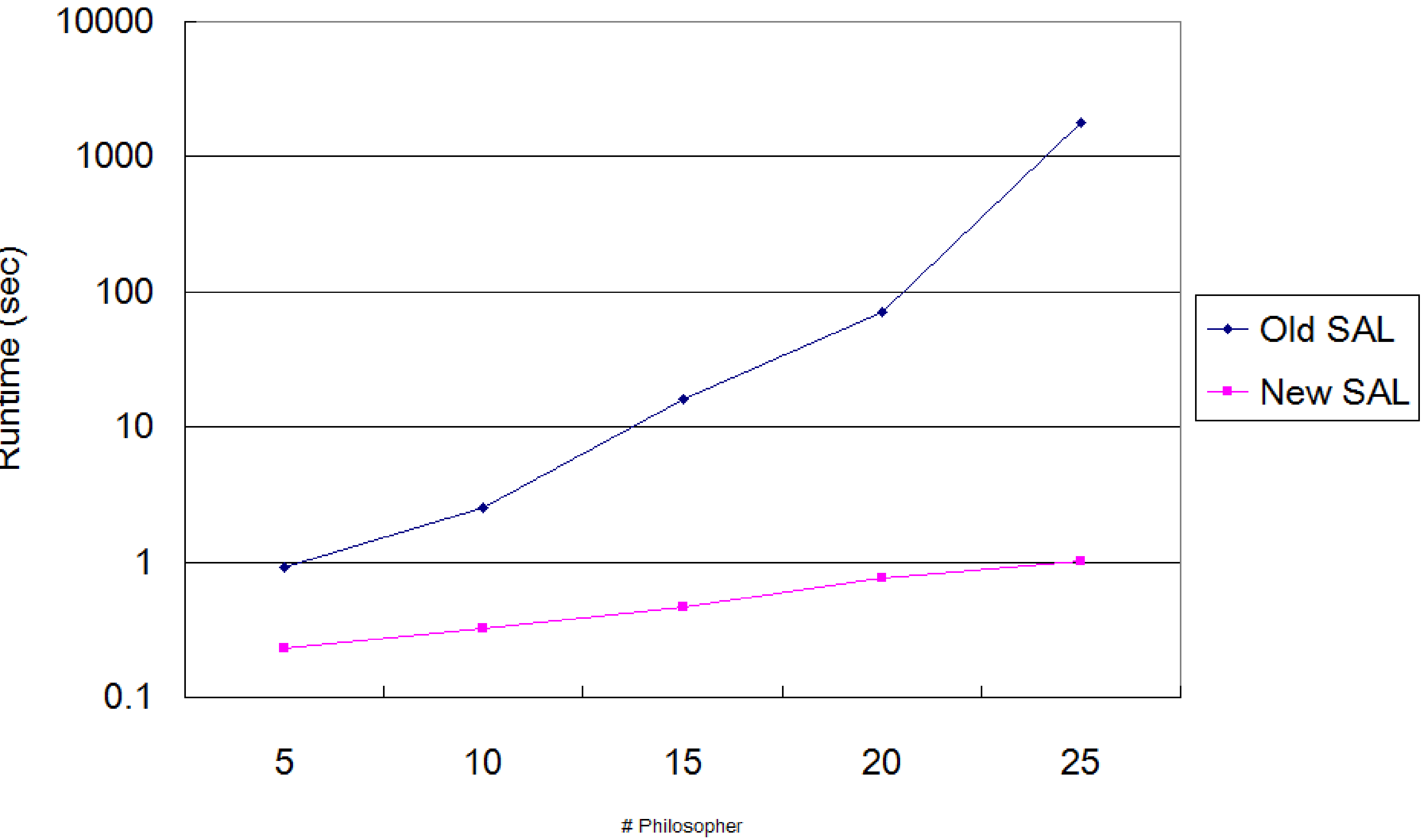
Supported syntax summary

Syntax in SAL	New SAL	EVMDD-SMC
Variable Type:		
Basic type	$\surd$	$\surd$
Enumeration	$\surd$	$\times$
Array	$\surd$	$\times$
Record	$\times$	$\times$
Transition language:		
Assignment	$\surd$	$\surd$
Guarded command	$\surd$	$\surd$
Undeterministic assignment	$\surd$	$\times$
IF statement	$\times$	$\times$
Statement composition	$\star$	$\star$
Module:		
Parameter	$\surd$	$\times$
Module composition	$\star$	$\times$
Other feature:		
Variable ordering	Ongoing	$\times$

