Part 3: Advanced State Space Methods

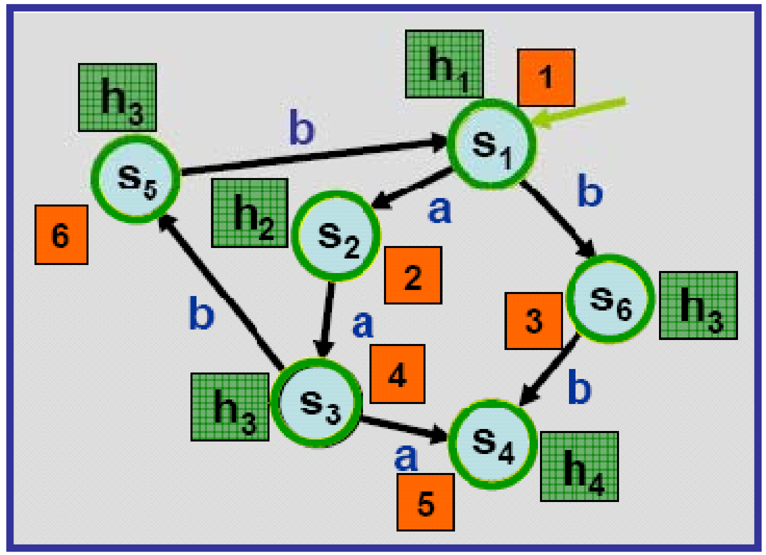
Overview

- **Will present three examples of advanced state space methods for alleviating state explosion.**
- **The comback method:**
  - Relies on hash-compaction for compact storage of states.
  - State reconstruction to ensure full state space coverage.
- **The sweep-line method:**
  - Exploits progress to delete states from memory during state space exploration.
- **State space partitioning:**
  - Divides the state space into partitions.
  - Applied in distributed and external memory model checking.

The ComBack Method - Extending Hash Compaction with Backtracking

The ComBack Method – Extending Hash Compaction with Backtracking.
M. Westergaard, L.M. Kristensen, G.S. Brodal, and L. Arge. Proceedings of Petri Nets 2010, LNCS 4546, pp. 445-464, Springer, 2007.

The ComBack Method Revisited – Caching Strategie and Extensions with Delayed Duplicate Detection.
S. Evangelista, *M. Westergaard, L.M. Kristensen. Transactions on Petri Nets and Other Models of Concurrency*, LNCS 5800 pp. 189-215, Springer, 2009.



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 ➔ 01100011000110001110000011100010 1

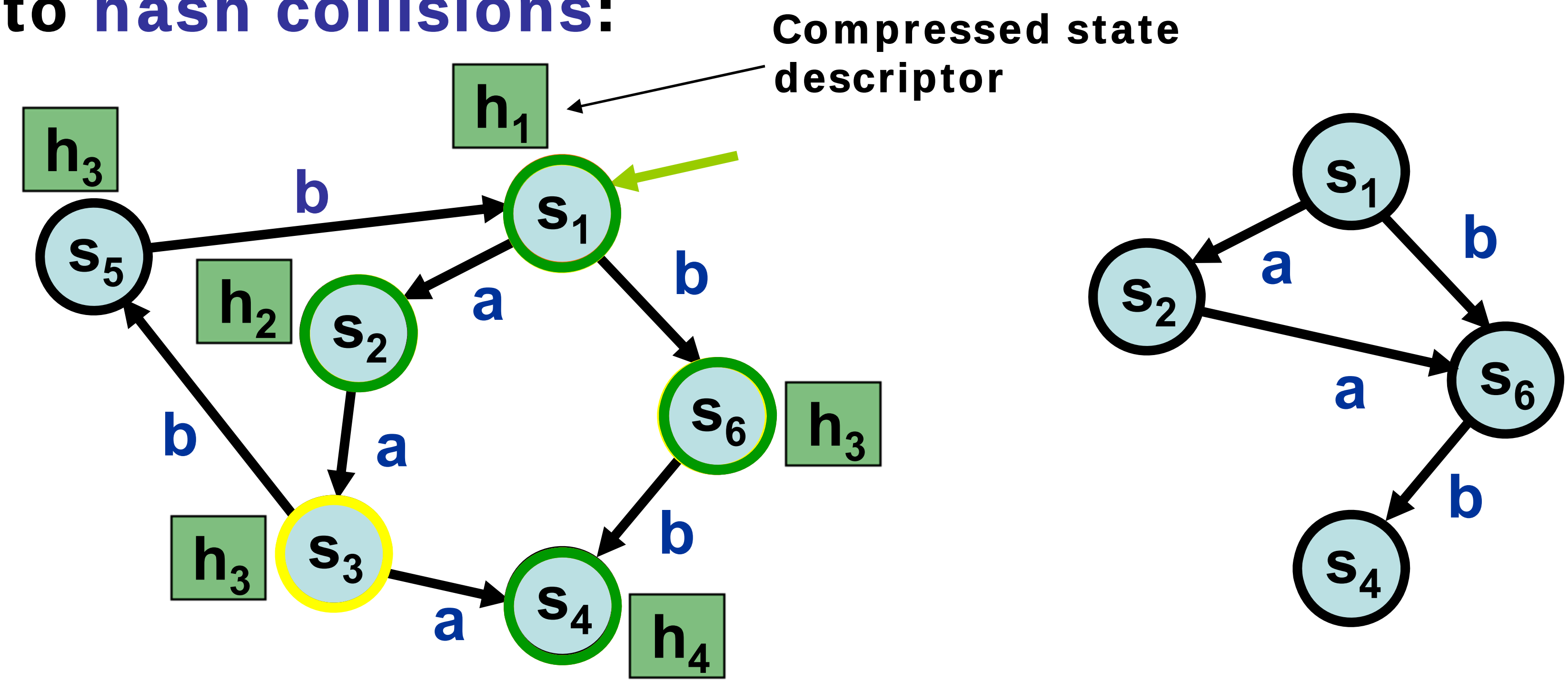
Full state descriptor (100-1000 bytes)

Compressed state descriptor (4-8 bytes)

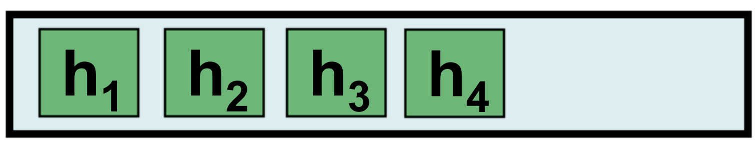
- **Only the compressed state descriptor is stored in the state table of visited states.**

Example: Hash Compaction

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



State table:



