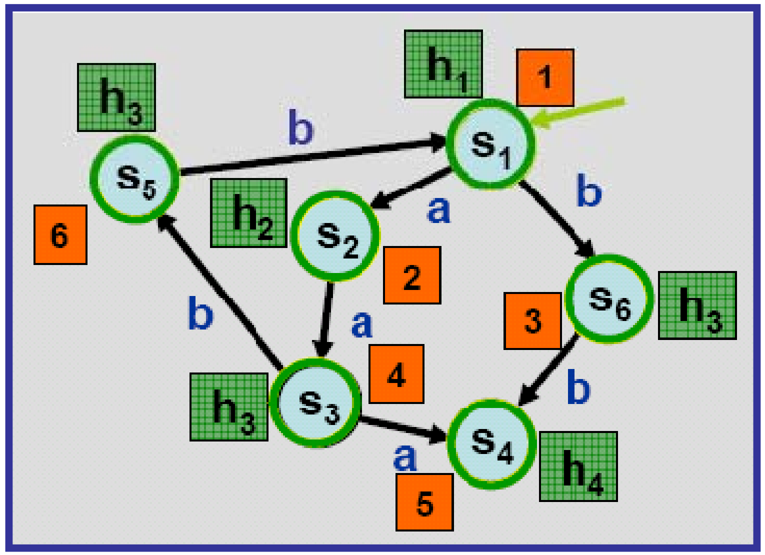
Advanced State Space Methods

Overview

1. G.E. Gallasch, J. Billington, S. Vanit-Anunchai, and L.M. Kristensen. Checking Safety Properties On-the-fly with the Sweep-Line Method. International Journal on Software Tools for Technology Transfer, Vol 9, No. 3-4, pp. 371-392. Springer-Verlag, 2007.
2. M. Westergaard, L.M. Kristensen, G.S. Brodal and L. Arge. The ComBack Method - Extending Hash Compaction with Backtracking. In Proc. of 28th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency, Vol. 4546 of Springer Lectures Notes in Computer Science, pp. 445-464. Springer-Verlag, 2007.
3. S. Evangelista, M. Westergaard, and L.M. Kristensen. The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. *ToPNoC*, 2009. To appear.
4. S. Evangelista, M. Westergaard, and L.M. Kristensen. The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. In *CPN'2008*, 2008.
5. S. Evangelista. Dynamic Delayed Duplicate Detection for External Memory Breadth-First Search. In Proc. of SPIN'2008 Workshop on Model Checking of Software. LNCS. Springer-Verlag, 2008.
6. S. Evangelista and L.M. Kristensen. Search-Order Independent State Caching. In *CPN'2009*, 2009.
7. S. Evangelista and C. Pajault. Solving the Ignoring Problem for Partial Order Reduction. Submitted to STTT.
8. M. Westergaard, S. Evangelista, and L.M. Kristensen. ASAP: An Extensible Platform for State Space Analysis. In *ATPN'2009*, volume 5606 of *LNCS*, pages 303-312. Springer, 2009.
9. S. Evangelista and L.M. Kristensen. Dynamic State Space Partitioning for External Memory Model Checking. In *FMICS'2009*, volume 5825 of *LNCS*, pages 70-85. Springer, 2009.

The ComBack Method - Extending Hash Compaction with Backtracking



The Hash Compaction Method

[Wolper&Leroy'93, Stern&Dill'95]

- Relies on a hash function H for memory efficient representation of visited (explored) states:

$H : S \rightarrow \{0,1\}^w$

s ➔ 0110001100011000111000001110001 01

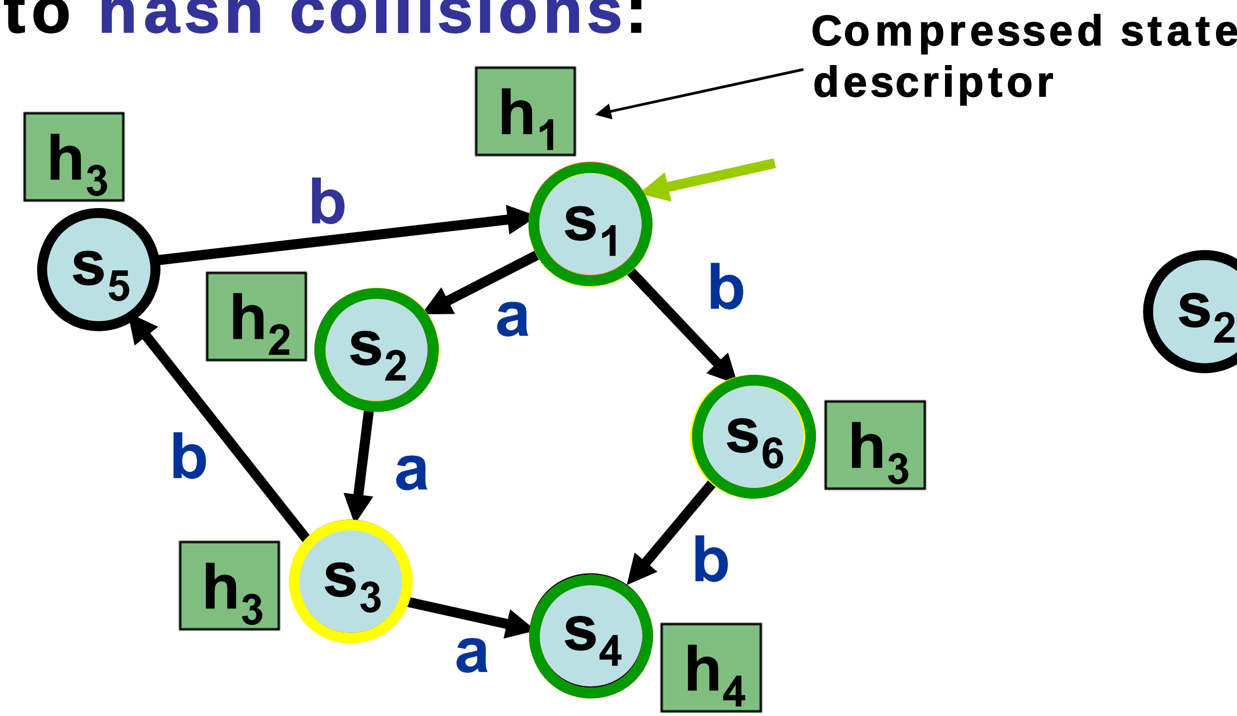
Full state descriptor (100-1000 bytes)

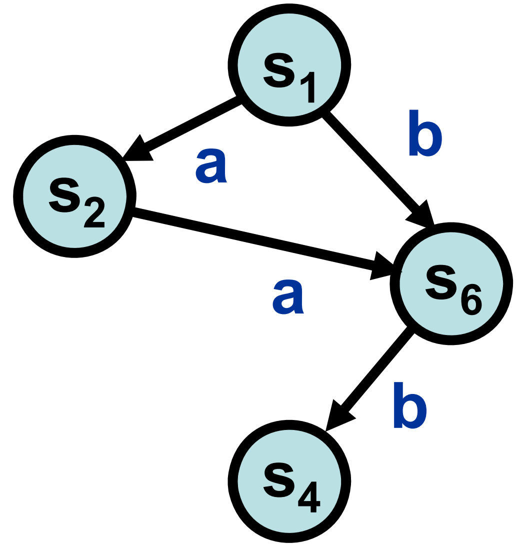
Compressed state descriptor (4-8 bytes)

- Only the compressed state descriptor is stored in the state table of visited states.

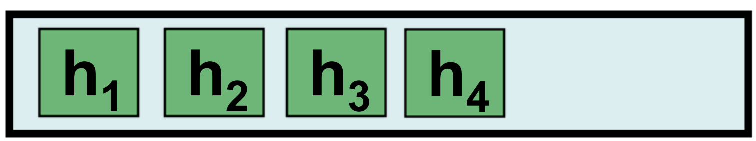
Example: Hash Compaction

- Cannot guarantee full state space coverage due to hash collisions:





State table:



