Organisation of the State Space

System Validation #5
Monday, 19 May 2008

SV #5 http://fmt.cs.utwente.nl/courses/systemvalidation/

Theo Ruys
kamer: INF 5037
telefoon: 3716
email: ruys@cs.utwente.nl

University of Twente
Department of Computer Science
Formal Methods & Tools

Lectures

Note that the original lectures 5 and 6 have been combined.

Also note that there will no presentation on dynamic POR.

#	date		Topic	Material
1	Mon	14 April	SPIN	[Gerth 1997] [SPIN QuickRef] [Hatcliff 2001]
	Wed	16 April	no lecture!	
2	Mon	21 April	Linear Temporal Logic	[Merz 2000]
	Wed	23 April	no lecture!	
3	Wed	7 May	Building a Model Checker	[Kattenbelt et.al. 2007]
4	Wed	14 May	Partial Order Reduction	[Peled 1999] ~~[Flanagan & Godefroid 2005]~~
5	Mon	19 May	Hashing & Compression	[Kuntz & Lampka 2004] [Holzmann 1997]
6	Mon	26 May	Software Verification	[Visser et. al. 2003] [Ruys & Aan de Brugh 2007] [Ball & Rajamani 2001]
	Mon	2 June	no lecture!	
	Mon	9 June	no lecture!	

Important notes

- The lecture on Monday, 9 June 2008, has been cancelled.
- The paper [Flanagan & Godefroid 2005] is not longer part of the required reading material for the examination.

SUMO project

- Description of SUMO project is on-line (finally).
- Deadline of the SUMO project has been postponed to
  - Thursday, 12 June 2008 23.59h (strict deadline)
    (was: Monday, 9 June 2008)
- Website of System Validation hosts some additional resources:
  - ANTLR/Java source code of a front end for a SUMO compiler (i.e., scanner, parser and context analyser).
  - several test- and benchmark SUMO models, which will be used to check the SUMO explorers for correctness and performance.

Overview of lecture 5

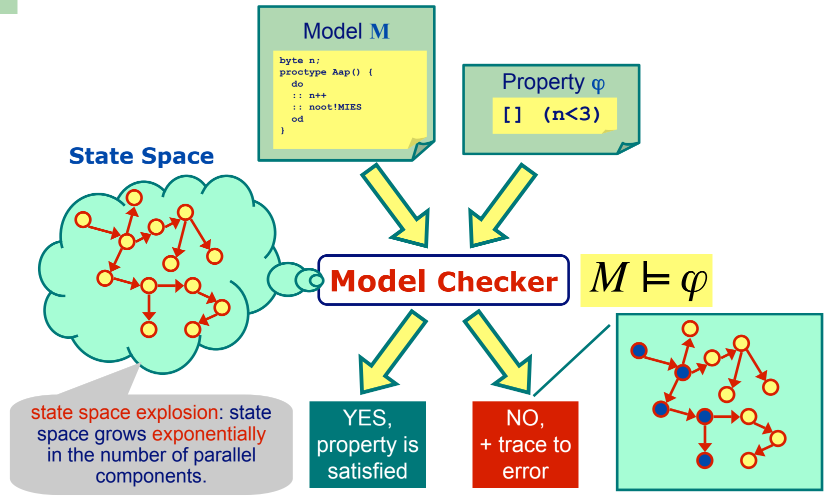
Emphasis on intuition

- Introduction
- State Compression
  [Holzmann - SPIN 1997]
- Bitstate Hashing
  [Kuntz & Lampka - 2004]
- Minimised Automaton
  [Holzmann & Puri - STTT 1999]

Not part of the material for the examination.

Ideas of lecture 5 might prove useful for the SUMO project.

Model Checking



Pragmatic recipe to verify $M \models \varphi$

using SPIN

1. Sanity check
   Interactive and random simulations.

2. Partial check
   Use SPIN's bitstate hashing mode to quickly sweep over the state space.

3. Exhaustive check
   If this fails, SPIN supports several options to proceed:
   1. Compression (of state vector)
   2. Optimisations (SPIN-options or manually, e.g. -DMA)
   3. Abstractions (manually, guided by SPIN's slicing algorithm)
   4. Hash-compaction (approximate)
   5. Bitstate hashing (approximate)

Verification Algorithm

- In this lecture we use a depth first search algorithm (DFS) to generate and explore the state space.

```
dfs(s0)

procedure dfs(s: state)
    store s in state space
    foreach successor s' of s do
        if s' not in state space
        then dfs(s')
    od
end dfs
```

SV #4: PO reduction tries to visit a subset of the successors only without sacrificing the full coverage of the state space.