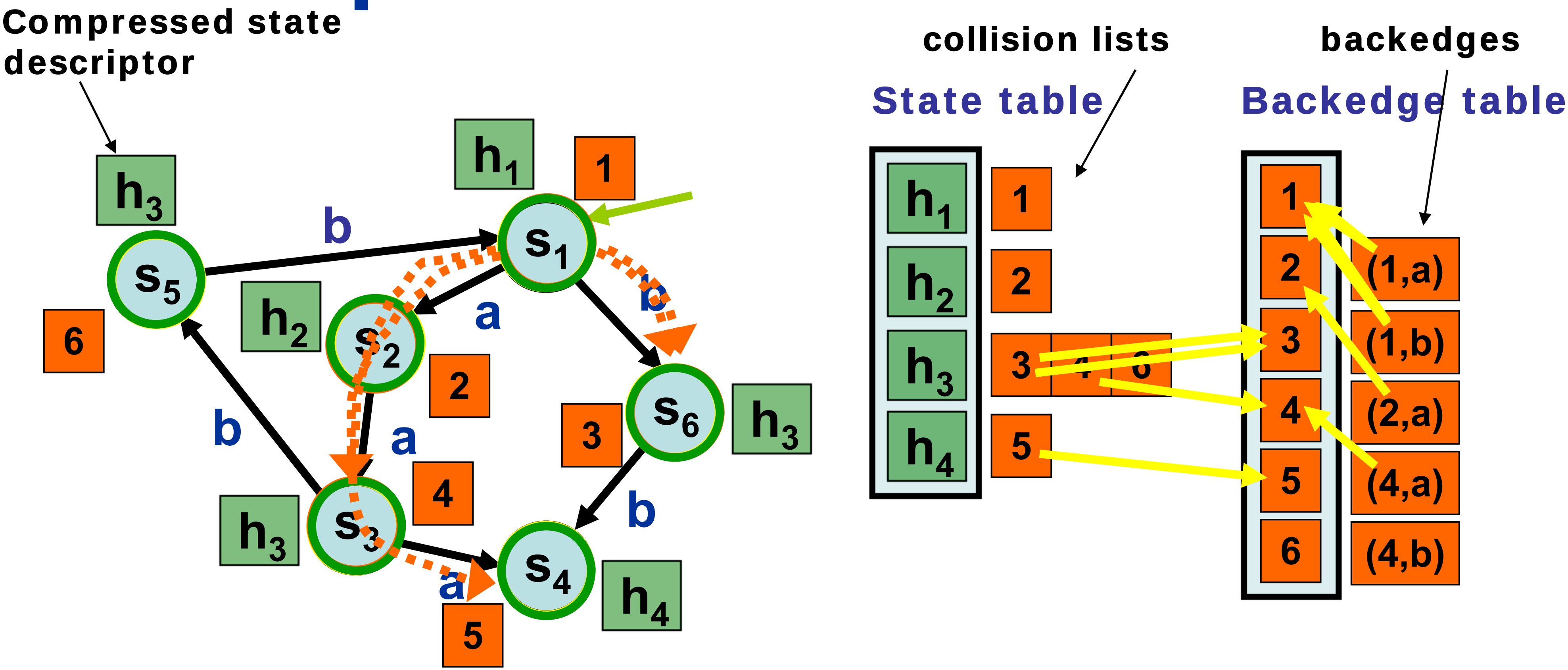
The Comback Method

- 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

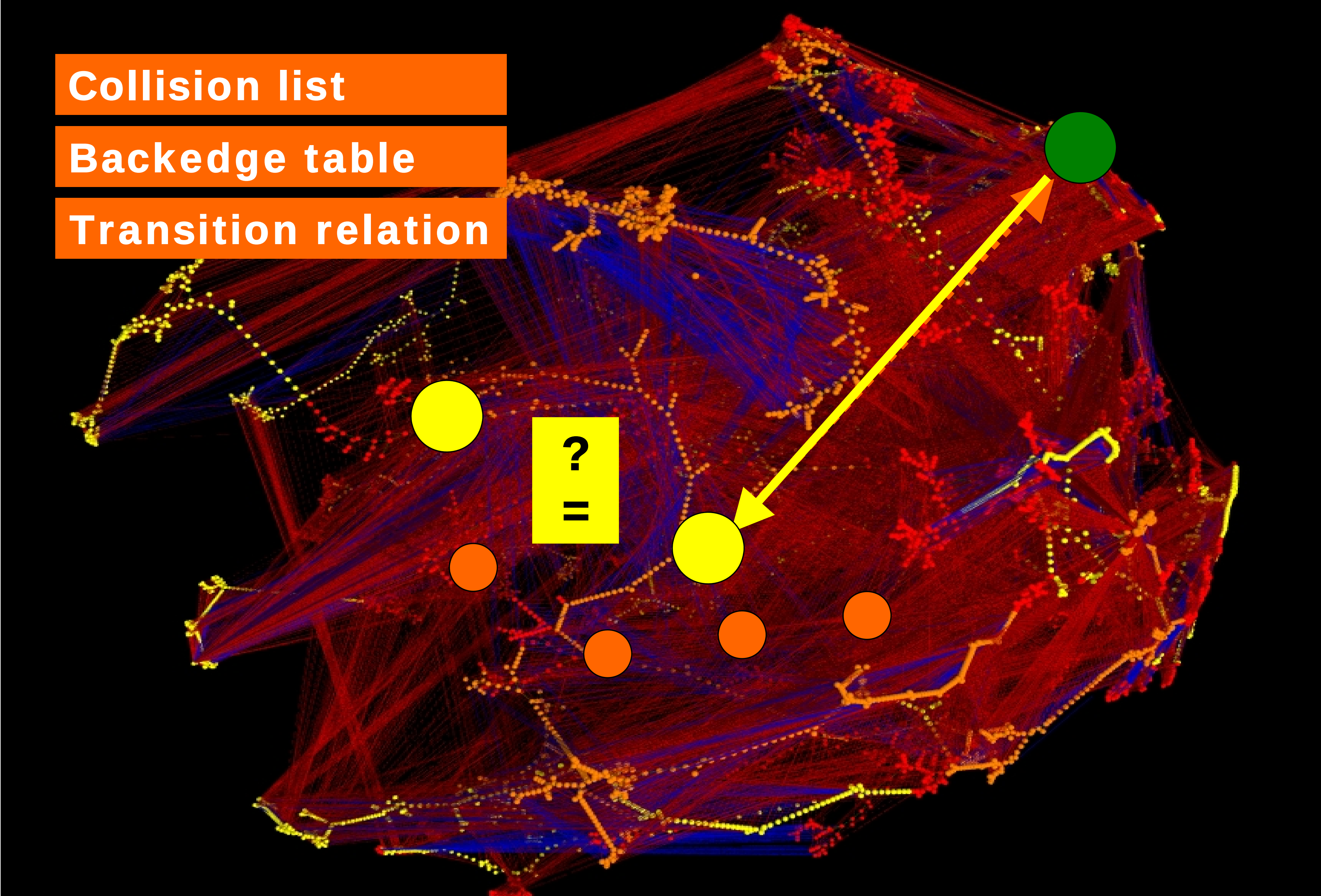
Example: The ComBack Method



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) ➔ $S_4 = S_4$



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:

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

Overhead compared to hash compaction

- Number of state reconstructions bounded by:

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

Implementation

- **Implementation in 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).