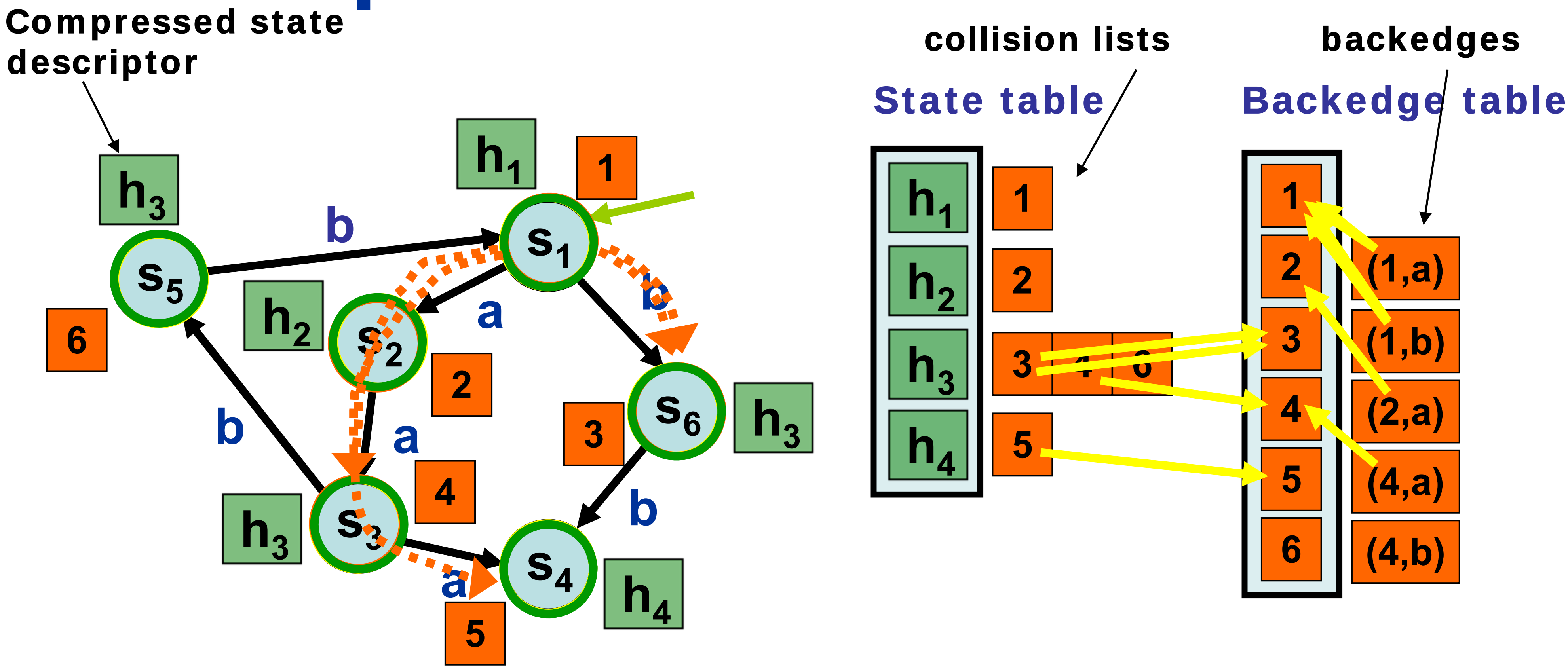
The Comback Method

- Reconstruction of full state descriptors to resolve hash collisions during state space exploration.
- Reconstruction is achieved by augmenting the hash compaction method:
  - A state number is assigned to each visited state.
  - The state table stores for each compressed state descriptor a collision list of state numbers.

    to detect (potential) hash collisions
  - A backedge table stores a backedge for each state number of a visited state.