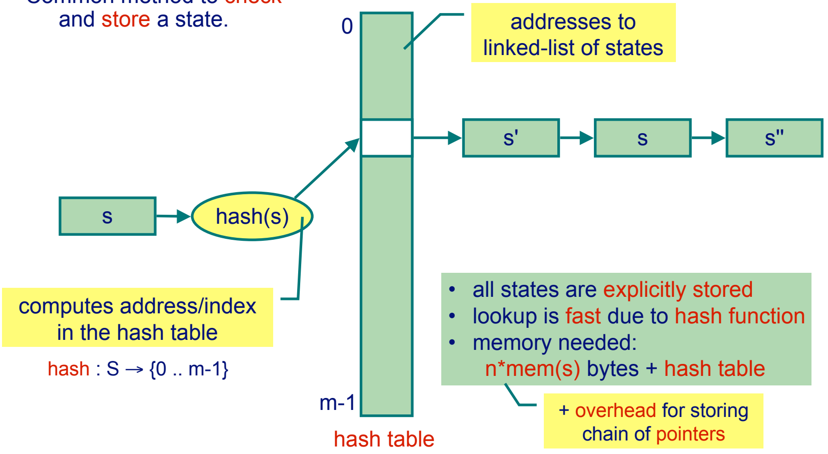
The old states s are kept on the dfs-stack, which corresponds with a complete execution path.

- Topics of this lecture:
  - organisation of the state space
  - representation of the states

Note: states do not have to be stored; DFS stack is enough to guarantee completeness.

Storing States

Common method to check and store a state.



- all states are explicitly stored
- lookup is fast due to hash function
- memory needed:
  n*mem(s) bytes + hash table

Overview

Gerard J. Holzmann.
State Compression in SPIN: Recursive Indexing and Compression Training Runs.
Proceedings of the Third SPIN Workshop, SPIN 1997, Enschede.

Matthias Kuntz and Kai Lampka.
Probabilistic Methods in State Space Analysis.
In: C. Baier et. al. (Eds): Validation of Stochastic Systems.
LNCS 2925, pp. 339-383. 2004.

Gerard J. Holzmann and Anuj Puri.
A Minimized Automaton Representation of Reachable States.
STTT, Vol. 2, No. 3 (1999), pp. 270-278.

State vector

- A state vector is the information to uniquely identify a system state s. For example, for SPIN it contains:
  - global variables
  - contents of the channels
  - for each process in the system:
    - local variables
    - process counter of the process
- It is important to minimise
  - the number of states, and
  - the size of the state vector

For example:
k = 100 bytes
n = 10^6 states
state space: 10^8 = 100 Mb

state vector = k bytes
state space = n states
→ storing the state space at least requires n*k bytes

State Compression

- Loss-less compression
  Given n reachable states, each consuming S bits of memory:
  - no more than log(n) bits of information is needed to be stored per state
    - if state vectors overlap in memory, the amount of memory is even less than log(n)
    - the system can be stored in n·log(n) bits
  - state vector is S bits long
  - compression of S should be efficient
  - decompression is never needed:
    - state matching always on the compressed form
- Bit-state hashing and hash compaction (2nd part of this lecture) are lossy compression techniques.

Static Compression (1)

- byte masking

Simple static compression technique: SPIN default compression mode.

  - This method predefines a compression by identifying byte values in the state-vector that are known to contain constant values. For example:
    - the number of active processes and channels,
    - process instantiation numbers,
    - unused elements of the state-vector,
    - the channel contents for rendez-vous channels,
    - all byte fields that are added by the compiler to secure proper alignment of variables and data.
  - Compression is reversible and lossless.
  - State vector is compressed to 85%, on average.

Static Compression (2)

- run-length encoding
  - compressed state vector is sequence of byte pairs (n,b), where n denotes the number of consecutive bytes b

For example:

00 00 03 03 03 A1 A1 62 62 62 5A 72 72 89 89 89 89 12 12 12 93 21 21 21 21

02 00 03 03 02 A1 03 62 01 5A 02 72 04 89 03 12 01 93 04 21

  - only helps if on average each byte value appears in more than two consecutive positions in the state vector

Static Compression (3)

Also used in compression programs for .zip .gz .jpeg .mpeg .mp3

- static Huffman compression
  - first find the relative frequency of byte values in state vectors (using experiments)
  - predefine a dictionary for the Huffman encoding
  - most frequently occurring values get the shortest bit-codes
  - bytes are then stored with variable length bit-sequences

E.g.

