State Space Exploration of Coloured Petri Nets

Lars M. Kristensen
Department of Computer Engineering
Bergen University College, NORWAY
Email: lmkr@hib.no /Web: www.hib.no/ansatte/lmkr
Explicit State Space Exploration

- Explicit state space exploration is the main approach to verification of CPN models:

1: \textbf{UNPROCESSED} $\leftarrow \{s_I\}$
2: \textbf{NODES.ADD}(s_I)
3: \textbf{while} $\neg$ \textbf{UNPROCESSED.EMPTY}() \textbf{do}
4: \textbf{s} $\leftarrow$ \textbf{UNPROCESSED.SELECT}()
5: \textbf{for all} $\langle e, s' \rangle$ such that $s \xrightarrow{e} s'$ \textbf{do}
6: \textbf{if} $\neg$(\textbf{NODES_CONTAINS}(s')) \textbf{then}
7: \textbf{NODES.ADD}(s')
8: \textbf{UNPROCESSED.ADD}(s')
9: \textbf{end if}
10: \textbf{end for}
11: \textbf{end while}

- Visited states (nodes)
- Unprocessed states
- Initial state and transition relation of the model
CPNs and State Space Methods

- A main guideline has been to support state space exploration of the full CPN modelling language:
  - The rich data types yields state vectors (markings) of typically 100-1000 bytes.
  - The expressive inscription language make it infeasible (in general) to exploit structural properties and rely unfolding to low-level Petri Nets.
  - Calculation of enabled events (binding elements) is expensive.

- Potentials of the CPN modelling language:
  - The possibility of compact modelling yields smaller state spaces (model-level reduction is very important).
  - The hierarchical structure facilitates sharing of sub-states.
  - Petri net locality can be exploited to reduce time spent on calculation of enabled events.
The Simple Protocol

- Transition \textit{SendPacket} can produce an unlimited number of tokens on place A.
- This means that the state space becomes \textit{infinite}:
Revised Protocol

- A new place **Limit** that limits the total number of tokens on the buffer places A, B, C, and D.
- This makes the state space **finite**.

\[
\text{colset UNIT = unit;}
\]
CPN Tools: State Space Exploration and Verification

- Typical steps in basic application of state space methods for verification of CPNs:
  - Generate the full state space of the CPN model.
  - Generate a state space report containing answers to a set of standard CPN behavioural properties.
  - Investigate system specific properties using the provided standard and user-defined query functions.

- Supports visualisation of state space fragments interactively or automatically:
  - Useful for debugging purposes.
  - Useful for visualisation of counter examples.

- Implementations of several advanced state space methods for alleviating state explosion.
State Space Report

- The state space report contains information about standard behavioural properties for CPNs.

- **Statistical information:**
  - Size and time used for state space generation.

- **Boundedness properties:**
  - Bounds for the number of tokens on each place (integer bounds)
  - Information about the possible token colours (multi-set bounds).

- **Home and liveness properties:**
  - List of home markings and list dead markings.
  - Dead and live transitions.

- **Fairness properties for transitions.**

- If the system contains errors this is very often reflected in the state space report.
Statistical Information

- State space contains more than 13,000 nodes and more than 52,000 arcs.
- State space was constructed in less than one minute and it is full (contains all reachable markings).
- The Strongly Connected Component Graph (SCC graph) is smaller (hence we have cycles).
- The SCC graph was constructed in 2 seconds.

**State Space Statistics**

<table>
<thead>
<tr>
<th></th>
<th>State Space</th>
<th>Scc Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes:</td>
<td>13,215</td>
<td>5,013</td>
</tr>
<tr>
<td>Arcs:</td>
<td>52,784</td>
<td>37,312</td>
</tr>
<tr>
<td>Secs:</td>
<td>53</td>
<td>2</td>
</tr>
<tr>
<td>Status:</td>
<td>Full</td>
<td></td>
</tr>
</tbody>
</table>
Integer bounds

- The **best upper integer bound** for a place is the **maximal number** of tokens on the place in a reachable marking.
- The **best lower integer bound** for a place is the **minimal number** of tokens on the place in a reachable marking.