    to reconstruct full state descriptors

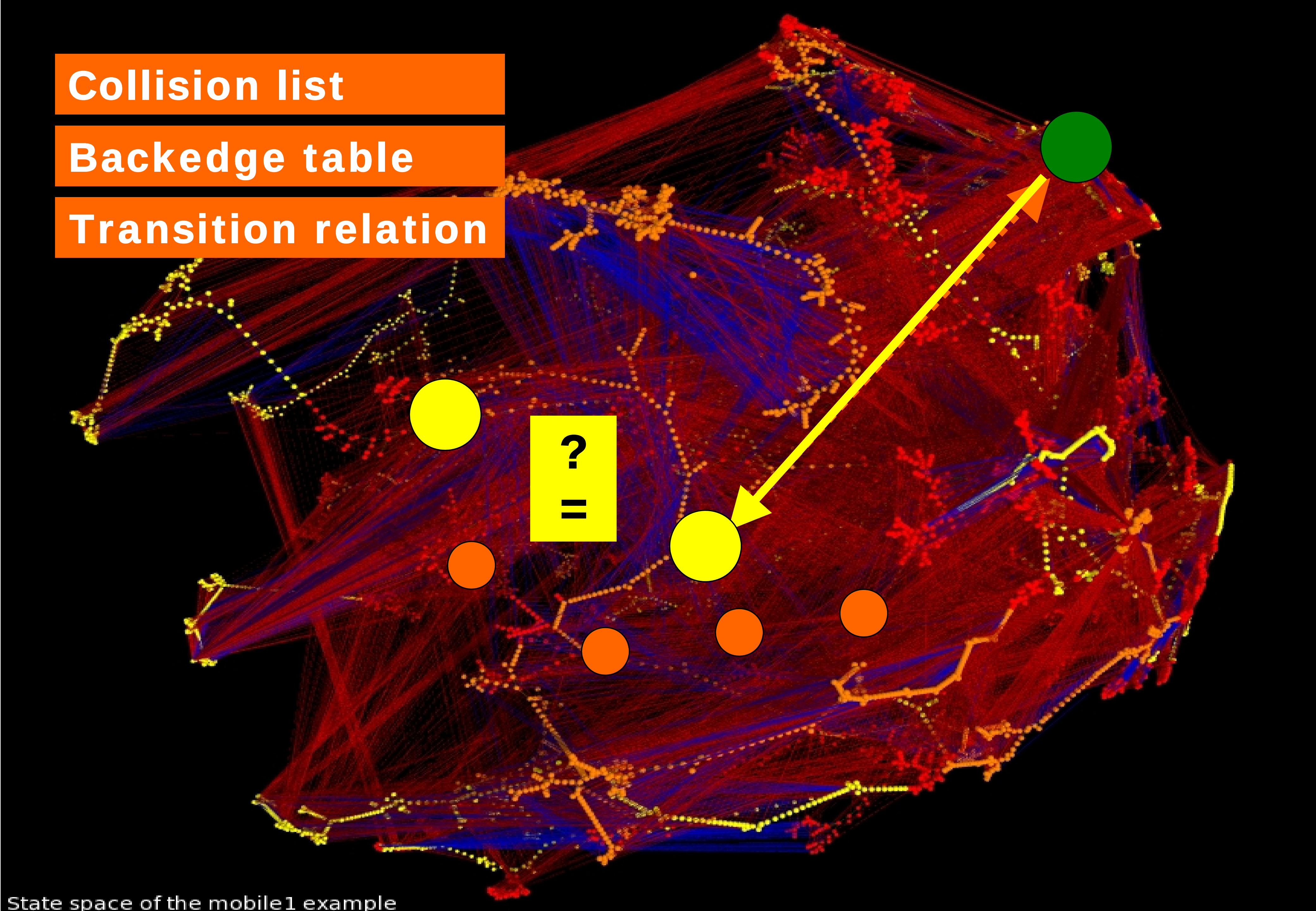
Example: The ComBack Method



State Reconstruction

3 (1,b) ➔ $S_6 \neq S_3$ 4 (2,a) (1,a) ➔ $S_3 \neq S_5$

3 (1,b) ➔ $S_6 \neq S_5$ 5 (4,a) (2,a) (1,a) ➔ $S_4 = S_4$


Collision list
Backedge table
Transition relation
?
=
State space of the mobile1 example

Main Theorem

- ComBack algorithm terminates after having processed all reachable states exactly one.