00	12.1%	1	89	2.6%	00010
12	7.2%	001	21	2.1%	00011
FF	5.3%	010	A1	1.9%	0000001
62	3.1%	011	93	1.8%	0000010
2B	2.9%	00001	03	1.7%	0000011

source 00 03 03 12 62 00 00 00 12 93 00 00 89 89 12 12 12 93 21 21 (20 bytes = 160 bits)

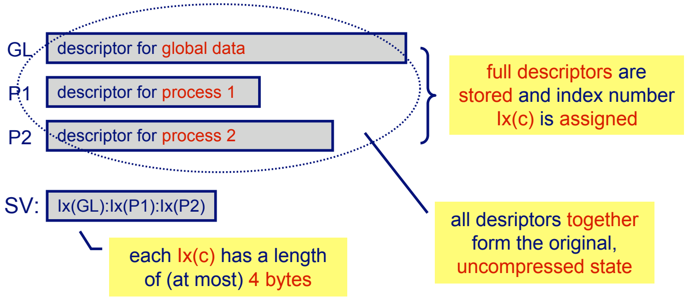
target 1 0000011 0000011 001 011 1 1 1 001 0000010
1 1 00010 00010 001 001 001 0000010 00011 00011 (72 bits)

Collapse Compression (1)

- Observation
  - number of distinct system states grows very fast
  - despite the fact that each process and each data object can typically reach only a small number of distinct states
  - explosion of the number of reachable states is caused by the large number of ways in which local states of individual components can be combined.
  - replicating a complete description of all local components is therefore an inherently wasteful technique
- Idea of collapse compression
  - store small state components separately
  - assign small unique index numbers to each small component
  - global state descriptor is now formed by the combination of the unique index numbers.

Collapse Compression (2)

- Possible organisation of components
  - global component: all global data
  - one component for each process, recording its control state together with the state of all its local variables



Collapse Compression (3)

- Example:

Given the following tables with partial components:

globals

01	00 01 39 89 90 12 30 40
02	12 38 45 24 91 30 29 80
03	21 84 92 23 78 63 54 12
04	31 00 29 56 23 91 80 22
05	49 23 24 89 12 34 26 19
06	79 23 11 91 53 91 23 78

P1

01	00 01 11
02	12 38 45
03	78 63 23

P2

01	31 00 91 80 22
02	49 23 34 26 19
03	79 23 78 26 19

P3

01	30 40
02	12 38
03	54 12
04	31 22
05	49 23

Now the state: $\underbrace{\text{79 23 11 91 53 91 23 78}}_{\text{globals}}$ $\underbrace{\text{12 38 45}}_{\text{P1}}$ $\underbrace{\text{79 23 78 26 19}}_{\text{P2}}$ $\underbrace{\text{12 38}}_{\text{P3}}$

is represented by: 06 02 03 02

Collapse Compression (4)

- Huffman encoding and collapse compression can even be combined:
  - first perform a training run, which does not need to be complete (e.g. using bitstate hashing)
  - gather statistics on the state descriptors, i.e. the frequency of the distinct components
  - use the statistics to build a custom Huffman dictionary for the components

    again, assigning the most frequently occurring values the shortest bit-codes.
  - use the Huffman indices in the final verification run

Collapse Compression (5)

- Collapse compression
  - typically reduces the memory requirements for the state table to 20% of the non-compressed version
  - running time may be multiplied by a factor three
  - SPIN: **`-DCOLLAPSE`**
- Collapse + Huffman encoding
  - slightly better than “standard” collapse compression
  - but notably slower (one third)

Overview

Gerard J. Holzmann.
State Compression in SPIN: Recursive Indexing and Compression Training Runs.
Proceedings of the Third SPIN Workshop, SPIN 1997, Enschede.

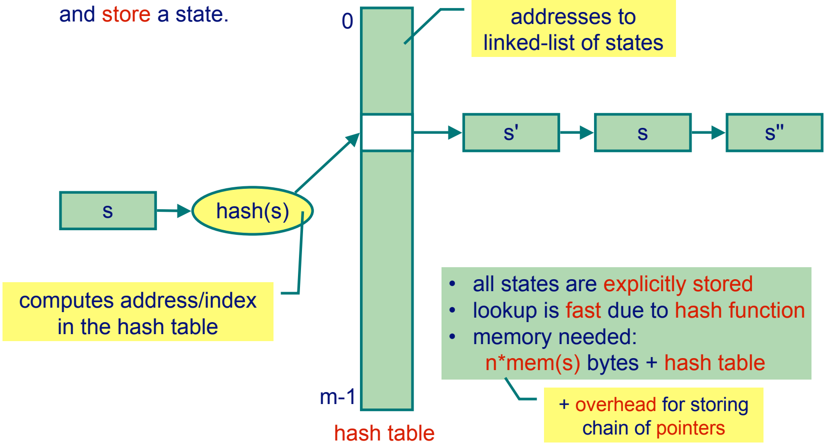
Matthias Kuntz and Kai Lampka.
Probabilistic Methods in State Space Analysis.
In: C. Baier et. al. (Eds): Validation of Stochastic Systems.
LNCS 2925, pp. 339-383. 2004.