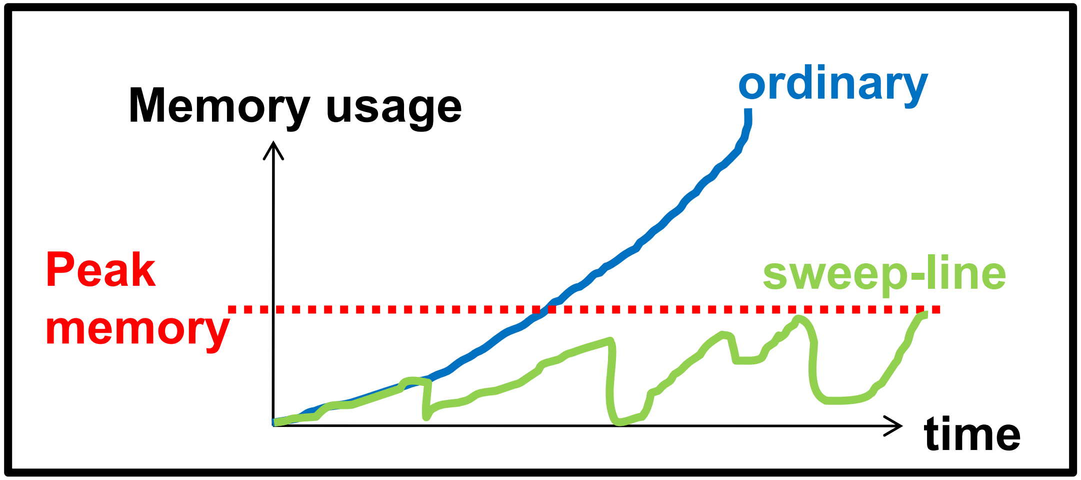
Summary of Experimental Results

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.

The Sweep-Line Method

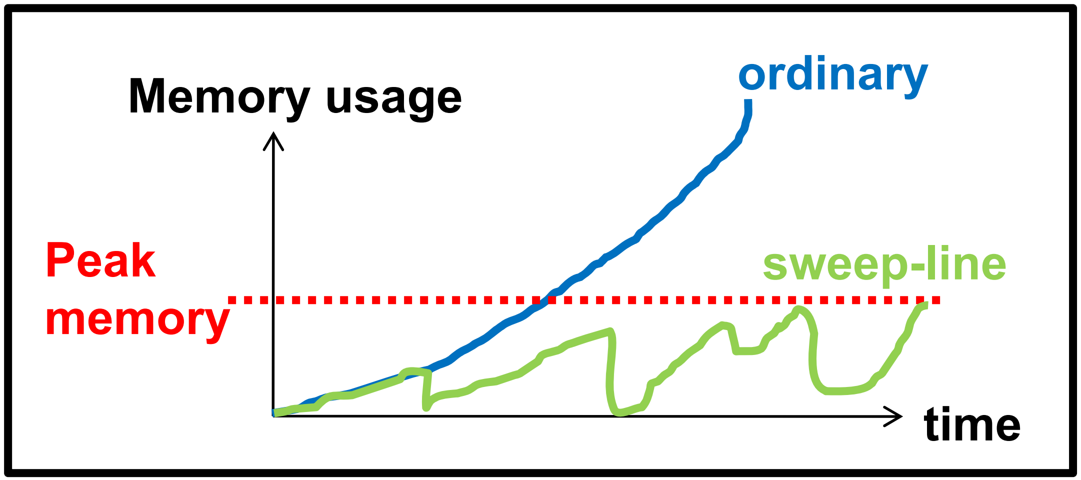


A Sweep-Line Method for State Space Exploration
S. Christensen, L.M. Kristensen and T. Mailund
Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001), LNCS 2031 pp. 450-464.Springer, 2000.

A Generalised Sweep-Line Method for Safety Properties
L.M. Kristensen and T. Mailund
Proceedings of Formal Methods Europe (FME 2002), LNCS 2391 pp. 549-567, Springer, 2002.

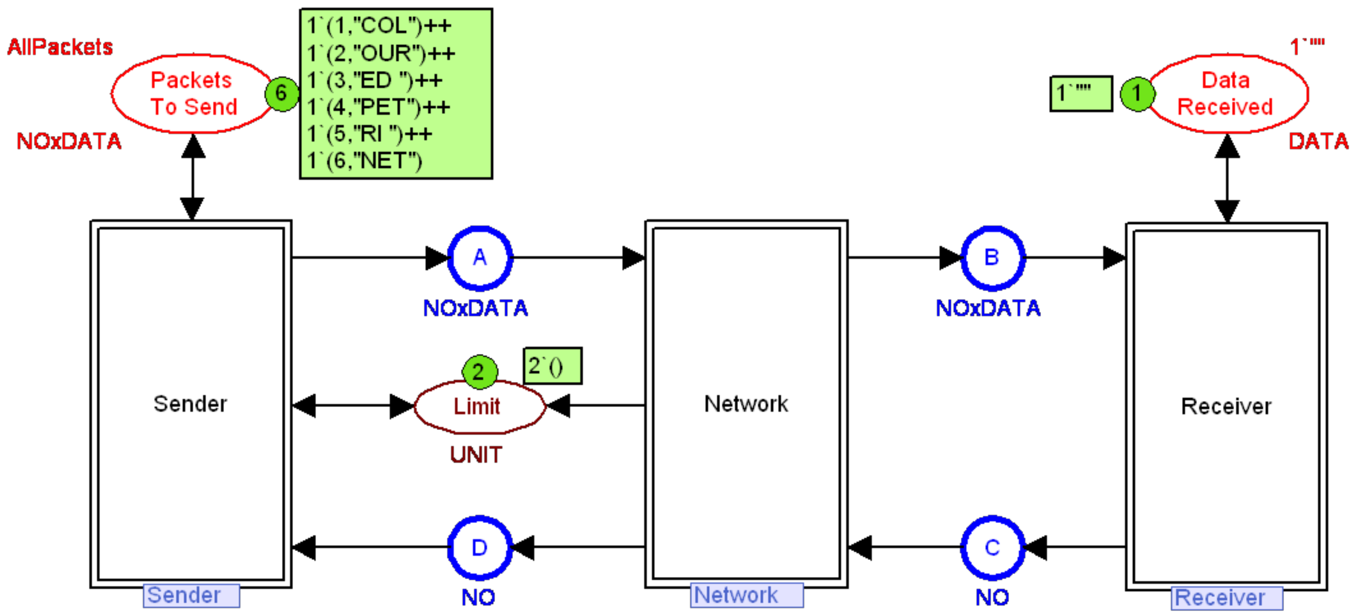
The Sweep-line Method

- The basic idea is to exploit a certain kind of progress exhibited by many systems:
  - Retransmission counters and sequence numbers in protocols.
  - Phases in transaction protocol.
  - Control flow in programs.
  - Time in timed CPN models (value of global clock).
- Makes it possible to explore all the reachable states, while only storing small state space fragments in memory:
- This means that the peak memory usage is reduced.
- Aimed at on-the-fly verification of safety properties.