- The elements in the state table and the backedge table can be represented using:

$$|\text{reach}(s_I)| \cdot (w_H + 3 \cdot \lceil \log_2 |\text{reach}(s_I)| \rceil + \lceil \log_2 |T| \rceil) \ bits$$

Overhead compared to hash compaction

- Number of state reconstructions bounded by:

$$\max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \sum_{s \in \text{reach}(s_I)} in(s \ )$$

Implementation

- **Prototype implemented on the ASCoVeCo State Space Analysis Platform (ASAP):**
  - State table with collision lists implemented using a hash table.
  - Backedge table implemented as a dynamic array.
  - Compressed state descriptors and state numbers: 31 bit UI.
  - Breadth-first (BFS) and depth-first search (DFS) implemented.
  - Variant of ComBack method with caching implemented.
- **Performance of ComBack method compared to:**
  - Standard full state space exploration (BFS and DFS).
  - Hash compaction method (BFS and DFS).

Summary of Experimental Results

ComBack performance relative to standard DF full state space exploration				DFS		BFS	
Model	Method	Nodes	Arcs	%Time	%Space	%Time	%Space
DB	ComBack	196,832	1,181,001	37	10	39	26
	HashComp	196,798	1,180,790	18	3	21	21
	Standard	196,832	1,181,001	100	100	106	100
SW	ComBack	215,196	1,242,386	178	42	258	48
	HashComp	214,569	1,238,803	92	12	103	23
	Standard	215,196	1,242,386	100	100	111	100
TS	ComBack	107,648	1,017,490	383	85	198	30
	HashComp	107,647	1,017,474	93	75	96	24
	Standard	107,648	1,017,490	100	100	106	73
ERDP	ComBack	207,003	1,199,703	180	34	353	42
	HashComp	206,921	1,199,200	93	6	100	21
	Standard	207,003	1,199,703	100	100	115	101
ERDP	ComBack	4,277,126	31,021,101	-	-	-	-
	HashComp	4,270,926	30,975,030	-	-	-	-

Conclusions

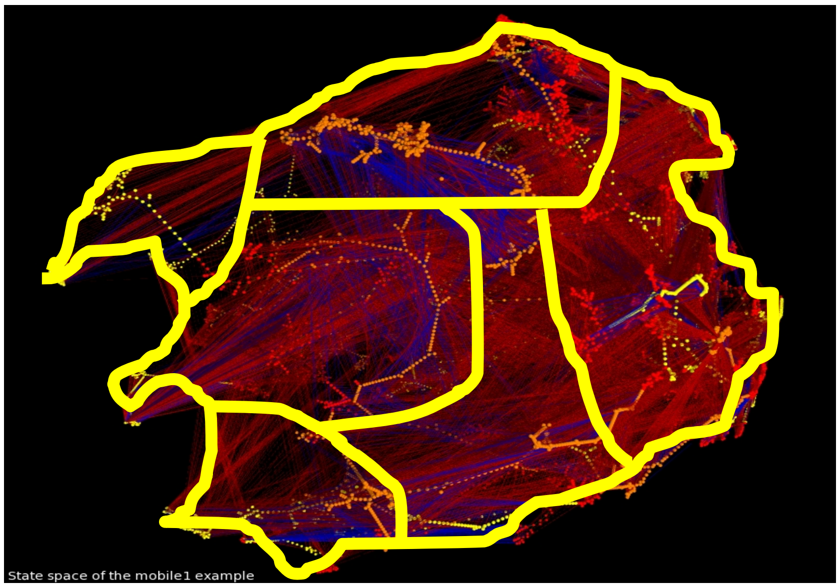
- **ComBack method for alleviating state explosion:**
  - Extension of the hash compaction to guarantee full coverage.
  - Search-order independent and transparent state reconstruction.

- **Practical experiments:**
  - Uses more time and space than hash compaction, less memory than standard full state space exploration.
  - ComBack method suited for late phases of the verification process.

Dynamic State Space Partitioning for External Memory Model Checking

State Space Partitioning

- **The state explosion problem can be addressed by dividing the state space into partitions:**



Distributed model checking:

- State space exploration is conducted using a set of machines / processes.
- Each process is responsible for exploring the states of a partition.

External-memory model checking:

- One partition is loaded into memory at a time.
- The remaining partitions are stored in external memory (disk).