Gerard J. Holzmann and Anuj Puri.
A Minimized Automaton Representation of Reachable States.
STTT, Vol. 2, No. 3 (1999), pp. 270-278.

Storing States

revisited

Common method to check and store a state.



- all states are explicitly stored
- lookup is fast due to hash function
- memory needed:
  n*mem(s) bytes + hash table

+ overhead for storing chain of pointers

Hashing

Popular, and good hash functions:
Bob Jenkins: http://burtleburtle.net/bob/hash/doobs.html
Paul Hsieh: http://www.azillionmonkeys.com/qed/hash.html

- Hash function h: S → {0,..., m-1}.
  - parameter k of h(k) is called key
  - value h(k) is called hash address or hash value
- Requirements of hash function:
  - efficiency: computation should be fast
  - distribution: to avoid hash collisions, the hash values should be distributed evenly over the hash table
- Examples:
  - division with rest:
    h(k) = k mod m
  - multiplicative method:
    $h(k) = \lfloor m \cdot (k \cdot \varphi - \lfloor k \cdot \varphi \rfloor) \rfloor$ with $\varphi \in R$

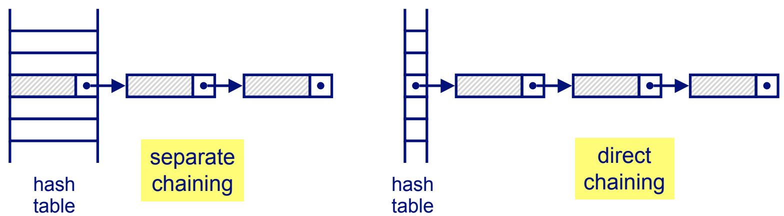
Collision resolution: what to do when h(k)=h(k'), where k≠k'?

Hashing - chaining

Collision resolution: what to do when h(k)=h(k'), where k≠k'?

chaining: (collided) keys are stored <u>outside</u> the hash table

- separate chaining of collided keys: each element of the hash table is the first element of a linked list
- direct chaining of collided keys: each element of the hash table is a pointer to a linked list



Hashing - open addressing

Collision resolution: what to do when $h(k)=h(k')$, where $k \neq k'$?

open addressing: <u>all</u> keys are stored inside the hash table

- if upon insertion of key k' another key is already stored at h(k'), another slot for k' has to be searched for in the hash table.
- probe sequence: sequence of slots that is inspected during the search for an empty slot
- double hashing:
  - second hash function h' (ideally independent from h) defines the probing sequence in case of a collision

    h(k) mod m

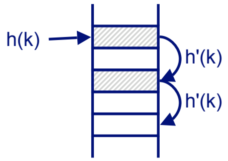
    (h(k) + 1 · h'(k)) mod m

    (h(k) + 2 · h'(k)) mod m

    ...

    (h(k) + (m-1) · h'(k)) mod m
  - probing sequence should form a permutation of the hash addresses

Otherwise, the probing seqence for each synonym is related.



Bitstate Hashing (1)

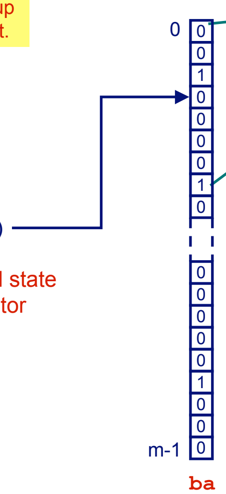
[Holzmann 1987]

- Suppose: n << m
  - if hash function is appropriate: no hash collisions
  - storing the full state descriptors is not needed
- Bitstate hashing: not storing the full state descriptors of visited states, but only storing a bit per state.
  - ensure: n << m
  - no collision resolution!
  - different state descriptors may be mapped upon the same bit address: successors will not be explored

    not longer exhaustive, chance of false positives
  - p_{om} = chance of error = probability of omission of states
  - p_{cov} = $1-p_{om}$ = probability of visiting all states
  - p_{no} = chance of no state-collision

Bitstate Hashing (2)

Note: lookup is very fast.



h(s)

s is the full state descriptor

The hash table is a bit array which only holds bits: 0 or 1.

If h(s)=n and ba[n]=1, we have to conclude that s has already been visited.