Example CPN Model

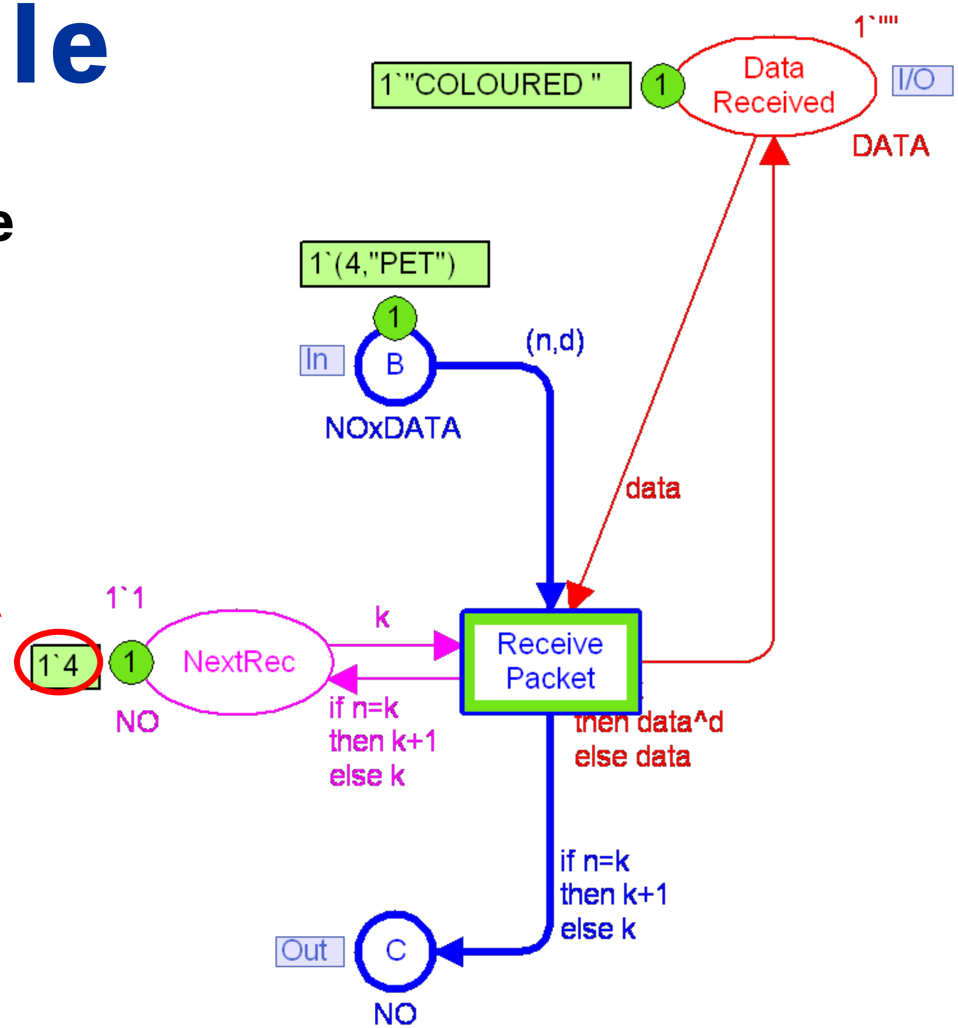
- Stop-and-wait communication protocol:



- Sender keep sending the same **data packet** until a matching **acknowledgement** is received.
- **Sequence numbers** are used to match data packets and acknowledgements.

Protocol Example

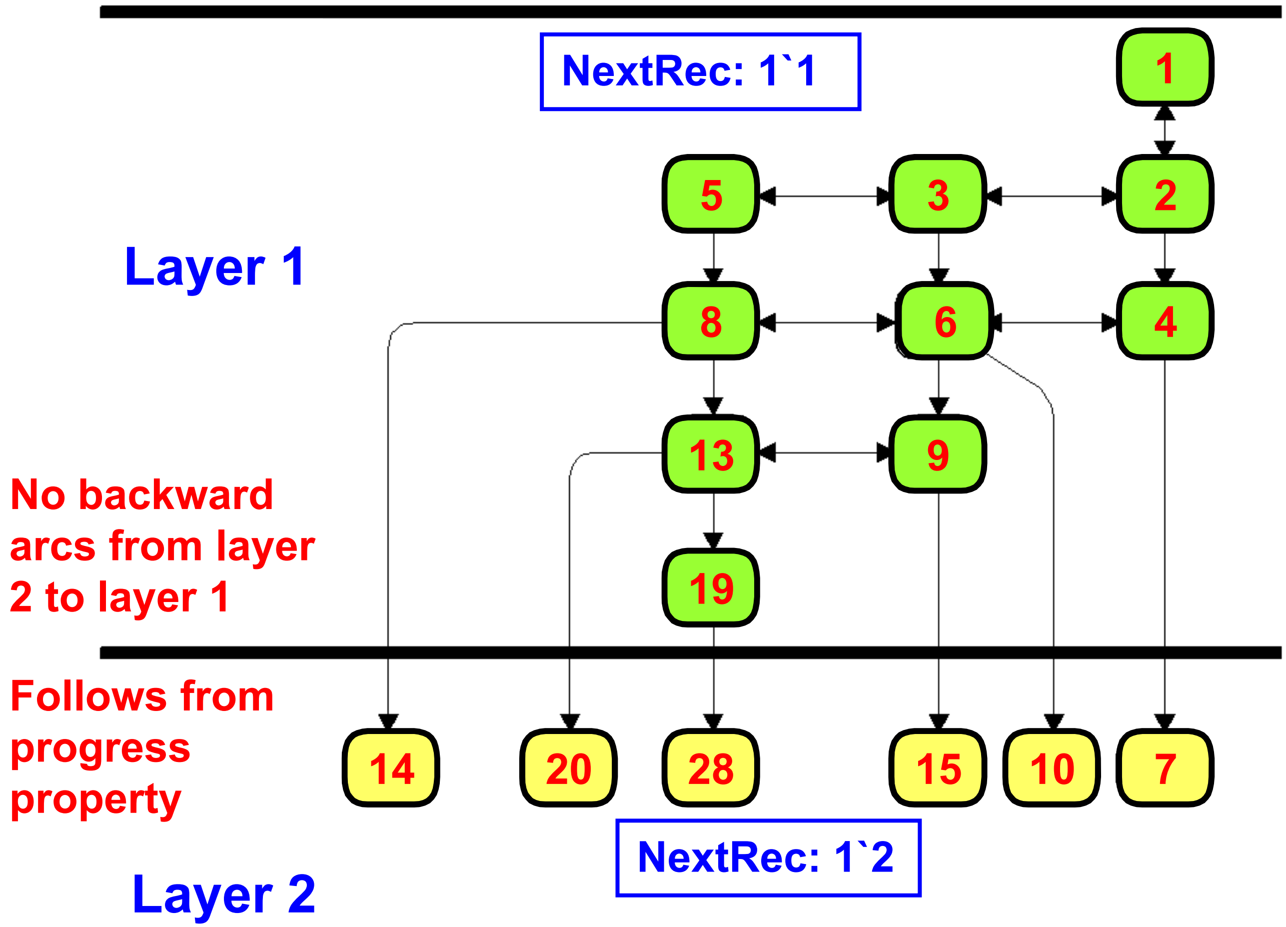
- Measures the progress of the transmission.