- **Requires a partition function mapping from the set of states to partitions.**

External-Memory Algorithm

- Uses a **queue Q** of unprocessed states, a set of **visited states V**, and a **file F** for each partition:

for i **in** 1 **to** N **do**
    $\mathcal{Q}_i := \emptyset \ ; \ \mathcal{V}_i := \emptyset \ ; \ \mathcal{F}_i := \emptyset$
$\mathcal{Q}_{part(s_0)}.enqueue(s_0)$
while $\exists i : \neg \mathcal{Q}_i = \emptyset$ **do**
    $i := longestQueue()$
    $\mathcal{F}_i.load(\mathcal{V}_i)$
    $search_i()$
    $\mathcal{V}_i.unload(\mathcal{F}_i)$
    $\mathcal{V}_i := \emptyset$

procedure $search_i$ **is**
    while $\mathcal{Q}_i \neq \emptyset$ **do**
        $s := \mathcal{Q}_i.dequeue()$
        if $s \notin \mathcal{V}_i$ **then**
            $\mathcal{V}_i.insert(s)$
            for e **in** $en(s)$, $s' = succ(s, e)$ **do**
                $j := part(s')$
                if $i = j$ **then** (* local transition *)
                    if $s' \notin \mathcal{V}_i$ **then** $\mathcal{Q}_i.enqueue(s')$
                else $\mathcal{Q}_j.enqueue(s')$ (* cross transition *)

Partitioning Functions

- **Desirable properties:**
  - Limit the number of cross transitions to reduce disk access and network communication.
  - Even distribution of states into partitions to ensure that all processes receives a comparable workload.

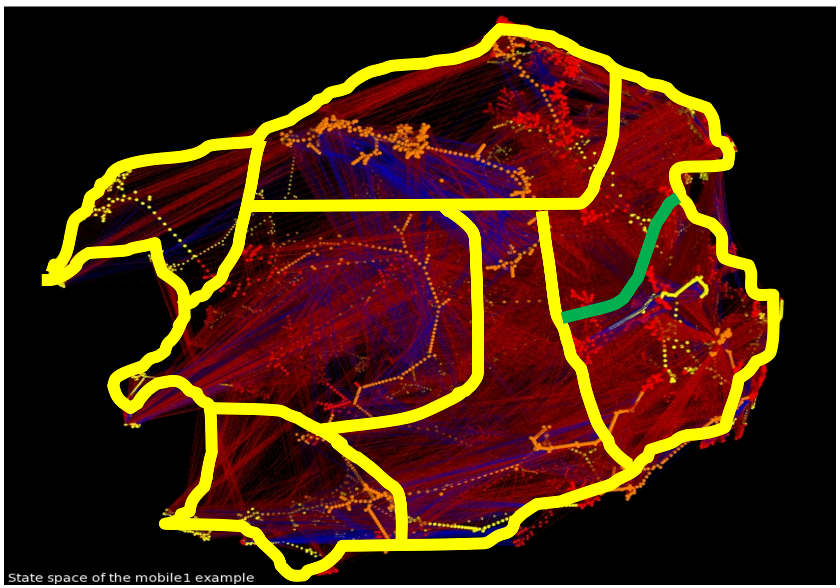
- **Main contributions of this work:**
  1. A dynamic partitioning scheme based on partition refinement and compositional partition functions.
  2. A set static and dynamic heuristics for implementing partition refinement in the context of external memory model checking.
  3. An implementation and experimental evaluation of the dynamic partitioning scheme and the associated heuristics.

Dynamic Partitioning

- Assumes that the system states can be represented as a vector of **state components**:

$$S = (C_1, C_2, \ldots, C_n)$$

- A partition is determined from a subset of the state components:

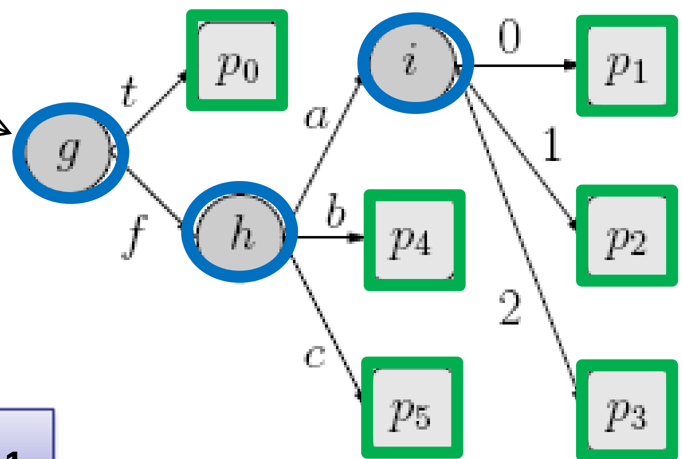


- A partition is **split** into **sub-partitions** (refined) when it exceeds the available memory.
- The refinement is realised by taking into account an additional state component.