Example
m = 2^{30} = 1 gigabit (1 billion bits)
size(hash table) = 128Mb

Bitstate hashing can only be recommended if the size of the state space and the memory requirements render an exhaustive search infeasible.

Bitstate Hashing (3)

- Assumptions: (are not always realistic)
  - one hash collision leads in average to the omission of only one successor state
  - reachability graph is well connected

$$p_{no} = (1-\frac{0}{m}) \times \ldots (1-\frac{i-1}{m}) \times (1-\frac{n-1}{m})$$

p_{no} = chance of no state collision

$$= \prod_{i=0}^{n-1} 1-\frac{i}{m}$$

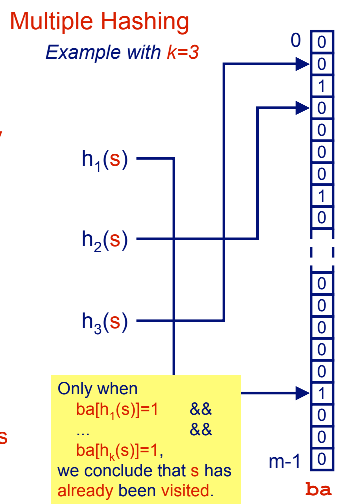
$$\approx \prod_{i=0}^{n-1} e^{-\frac{i}{m}} = e^{-\sum_{i=0}^{n-1}\frac{i}{m}} = e^{-\frac{n(n-1)}{2m}} \approx e^{-\frac{n^2}{2m}}$$

To achieve a high probability of no collision: $2m >> n^2$

Major disadvantage of single bitstate hashing, is the waste of memory, when one wants to achieve a p_{no} very close to 1.

Bitstate Hashing (4)

- Multiple Hashing
  - single bit array
  - k independent hash functions
  - a state s is considered to be already visited, iff all k bit positions in the bitstate are occupied
- [Wolper & Leroy 1993]
  - k large
    - bitstate array will fill up too quickly
    - time consuming
  - k small
    - huge bitstate array is needed in order to achieve high coverage rates



Bitstate Hashing (5)

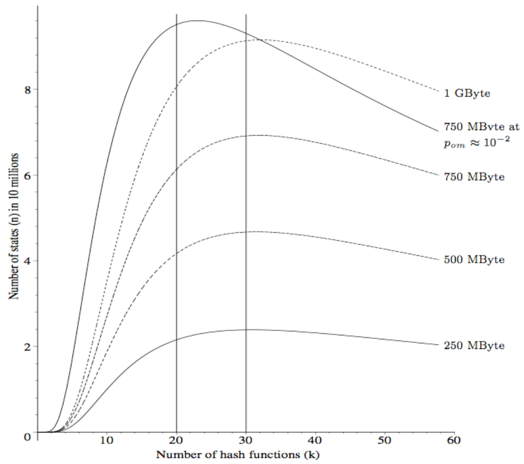


Fig. 5. Fixed SSp coverage ($p_{om} = 10^{-6}$) and varying size of memory

- Fixed $p_{om} = 10^{-6}$
- Varying size of memory.
- When k is increasing, the bitstate array will fill up too quickly.
- Note that k=20 is only appropriate when $p_{om} < 10^{-6}$.
- For current hardware configurations, k=30 seems more appropriate.

Bitstate Hashing (6)

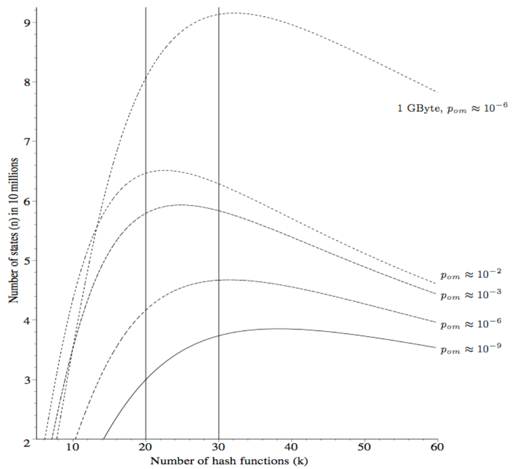


Fig. 6. Memory Space of 500 MByte and varying levels of SSp coverage

- Fixed memory: 500Mb.
- Varying levels of coverage, i.e. p_{om}.
- Note again that k=20 is only appropriate when $p_{om} < 10^{-6}$.