- The token value on NextRec increases during execution.
- It is never decreased.

Initial fragment of state space

- Each state has successor states either in the same layer or in higher layers – never in lower layers.
- Layer 1 states can be deleted from memory when they have been processed.



Process states layer by layer

- Process the states (i.e., calculate successor states) one layer at a time (least progress first-order).
- Move from one layer to the next when all states in the first layer have been processed.
- Delete states when moving from one layer to the next.
- A conceptual sweep-line moves through the state space layer by layer:
  - All states in the layer are "on" the sweep-line.
  - All new states calculated are either on the sweep-line or in front of the sweep-line (i.e., in a higher layer).

Progress Measure

- **The progress can be captured by a progress measure mapping each state into a progress value.**

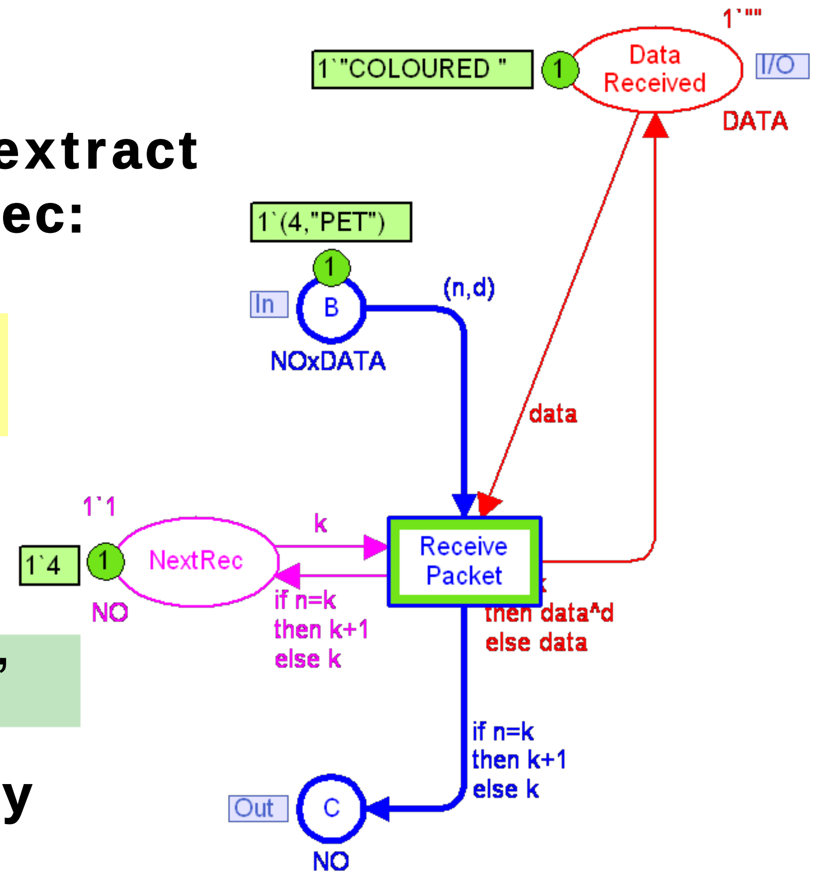
- **Implemented as a function that extract the colour of the token on NextRec:**

```
fun Progress'Receiver { NextRec } =
    List.hd NextRec
```

- **This is an example of a monotonic progress measure:**

$$s \rightarrow s' \Rightarrow \text{progress } s \leq \text{progress } s'$$

- **Monotonicity can be checked fully automatically.**



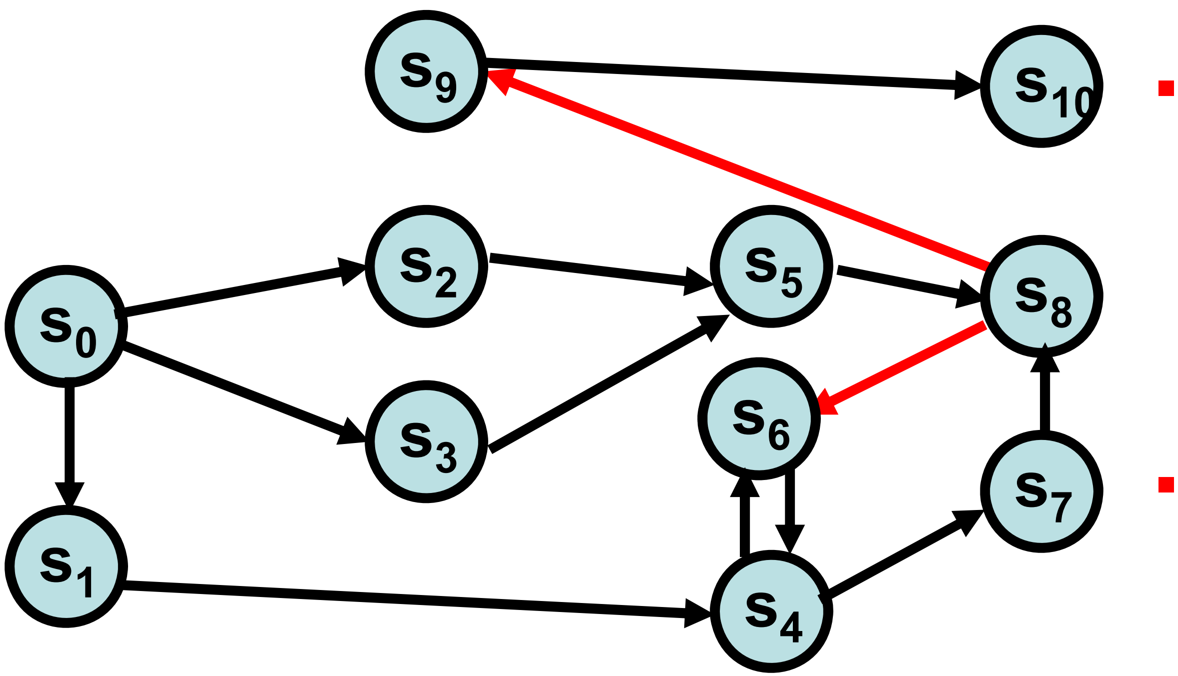
Sample Experimental Results

Limit	Packets	Nodes	Arcs	Nodes (peak)	Nodes	Time
1	4	33	44	33	1.00	1.00
2	4	293	764	134	2.19	1.00
3	4	1,829	6,860	758	2.41	1.00
4	4	9,025	43,124	4,449	2.03	1.78
5	4	37,477	213,902	20,826	1.80	1.65
6	4	136,107	891,830	82,586	1.65	1.51
4	5	20,016	99,355	8,521	2.35	1.95
4	6	38,885	198,150	14,545	2.67	2.19
4	7	68,720	356,965	22,905	3.00	2.27
4	8	113,121	596,264	33,985	3.33	2.41