Partitioning Diagrams

- A compositional partition function can be represented as a **partitioning diagram**:

Single root node



Branching nodes (branching functions)

Terminal nodes (state partitions)

State s

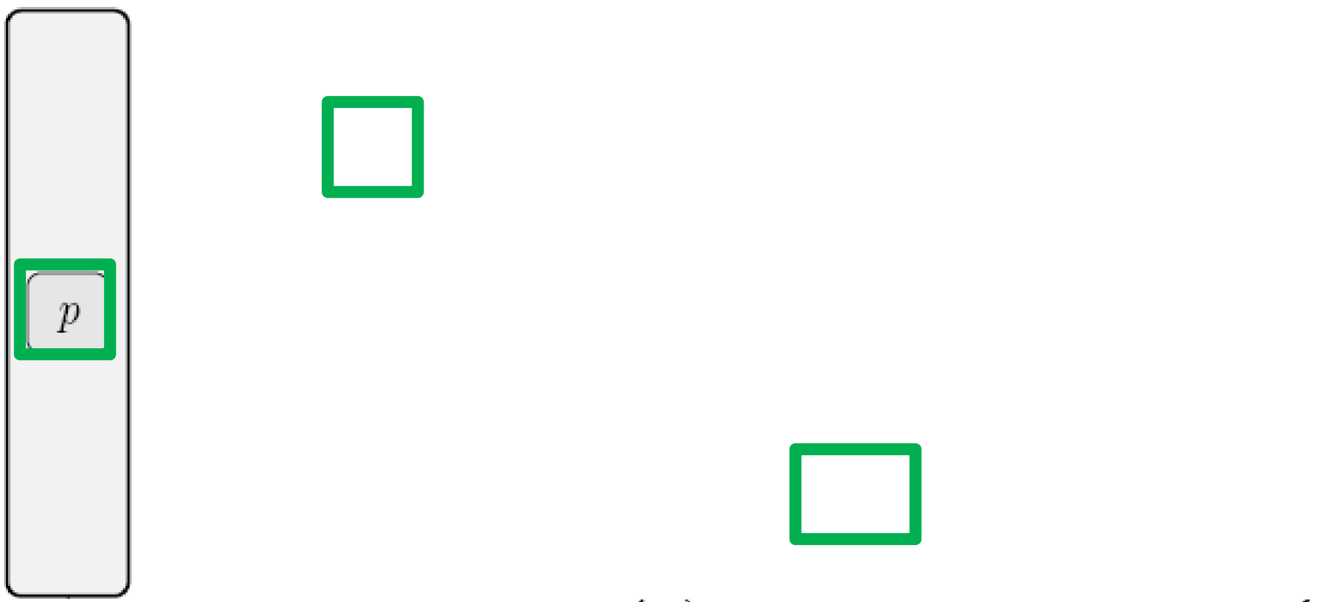
g(s)	g(s)=f
h(s)	h(s)=a
i(s)	i(s)=0

p_1

- The partition of a state is determined by applying the branching functions starting from the root.

Example: Partition Refinement

- **A state vector with three state components (b {t,f} ,c {t,f} ,i {0,1,2,3}):**



Heuristics

- **The refinement step requires the selection of state component to be used for the refinement.**

- **Static Analysis (SA):**
  - Count for each state component, the number of events in the analysed system that modifies it.
  - Among candidate components, select the component with the lowest count (to reduce cross transitions).
- **Static Sample (SS):**
  - Explore a sample of the state space and count the number of times a state component is modified (randomized search).
  - Among candidate components, select the component with the lowest count (to reduce cross transitions).