Bitstate Hashing (7)

- Multiple Hashing, some typical numbers:
  - k ∈ {20…30} seems most appropriate
  - SPIN uses k=3 (since a few years, before it was k=2)
- Variation
  - partioning the bitstate array into p disjoint parts
  - each sub-array H_i is administered by its own independent hash function h_i
  - partioned bit-arrays do not perform better than a single, non-partioned bitstate array
  - non-partioned bitstate array will consume fewer bits per state, and thus has lower probability of error

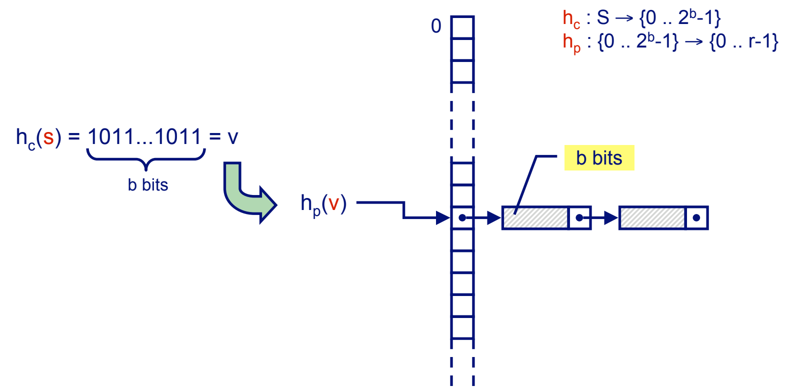
Hash Compaction (1)

[Wolper & Leroy 1993]

- Disadvantage of 1-fold bitstate hashing:
  - a very large bitarray is needed to ensure a high coverage
    - we are only interested in the 1s of the bitarray
    - the bitarray should be very sparse
- Idea of hash compaction: use indirection
  - hash function $h_c : S \rightarrow \{0 .. m\text{-}1\}$ where m is very large

    the bitarray of h_c is too large to fit in memory
    e.g. $m = 2^{64} = 1.8 \times 10^{19} = 2.1 \times 10^9$ Gb
  - compute $h_c(s)$
  - instead of setting $ba[h_c(s)] = 1$, store $h_c(s)$ in an ordinary hash table with hash function $h_p : \{0 .. m\text{-}1\} \rightarrow \{0 .. r\text{-}1\}$
  - thus $h_c(s)$ acts as a compressed value of s, and only this compressed value is stored

Hash Compaction (2)



Hash Compaction targets:
- a low probability of errors
- to avoid computational overhead (only 2 hash functions are employed)

Hash Compaction (3)

- Comparison:
  - Hash compaction outperforms 2-fold bitstate hashing by far.
  - Hash compaction also outperforms 20-fold bitstate hashing:
    - less memory is needed to ensure the same coverage
    - better run-time behaviour
- Disadvantage of hash compaction:
  - h_c and h_p are not independent
    - suppose $s1 \neq s2$ but $h_c(s1) = h_c(s2)$, then hash compaction scheme will store both values in the same slot of the hash table of h_p
    - (pure) hash compaction thus simulates 1-fold bitstate hashing in a naive way

Hash Compaction (4)

[Knottenbelt et. al. 1998]

- Dynamic Hash Compaction
  - solves the disadvantage of (classic) hash compaction
  - hash function $h_c : S \rightarrow \{0 .. m\text{-}1\}$ (with m very large)
  - hash function $h_s : \underline{S} \rightarrow \{0 .. r\text{-}1\}$

    The slot where $h_c(s)$ is stored is computed on basis of the original state s and not on basis of the compressed $h_c(s)$.
  - as a consequence, two states s1 and s2 are only considered equal, if both

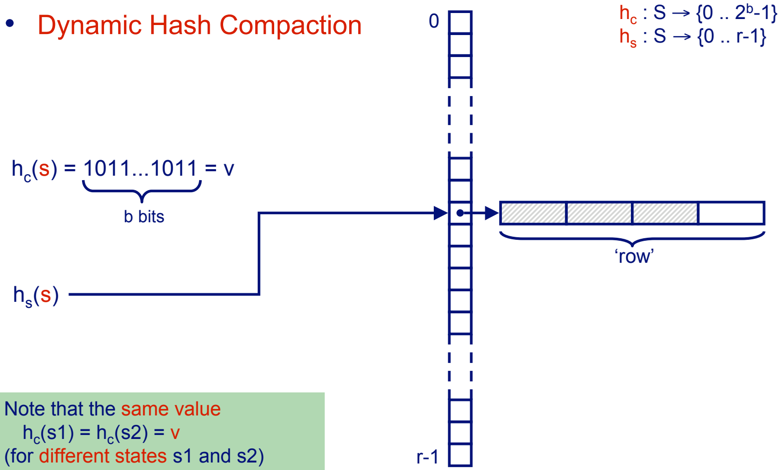
    $h_c(s1) = h_c(s2)$ *and* $h_s(s1) = h_s(s2)$
  - collisions are resolved by direct chaining

    Slot overflows are not stored in lists but in dynamic allocated arrays. Dynamic HC can be seen as 2-dimensional hashing scheme.