Configuration | Standard method | Sweep-line | Gain

Generalised Sweep-Line Method

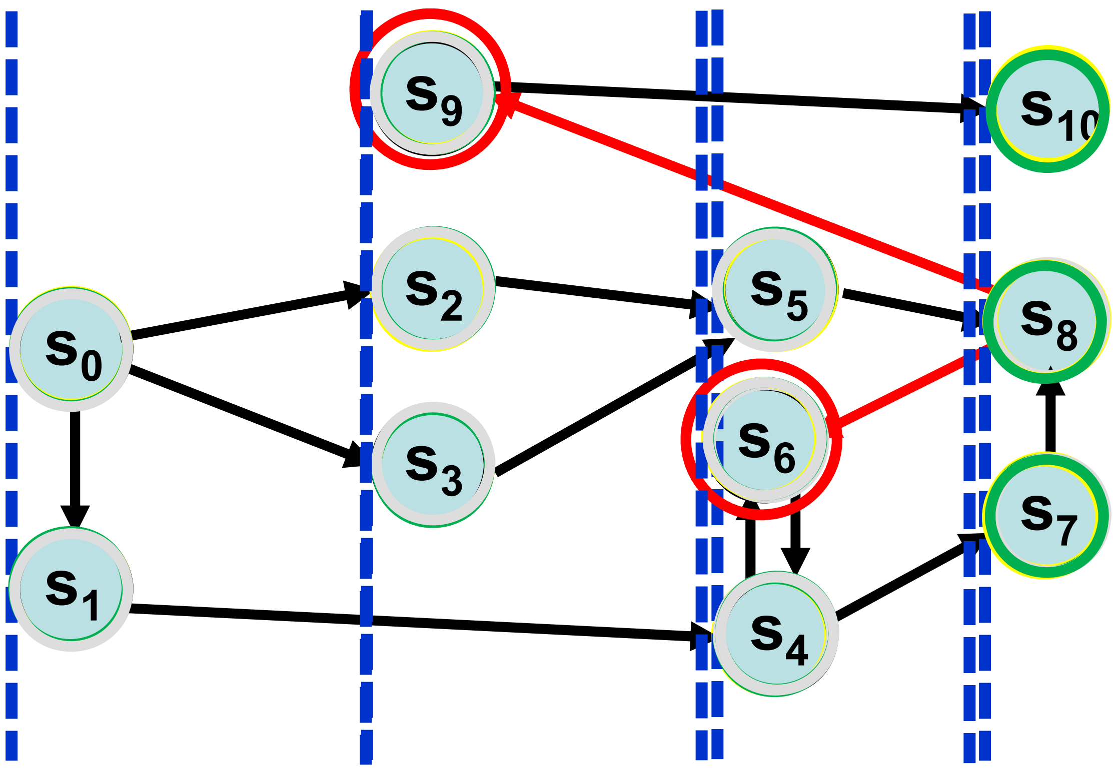
- **Monotonic progress measures are sufficient for systems exhibiting global progress.**
- **Many systems exhibit local progress and occasional regress** (e.g., sequence number wrap, control flow loops,…)**:**



- **Cannot determine whether a destination state of a regress has already been explored.**
- **Termination is no longer guaranteed.**

Generalised Sweep-Line Method

- Detect backwards/regress edges during exploration.
- Mark destination of regress edges persistent.
- Conduct multiple sweeps using persistent states as roots.



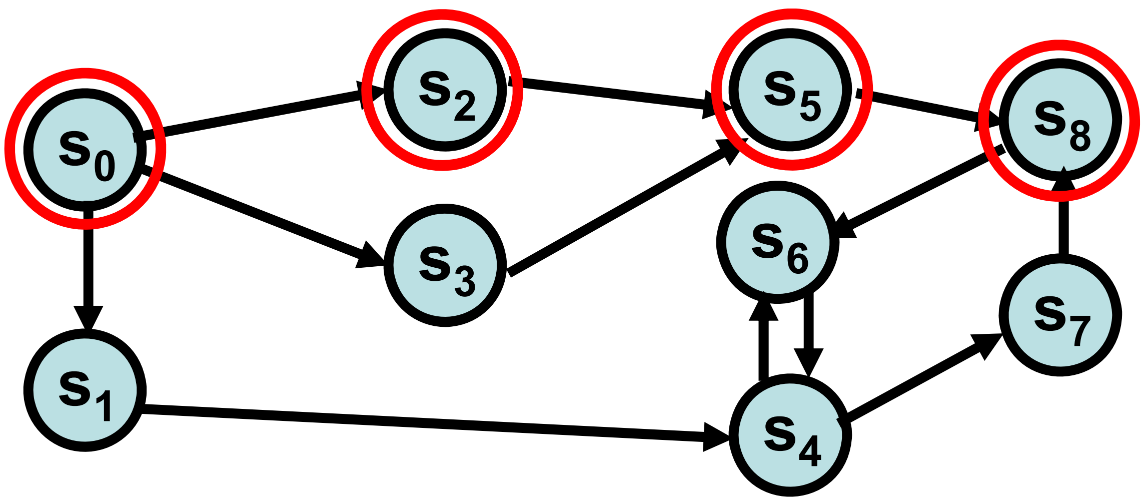
Algorithm and Implementation

```
ROOTS ← {s_I}
NODES.ADD(s_I)
while ¬ (ROOTS.EMPTY()) do
  UNPROCESSED ← ROOTS
  ROOTS ← ∅
  while ¬ (UNPROCESSED.EMPTY()) do
    s ← UNPROCESSED.GETMINELEMENT()
    for all (t, s') such that s -t-> s' do
      if ¬(NODES.CONTAINS(s')) then
        NODES.ADD(s')
        if ψ(s) ⊐ ψ(s') then
          NODES.MARKPERSISTENT(s')
          ROOTS.ADD(s')
        else
          UNPROCESSED.ADD(s')
        end if
      end if
    end for
    NODES.GARBAGECOLLECT(min{ψ(s) | s ∈ UNPROCESSED})
  end while
end while
```