<table>
<thead>
<tr>
<th>Best Integers Bounds</th>
<th>Upper</th>
<th>Lower</th>
</tr>
</thead>
<tbody>
<tr>
<td>PacketsToSend</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>DataReceived</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>NextSend, NextRec</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A, B, C, D</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Limit</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

- PacketsToSend has exactly 6 tokens in all reachable markings.
- DataReceived, NextSend and NextRec have exactly one token each in all reachable markings.
- The remaining five places have between 0 and 3 tokens each in all reachable markings.
Upper Multi-set Bounds

- The best upper multi-set bound specifies for each colour $c$ the maximal number of tokens with colour $c$ in a reachable marking.

Best Upper Multiset Bounds

- **PacketsToSend**: $1'(1,"COL")++1'(2,"OUR")++1'(3,"ED ")+\nonumber 1'(4,"PET")++1'(5,"RI ")+\nonumber 1'(6,"NET")$
- **DataReceived**: $1""++1""COL""++1""COLOUR""++1""COLOURED "+\nonumber 1""COLOURED PET""++1""COLOURED PETRI "+\nonumber 1""COLOURED PETRI NET"
- **NextSend, NextRec**: $1'1++1'2++1'3++1'4++1'5++1'6++1'7$
- **A, B**: $3'(1,"COL")++3'(2,"OUR")++3'(3,"ED ")+\nonumber 3'(4,"PET")++3'(5,"RI ")+\nonumber 3'(6,"NET")$
- **C, D**: $3'2++3'3++3'4++3'5++3'6++3'7$
- **Limit**: $3'(\ )$
Home Markings

- A **home marking** is a marking $M_{\text{home}}$ which can be reached from any reachable marking.

- Impossible to have an occurrence sequence which cannot be extended to reach $M_{\text{home}}$.

- There is a **single home marking** represented by node number 4868.

**Home Properties**

Home Markings: [4868]
Home Marking

Successful completion of transmission.

All packets have been received in the correct order

Receiver is waiting for packet no. 7

Sender is ready to send packet no. 7

All buffer places are empty
Liveness Properties

- A marking $M$ is **dead** if $M$ has no enabled transitions.
- A transition $t$ is **dead** if it is disabled in all reachable markings.
- A transition is **live** if it can be made enabled from any reachable marking:

<table>
<thead>
<tr>
<th>Initial marking</th>
<th>Arbitrary reachable marking</th>
<th>Marking where $t$ is enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_0$</td>
<td>$M_1$</td>
<td>$M_2$</td>
</tr>
</tbody>
</table>

Liveness Properties

- Dead Markings: [4868]
- Dead Transitions: None
- Live Transitions: None
Marking no 4868

- We have seen that marking $M_{4868}$ represents the state corresponding to successful completion of the transmission.
- $M_{4868}$ is the only dead marking.
- If the protocol execution terminates we are in the desired terminating state.

- $M_{4868}$ is also a home marking.
- Tells us that it always is possible to reach the desired terminating state.
Query Functions

- The state space report considers behavioural properties applicable to all CPN models.
- Model-specific properties can be investigated by means of user-defined queries.

- The queries typically consists of 5-20 lines of code written in Standard ML using:
  - A set of standard query functions.
  - A set of state space search function.
- The ASK-CTL library is also available for writing queries in a state-and-event variant of CTL.
Example: A User-defined Query Function

- Investigate whether the protocol obeys the **stop-and-wait** strategy:

```ml
fun StopWait n = 
    let
      val NextSend = ms_to_col (Mark.Protocol'NextSend 1 n);
      val NextRec = ms_to_col (Mark.Protocol'NextRec 1 n);
    in
      (NextSend = NextRec) orelse (NextSend = NextRec - 1)
    end;

val SWviolate = PredAllNodes (fn n => not (StopWait n));
```

- The **stop-and-wait** strategy is **not** satisfied (7020 violations).