Dynamic Heuristics

- **Dynamic Randomized (DR):**
  - Picks a random state component not yet considered.
  - Serve as a baseline for the other dynamic heuristics.
- **Dynamic Event Execution (DE):**
  - Count during state space exploration the number of times a component has been modified (select lowest count).
- **Dynamic Distribution (DD):**
  - Select the component that gives the lowest standard deviation in sub-partition sizes.
- **Dynamic Distribution and Event Execution (DDE):**
  - Combines heuristics DE and DD:

$$h(C_i) = updates[i] \cdot std(C_i)$$

Experimental Context

- **Implementation in the ASAP model checking platform:**
  - The PART external memory algorithm [Bao, Jones (TACAS'05)]: uses a global hash function on the state vector.
  - A static partitioning scheme [Lerda, Sisto (SPIN'99)]: The partitions are determined from a single state component.
  - A dynamic partitioning scheme [Lerda, Visser (SPIN'01)]: partitions consists of classes that can be reassigned.

- **Experiments conducted on models from the BEEM benchmark database [Pelánek (SPIN'07)].**

Experimental Results (1)

- **Measures the number of cross transitions (CT) and disk accesses (IO):**

PART | SPIN'99 | SPIN'01

	Static		Dynamic		Dynamic + Compositional					
	GHC	LHC	GHC	LHC	SS	SA	DR	DE	DD	DDE
`bopdp.3`					*1,040,953 states*			*2,747,408 transitions*		
CT	2.7 M	0.091	0.965	**0.078**	0.223	0.300	0.311	0.183	0.256	0.306
IO	39 M	**0.148**	1.008	0.189	0.311	0.243	0.324	0.370	0.323	0.304
`brp.6`					*42,728,113 states*			*89,187,437 transitions*		
CT	88 M	0.281	0.899	0.277	**0.040**	0.083	0.286	0.042	0.170	0.049
IO	5.9 G	0.346	1.057	0.292	0.132	0.130	0.979	0.123	**0.046**	0.082
`collision.4`					*41,465,543 states*			*113,148,818 transitions*		
CT	112 M	0.088	0.969	0.087	0.078	0.030	0.255	**0.011**	0.131	0.056
IO	1.5 G	0.183	1.135	0.235	0.178	0.220	0.395	**0.176**	0.211	0.294

- **Performance is relative to the PART algorithm with a global hash code (Static + GHC).**

Experimental Results (2)

- **Summary across 35 model instances:**

	Static		Dynamic		Dynamic + Compositional					
	GHC	LHC	GHC	LHC	SS	SA	DR	DE	DD	DDE
Average on 35 models										
CT	1.000	0.255	0.962	0.236	0.163	0.206	0.327	**0.152**	0.419	0.179
IO	1.000	0.504	1.050	0.496	0.458	0.423	0.661	0.411	0.531	**0.393**

- **Main observations:**
  1. Compositional dynamic refinement generally outperforms the earlier approaches (GHC and LHC).
  2. DR generally worse than all other heuristics – and always worse than SS and DE which performed comparable.
  3. A general correlation between disk accesses and cross transitions: except in cases with uneven partition distribution.

Partition Overflow

- **Dynamic partitioning can avoid overflow when a some partition cannot be represented in memory.**
- **Ratio of overflowing states* with related approaches [SPIN'99, SPIN'01]:**

	S-LHC	D-LHC
bopdp.3	0.677	0.677
brp.6	0.735	0.735
collision.4	0.722	0.722
firewire_link.5	0	0
firewire_tree.5	0.785	0.785
fischer.6	0.969	0.969
iprotocol.7	0	0

	S-LHC	D-LHC
msmie.4	0.939	0.939
pgm_protocol.8	0	0
plc.4	0	0
rether.7	0.550	0.192
synapse.7	0.090	0.035
telephony.7	0.827	0.827
train-gate.7	0.950	0.950

***Partition size limit is 1% of the total state space.**

Conclusions and Future Work

- **A dynamic partitioning scheme applicable for external memory and distributed model checking.**
- **The heuristics have been evaluated in the context of external memory model checking.**
- **Improves cross transitions and disk access performance compared to earlier related work.**
- **The scheme can ensure an upper bound on size of any partition loaded into memory.**
- **Heuristics are still be investigated in the context of distributed model checking.**