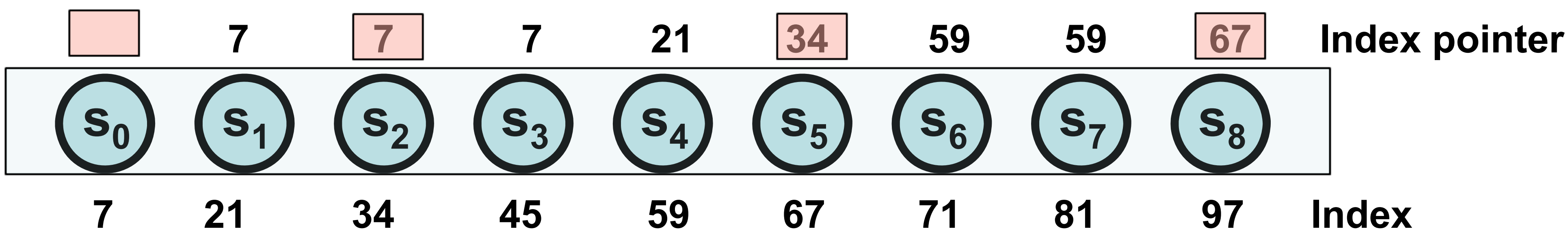
- **Unprocessed implemented as a priority queue on progress values.**
- **Deletion of states can be implemented efficiently by detecting when the sweep-line moves.**
- **Sharing of substate requires a reference count mechanism.**

Counter Example Generation

- **External storage can be exploited to support counter example generation:**

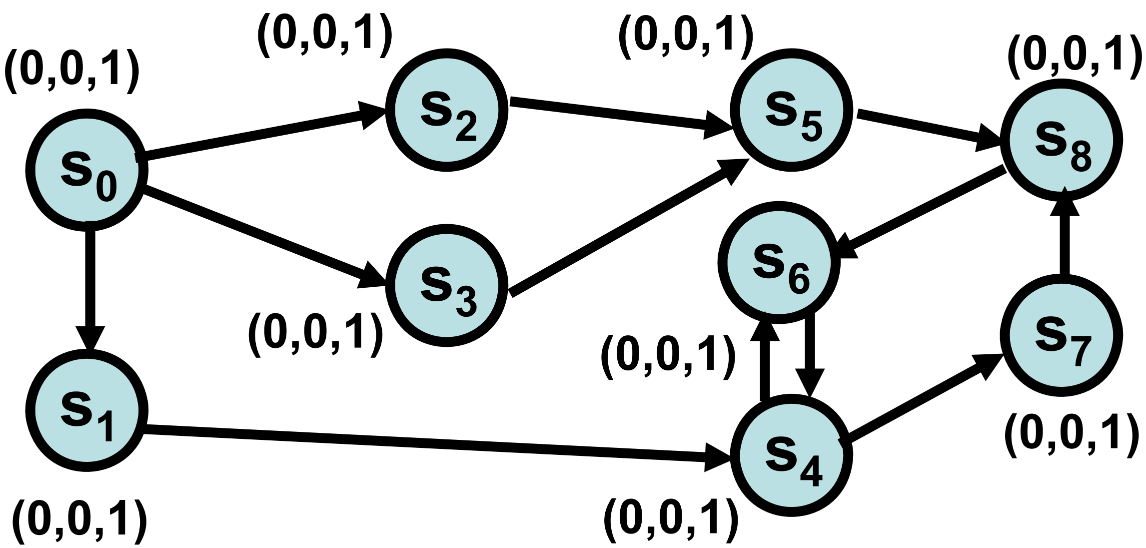


- Store the states being deleted from memory sequentially on disk.
- Store an index pointing to the generating predecessor of each state.
- Following index pointers backwards yields the counter example.



Beyond Safety Properties

- **Classical CTL and LTL model checking algorithms are not compatible:**
  - Accesses predecessor states (e.g., CTL).
  - Relies on a non-progress first search order (e.g., LTL).
- **The sweep through the state space can be used to compute a Kripke structure:**



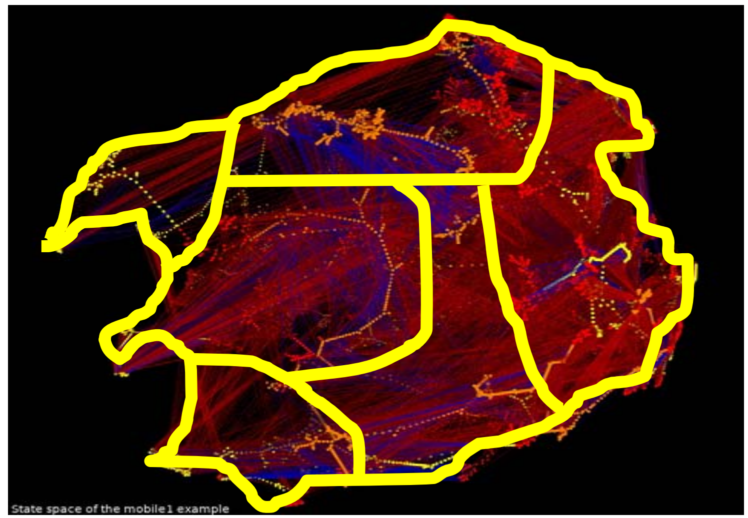
Atomic propositions:

p_1 p_2 p_3

- States are not deleted but instead replaced by a (small) bit vector.
- Standard model checking algorithm can be used.

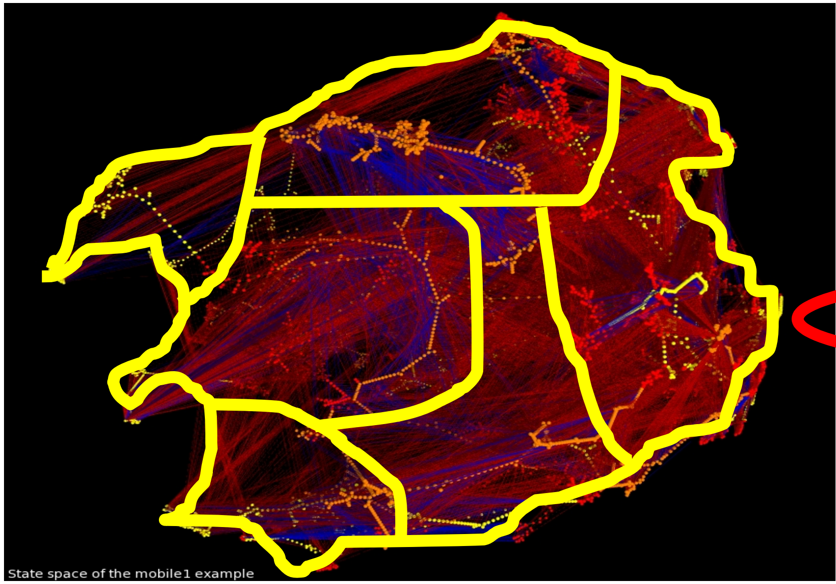
Dynamic State Space Partitioning for External Memory Model Checking

Dynamic State Space Partitioning for External Memory Model Checking. S. Evangelista and L.M. Kristensen.. In Proc. Formal Methods for Industrial Critical Systems, LNCS 5825, pp. 70-85. Springer, 2009.