We check whether some states violate the state predicate.
Counter Example

- The **binding elements** in the **path** can be obtained by the following query:

  \[
  \text{List.map (ArcToBE (ArcsInPath(1,557)))}
  \]

  - **Maps a state space arc into its binding element**
  - **State space arcs in one of the shortest paths leading to 557.**
  - **Lowest numbered node in the list SWviolate**

- The path can be **visualised** using the drawing facilities in the CPN Tools.
Counter Example

Packet no 1 and its ack
1 (SendPacket, <d="COL",n=1>)
2 (TransmitPacket, <n=1,d="COL",success=true>)
3 (ReceivePacket, <k=1,data="",n=1,d="COL">)
4 (SendPacket, <d="COL",n=1>)
5 (TransmitPacket, <n=2,d="COL",success=true>)
6 (ReceivePacket, <k=1,d="COL",n=1>)
7 (SendPacket, <d="OUR",n=2>)
8 (TransmitPacket, <n=1,d="COL",success=true>)
9 (TransmitPacket, <n=2,d="OUR",success=true>)
10 (ReceivePacket, <k=2,data="COL",n=1,d="COL">)
11 (ReceivePacket, <k=2,data="COL",n=2,d="OUR">)
12 (TransmitAck, <n=3,success=true>)
13 (ReceiveAck, <k=2,n=3>)

Packet no 2 and its ack

Packet no 3
NextRec = 4
Retransmission
NextSend = 2
14 (SendPacket, <d="ED ",n=3>)
15 (TransmitPacket, <n=3,d="ED ",success=true>)
16 (ReceivePacket, <k=3,data="COLOUR",n=3,d="ED ">)
17 (TransmitAck, <n=2,success=true>)
18 (ReceiveAck, <k=3,n=2>)
Violation of stop-and-wait strategy

- Acknowledgements **may overtake** each other on C and D.
- It is possible for the sender to receive an old acknowledgement which decrements NextSend.
Advanced State Space Methods for Coloured Petri Nets
The Sweep-line Method
[Christensen, Evangelista, Kristensen, Mailund, Westergaard]

- The basic idea is to exploit a certain kind of progress exhibited by many concurrent systems:
  - Retransmission counters and sequence numbers in protocols.
  - Commit phases in transaction protocols.
  - Control flow in programs and business processes.
  - Time in timed CPN models (value of global clock).
  - Object identifiers in OO-CPNs.
  - ...

- Makes it possible to explore all the reachable states.
- Storing only small state space fragments in memory at a time.
Example: On-the-fly Verification of the Datagram Congestion Control Protocol

- DCCP connection management proceeds in **phases**:

  ![Diagram showing phases of DCCP connection management]

**Marking of ClientState place specifies current phase**
Sweep-line Exploration

- The inherent progress is reflected also in the state space of the CPN model:

  The state space is explored layer-by-layer in progress-first order.
Progress Measure

- The progress is captured by a user-provided monotonic progress measure:

A monotonic progress measure is a tuple $P = (O, \sqsubseteq, \psi)$ s.t:

- $O$ is a set of progress values.
- $(O, \sqsubseteq)$ is a total order.
- $\psi : S \rightarrow O$ is a progress mapping satisfying:

$$\forall s, s' \in \text{reach}(s_I) : s \rightarrow^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$$

**DCCP: Client state progress mapping**

$$\psi(s) = \begin{cases} 
0 & \text{if CS}(s) = \text{IDLE} \\
1 & \text{if CS}(s) = \text{REQUEST} \\
2 & \text{if CS}(s) = \text{RESPOND} \\
3 & \text{if CS}(s) = \text{PARTOPEN} \\
4 & \text{if CS}(s) = \text{DATATRANSFER} \\
5 & \text{if CS}(s) = \text{CLOSING} 
\end{cases}$$

**Observation:**

Monotonicity can be checked fully automatically during state space exploration.
## DCCP: Experimental Results

[Vanit-Anunchai, Billington, Gallasch (2007)]

- Refined progress measure taking into account also server state and retransmission counters:

<table>
<thead>
<tr>
<th>Sweep-line $s$ specification model</th>
<th>Sweep-line $A$ augmented model</th>
<th>(S/C)*100</th>
<th>(A/C)*100</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>total</strong></td>
<td><strong>peak</strong></td>
<td><strong>hh:mm:ss</strong></td>
<td><strong>total</strong></td>
</tr>
<tr>
<td>nodes</td>
<td>nodes</td>
<td>space</td>
<td>time</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2,397</td>
<td>918</td>
<td>00:00:02</td>
<td>4,870</td>
</tr>
<tr>
<td>11,870</td>
<td>4,435</td>
<td>00:00:10</td>
<td>29,212</td>
</tr>
<tr>
<td>61,239</td>
<td>24,289</td>
<td>00:01:22</td>
<td>172,307</td>
</tr>
<tr>
<td>116,745</td>
<td>42,486</td>
<td>00:03:24</td>
<td>362,528</td>
</tr>
<tr>
<td>296,961</td>
<td>123,463</td>
<td>00:12:51</td>
<td>934,049</td>
</tr>
<tr>
<td><strong>964,862</strong></td>
<td><strong>354,710</strong></td>
<td><strong>01:59:47</strong></td>
<td><strong>3,970,455</strong></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>31,872,091</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>219,200,989</td>
</tr>
<tr>
<td>3,270</td>
<td>1,148</td>
<td>00:00:02</td>
<td>6,244</td>
</tr>
<tr>
<td>9,080</td>
<td>3,321</td>
<td>00:00:08</td>
<td>20,150</td>
</tr>
<tr>
<td>8,890</td>
<td>3,550</td>
<td>00:00:07</td>
<td>17,536</td>
</tr>
<tr>
<td>45,368</td>
<td>17,214</td>
<td>00:00:46</td>
<td>159,818</td>
</tr>
<tr>
<td>79,320</td>
<td>30,774</td>
<td>00:01:30</td>
<td>169,728</td>
</tr>
<tr>
<td>127,195</td>
<td>49,737</td>
<td>00:02:40</td>
<td>289,062</td>
</tr>
<tr>
<td>305,807</td>
<td>110,955</td>
<td>00:08:48</td>
<td>1,441,029</td>
</tr>
<tr>
<td>477,764</td>
<td>175,913</td>
<td>00:21:10</td>
<td>2,058,949</td>
</tr>
<tr>
<td>1,493,946</td>
<td>569,749</td>
<td>03:16:27</td>
<td>8,141,588</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>17,594,060</td>
</tr>
</tbody>
</table>
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, ...):

  - The state space contains **regress edges**.
  - Termination is no longer guaranteed.

Bang and Olufsen Lock Management Protocol
Generalised Sweep-line Method

- Cannot determine whether a destination state of a regress edge has already been explored:

- Detect regress edges during exploration and mark destination markings as **persistent**.
Algorithm and Implementation

1: Roots ← \{s_I\}
2: Nodes.Add(s_I)
3: while ¬(Roots.Empty()) do
4:   Unprocessed ← Roots
5:   Roots ← ∅
6:   while ¬(Unprocessed.Empty()) do
7:     s ← Unprocessed.GetMinElement()
8:     for all \((t, s')\) such that \(s \rightarrow t \rightarrow s'\) do
9:       if ¬(Nodes.Contains(s')) then
10:          Nodes.Add(s')
11:          if \(ψ(s) \supseteq ψ(s')\) then
12:             Nodes.MarkPersistent(s')
13:             Roots.Add(s')
14:          else
15:             Unprocessed.Add(s')
16:       end if
17:     end for
18:   end while
19:   Nodes.GarbageCollect(\(\min\{ψ(s) \mid s \in \text{Unprocessed}\}\))
20: 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.
- **Sub-state sharing** requires a reference count mechanism.
Sweep-Line Extensions

- Counter example generation is not immediately possible due to state deletion:
  - A inverse spanning tree can be written on external storage during state space exploration.
  - Each visited state is written to disk with an associated index pointing to its generating predecessor state.
  - Following the index pointers backwards yields the counter example (number of disk seeks proportional to path length).

- Sweep-line exploration suited for verification of safety properties:
  - In automata-based approaches a progress measure can be computed automatically on the property automata prior to parallel composition.
  - For CTL (LTL) model checking, state deletion can be replaced by storing only the value of atomic propositions for each marking.
The Comback Method
The Hash Compaction Method
[Wolper&Leroy’93, Stern&Dill’95]

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

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

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

<table>
<thead>
<tr>
<th>Full state descriptor</th>
<th>Compressed state descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(100-1000 bytes)</td>
<td>(4-8 bytes)</td>
</tr>
</tbody>
</table>
Example: Hash Compaction

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

  ![Diagram of state transitions]

  **State table:**

<table>
<thead>
<tr>
<th>h₁</th>
<th>h₂</th>
<th>h₃</th>
<th>h₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>s₂</td>
<td>s₃</td>
<td>s₄</td>
</tr>
</tbody>
</table>

  Compressed state descriptor
The Comback Method
[Arge, Brodal, Evangelista, Kristensen, Westergaard]

- Uses **backtracking** and state **reconstruction** of full state descriptors to guarantee full coverage.