Hash Compaction (5)

- Dynamic Hash Compaction

$h_c : S \to \{0 .. 2^b\text{-}1\}$
$h_s : S \to \{0 .. r\text{-}1\}$



Note that the same value
$h_c(s1) = h_c(s2) = v$
(for different states s1 and s2)
may now appear in different rows.

Hash Compaction (6)

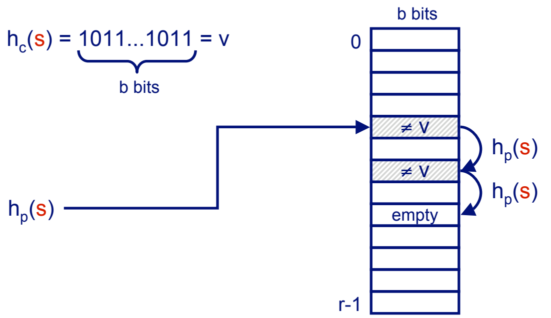
[Stern & Dill 1995]

- Improved Hash Compaction
  - hash function $h_c : S \to \{0 .. m\text{-}1\}$ (with m very large)
  - uses open addressing instead of chaining
    probing sequence $h_p : S \to \{0 .. r\text{-}1\}$
  - advantages:
    - h_c and h_p are independent
    - lower memory consumption (w.r.t. dynamic HC)
  - state insertion for a new state s using probing:
    - when slot at $h_p(s)$ is empty, store $h_c(s)$ at $h_p(s)$
    - when slot at $h_p(s)$ is not empty and this slot is not equal to $h_p(s)$, probing should continue
    - when slot at $h_p(s)$ is not empty and this slot is equal to $h_p(s)$, conclude that s has already been visited

Hash Compaction (6)

- Improved Hash Compaction

$h_c : S \to \{0 .. 2^b\text{-}1\}$
$h_p : S \to \{0 .. r\text{-}1\}$



Disadvantage of improved HC:
probing sequences might be long.

Hash Compaction (7)

- State Space Caching
  - stack stores only the states that are currently analysed
  - state table (or cache) only stores as many states as memory is available
    - if cache fills up, old states must be replaced by newly ones
  - pushes the limits of models that can be analysed on the expense of the run time
- Can be combined with the improved HC scheme
  - Only t slots are probed: after probing t slots the new state s replaces one of the t visited slots.
  - Note that replacement might also take place when the state table is not full.

Bitstate Hashing - Conclusions

- See [Kuntz & Lampka 2004] for all the details.
- Original bitstate hashing [Holzmann 1987] has led to several improvements:
  - original 1-fold bitstate hashing is still the fastest but is very inefficient with memory w.r.t. p_{cov}
  - 2-fold bitstate hashing does better with the memory requirements but is a bit slower
  - 30-fold is an improvement w.r.t. memory, but not in time
  - (classic) hash compaction is nearly as fast as 2-fold bitstate hashing but has much better memory characteristics
  - dynamic hash compaction improves on the coverage
  - improved hash compaction is slightly better on the memory requirements but slower due to the probing sequences

Iteratated Search Refinement

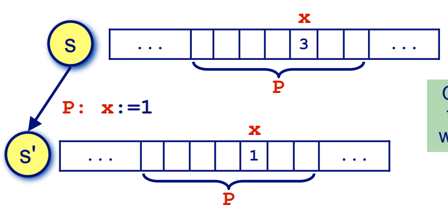
[Ernits 2007]

- Iterated Search Refinement through Bitstate Pruning
  - Counter-intuitive idea: use bitstate hashing in combination with a very small hash table (<< 1Mb) to search for an error
    - until an error is found
    - until the hash table is full
    - when no error has been found (i.e. the hash table is full), repeat the search with a slightly larger hash table
      - do this iteratively (or parallel!) until an error is found.
    - as the size of hash table changes, the hash function is also changing: different collissions for each run.
  - Assumptions:
    - state space is very well connected
    - several paths leading to the same error

Incremental Hashing

[Nguyen & Ruys SPIN 2008]

- Observation: transitions are local: for each transition only a small fraction of the state is changed.
  - Can we exploit the locality of transitions?



Given `h(s)` and the transion `x:=1`, can we compute `h(s')`?

```
void c_hash64(int i, int old, int new) {
    unsigned long knuth = 11400714819323198485UL;
    diff = (new*knuth) ^ (old*knuth);
    chash ^= ((diff << i) | (diff >> (64 - i)));
}
```

`chash` is the last hash value

IncHash is effective. But effect is minimalised when gcc's `-O3` switch is used.