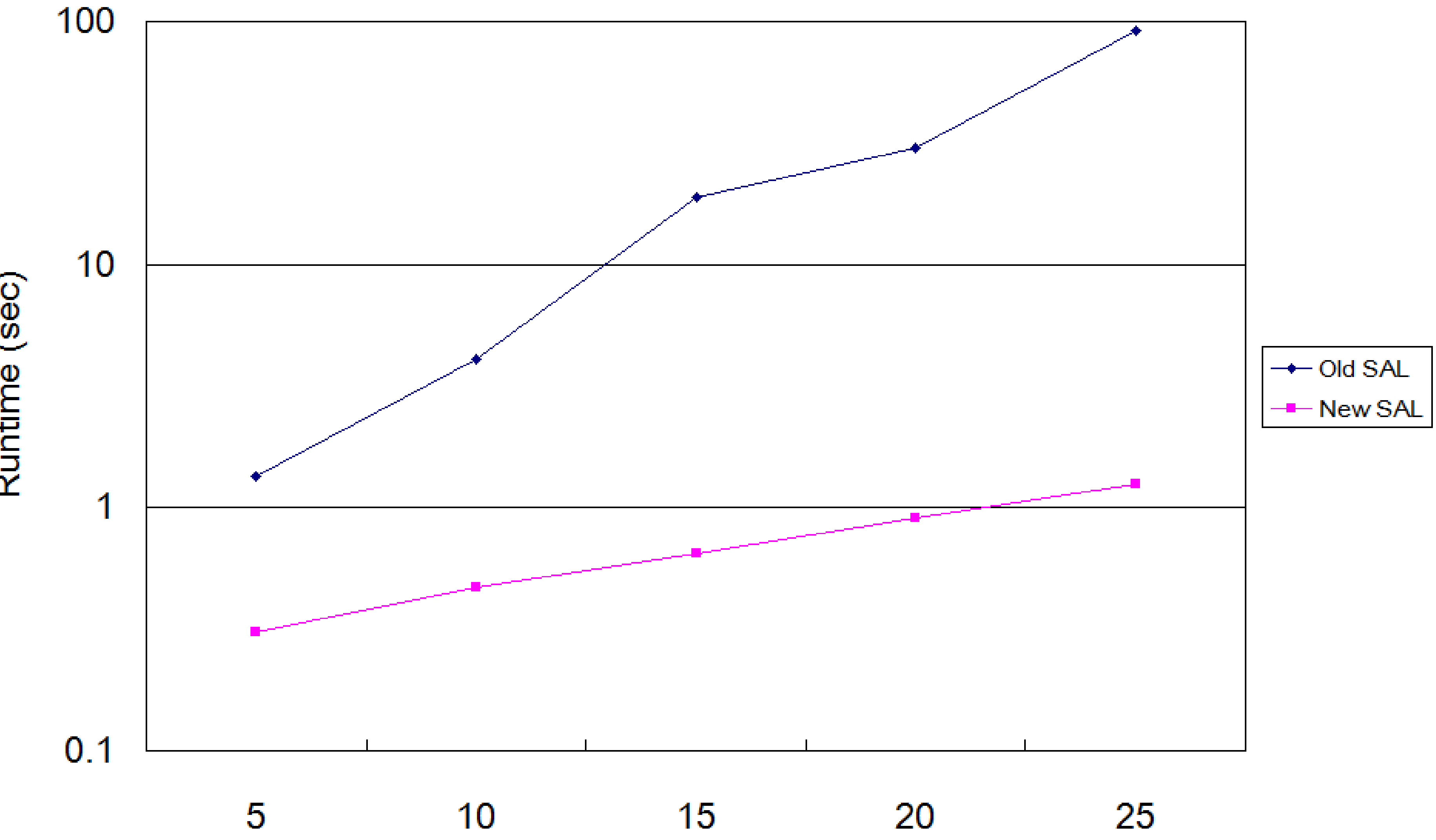
: *Asynchronous only*

Experimental results

Runtime: Dinning Philosopher



Runtime: Round Robin



New SAL vs. EVMDD-SMC

Model size	Reachable states	EVMDD-SMC ss (in s)	EVMDD-SMC total (in s)	new SAL ss (in s)	new SAL total(in s)
kanban 20	8×10^{11}	0.01	0.03	0.01	0.26
kanban 100	1×10^{19}	0.88	2.99	0.87	1.25
kanban 400	6×10^{25}	74.33	273.43	73.28	76.72
knights 5	6×10^{7}	0.19	0.27	0.22	0.74
knights 7	1×10^{15}	2.00	2.94	1.95	3.28
knights 9	8×10^{24}	9.43	15.43	9.43	13.40
phils 300	1×10^{188}	0.01	0.17	0.01	66.57
phils 400	6×10^{250}	0.01	0.29	0.02	120.80
phils 500	3×10^{313}	0.01	0.39	0.03	200.07
robin 40	9×10^{13}	0.06	0.21	0.07	2.51
robin 100	2×10^{32}	0.96	2.92	0.95	17.63
robin 200	7×10^{62}	7.52	25.21	7.63	94.43
slot 100	2×10^{105}	3.1	3.92	3.33	25.56
slot 200	8×10^{211}	26.14	34.28	26.64	150.53

$\star$ Old SAL can not complete any of these models in 1500 seconds.

Summary and future work

nmary:

- 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.

ure work:

- Variable ordering
- State-space generation for synchronous systems
- Capturing and exploiting more locality in the asynchrous systems

FILE:

Thank you!

Q & A