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 partitioning function mapping from the set of states into partitions.**

External-Memory Algorithm

- **Uses a queue Q_i of unprocessed states, a set of visited states V_i, and a file F_i for each partition i:**

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**
    $i := longestQueue()$
    $\mathcal{F}_i.load(\mathcal{V}_i)$
    $search_i()$
    $\mathcal{V}_i.unload(\mathcal{F}_i)$

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.

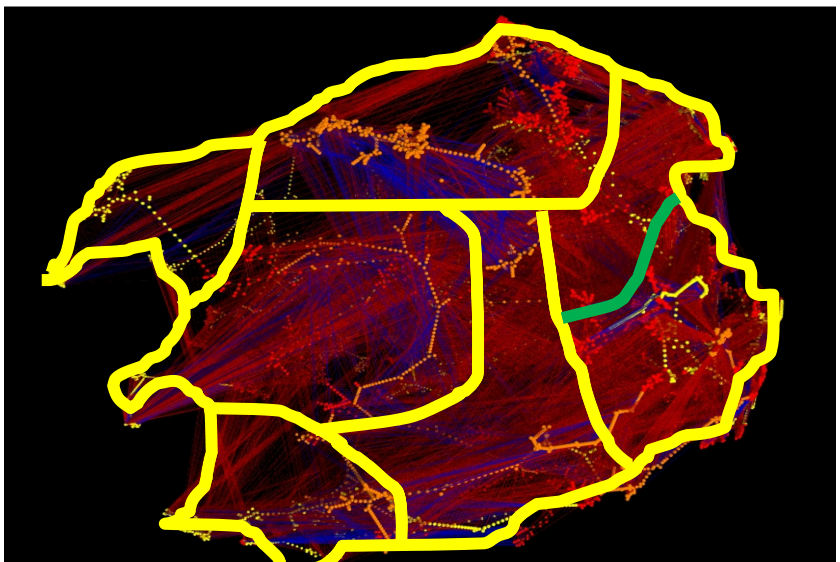
- **Main contributions of this work:**
  1. A dynamic partitioning scheme based on partition refinement and compositional partitioning functions.
  2. A set static and dynamic heuristics for implementing partition refinement in the context of external memory model checking.
  3. An implementation and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



State space of the mobile1 example

- 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

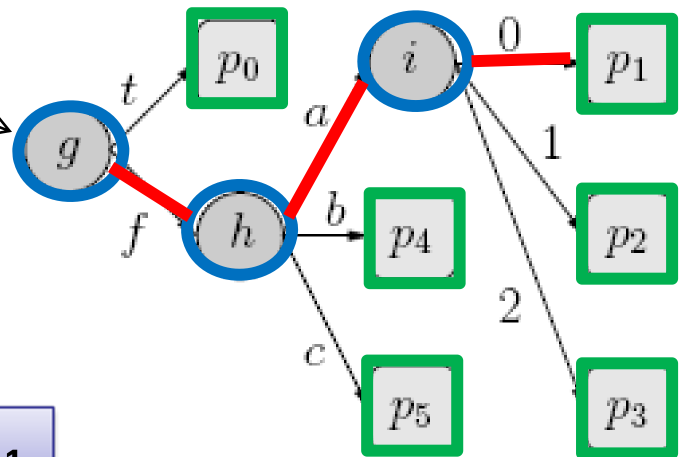
- A compositional partitioning function can be represented as a **partitioning diagram**:

Single root node

State s

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

p_1



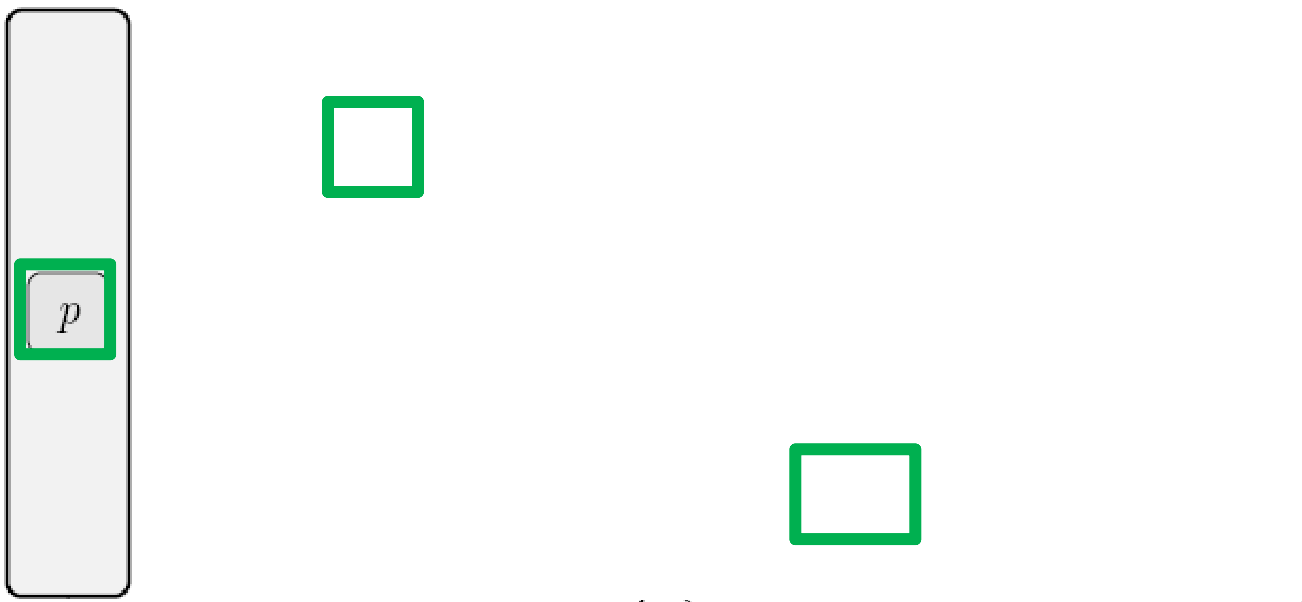
Branching nodes (branching functions)

Terminal nodes (state partitions)

- 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 a state component to be used for the refinement.**

- **Offline Static Analysis (SA):**
  - Count for each state component, the number of events in the analysed system model that modifies it.
  - Among candidate components, select the component with the lowest count (to reduce cross transitions).
- **Offline Dynamic State Space 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).

Online 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):**
  - Counts 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** **[www.daimi.au.dk/~ascoveco/download.html]:**
  - The PART external memory algorithm [Bao, Jones (TACAS'05)]: uses a global hash function on the state vector for partitioning.
  - A static partitioning scheme [Lerda, Sisto (SPIN'99)]: The partitions are determined from a single state component.
  - A semi-dynamic partitioning scheme [Lerda, Visser (SPIN'01)]: partitions consists of static classes that can be reassigned.
- **Experiments conducted on models from the BEEM benchmark database** **[Pelánek (SPIN'07)].**
- **Illustrates the use of ASAP as a multi-formalism platform.**

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 when partition distribution is uneven.

Partition Overflow

- **Dynamic partitioning can avoid overflow when 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

***A partition size limit of 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 to be explored in the context of distributed model checking.