But for bitstate hashing, it still pays off to use IncHash (gain: 10-20%).

Overview

Gerard J. Holzmann.
State Compression in SPIN: Recursive Indexing and Compression Training Runs.
Proceedings of the Third SPIN Workshop, SPIN 1997, Enschede.

Matthias Kuntz and Kai Lampka.
Probabilistic Methods in State Space Analysis.
In: C. Baier et. al. (Eds): Validation of Stochastic Systems. LNCS 2925, pp. 339-383. 2004.

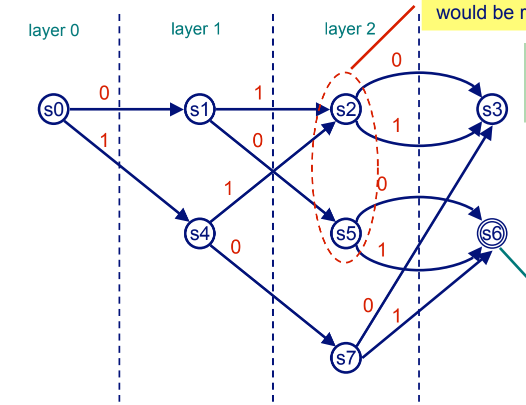
Gerard J. Holzmann and Anuj Puri.
A Minimized Automaton Representation of Reachable States.
STTT, Vol. 2, No. 3 (1999), pp. 270-278.

Minimised Automaton (1)

- Idea:
  - instead of storing the states in conventional lookup table
  - perform state matching by building and maintaining a minimal deterministic state automaton (DFA) which acts as a recogniser for sets of state descriptors
  - two important operations on MA
    - insertion
    - deletion

    both have a time complexity of $O(k\,|\Sigma|)$, where Σ is the alphabet, and k the length of the strings of Σ (i.e. state vector)
- Observation:
  - close resemblance between MA structure and OBDDs

Minimised Automaton (2)



would be removed in a OBDD.

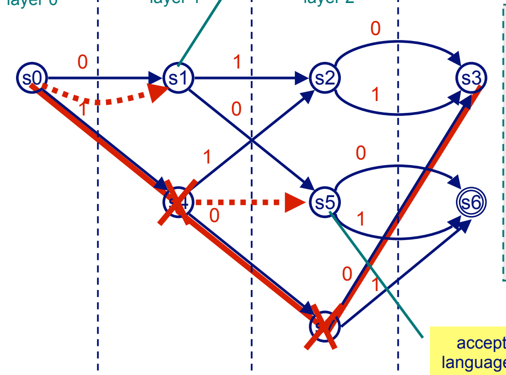
An OBDD can be obtained from a minimised DFA by removing all redundant states.

If path ends in the accepting state, the corresponding state s is in the set.

Minimised Automaton corresponding with the set {000,001,101}.

Minimised Automaton (3)

accepts the same language as s4 ∪ {00}.



accepts the same language as s7 ∪ {0}.

From n_i, the tail of σ should be recognized + all strings from the node of the prefix of σ.

We need to produce a new minimized DFA A' such that

$L(A') = L(A) \cup \{\sigma\}$

For each layer i we need a state n_i such that

$L_{A'}(n_i) = L_A(\sigma[0..i-1]) \cup \{\sigma[i..k-1]\}$

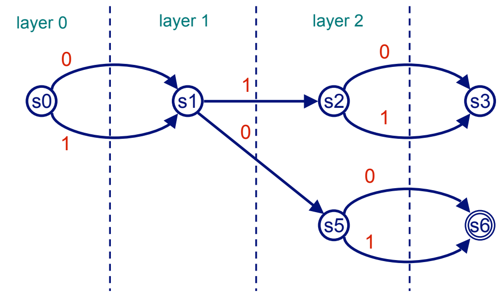
$\sigma[0..i-1] \in E_{A'}(n_i)$, where

$E_A(n)$ is the set of strings that end in node n.

In A', the prefix of σ should end in n_i.

Suppose we want to add state 100 to the set.

Minimised Automaton (4)



Minimised Automaton corresponding with the set {000,001,100,101}.

Minimised Automaton (5)

- Conclusions
  - Minimised Automaton has the same expected complexity as the standard search based on the storage in a hashed lookup table.
    - time complexity O(S), where S is the maximum length of the state descriptor
    - constant factors of both procedures are very different
    - minimised automaton can consume considerably more time than other optimisation techniques
  - But when memory is at a premium, it can be well worth the extra wait to use the more aggressive reduction technique.