- 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.
  - A **backedge table** stores a **backedge** for each state number of a visited state.

  **to detect (potential) hash collisions**

  **to reconstruct full state descriptors**
Collision list
Backedge table
Transition relation

State space of the mobile1 example
Example: The ComBack Method

Compressed state descriptor

State Reconstruction

Collision lists

Backedges

State table

Backedge table

Compressed state descriptor

State Reconstruction

Collision lists

Backedges
Comback 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:

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

Overhead compared to hash compaction

- Number of state reconstructions bounded by:

\[ \max_{h_k \in \hat{H}} |\hat{h}_k| \cdot \sum_{s \in \text{reach}(s_I)} \text{in}(s) \]
### Experimental Results

ComBack performance relative to standard DF full state space exploration

<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>Nodes</th>
<th>Arcs</th>
<th>%Time</th>
<th>%Space</th>
<th>%Time</th>
<th>%Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW</td>
<td>ComBack</td>
<td>215,196</td>
<td>1,242,386</td>
<td>178</td>
<td>42</td>
<td>258</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>HashComp</td>
<td>214,569</td>
<td>1,238,803</td>
<td>92</td>
<td>12</td>
<td>103</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>215,196</td>
<td>1,242,386</td>
<td>100</td>
<td>100</td>
<td>111</td>
<td>100</td>
</tr>
<tr>
<td>TS</td>
<td>ComBack</td>
<td>107,648</td>
<td>1,017,490</td>
<td>383</td>
<td>85</td>
<td>198</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>HashComp</td>
<td>107,647</td>
<td>1,017,474</td>
<td>93</td>
<td>75</td>
<td>96</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>107,648</td>
<td>1,017,490</td>
<td>100</td>
<td>100</td>
<td>106</td>
<td>73</td>
</tr>
<tr>
<td>ERDP</td>
<td>ComBack</td>
<td>207,003</td>
<td>1,199,703</td>
<td>180</td>
<td>34</td>
<td>353</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>HashComp</td>
<td>206,921</td>
<td>1,199,200</td>
<td>93</td>
<td>6</td>
<td>100</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Standard</td>
<td>207,003</td>
<td>1,199,703</td>
<td>100</td>
<td>100</td>
<td>115</td>
<td>101</td>
</tr>
<tr>
<td>ERDP</td>
<td>ComBack</td>
<td>4,277,126</td>
<td>31,021,101</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>HashComp</td>
<td>4,270,926</td>
<td>30,975,030</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Typically reduces memory usage to 30% at the cost of doubling the state space exploration time.
Summary

- The sweep-line method has been successfully applied to a number of real protocols:
  - Internet protocols: WAP, IOTP, TCP, and DCCP.
  - Progress is generally easy to identify and there is no proof obligation attached.

- The comeback method:
  - Search-order independent and transparent state reconstruction: compatible with most state space methods.
  - Experiments suggest that it represents a good time-space trade-off.
  - The Comback method is suited for late phases of the verification process.
Access/CPN Framework

JAVA and Standard ML interface providing access to the CPN model and the transitions relation:

- Explorations
  - Storages
  - Waiting sets
  - Query
  - Languages

CPN Tools Simulator

Standard ML

State Space Exploration Engine

References

The CPN Simulator Interface

- Defines how to access to the transition relation of CPN models:

  ```ocaml
  signature MODEL_SIMULATOR = sig
  
  eqtype state
  eqtype event
  
  (* --- get the initial state --- *)
  val getInitialState : unit -> state
  
  (* --- get the enabled events and successor states --- *)
  val nextStates : state -> (event * state) list
  
  end
  ```

- Makes it possible to extent CPN Tools and experiment with new state space methods.
State Representation

- Reflects the hierarchical structure of the CPN model:

```plaintext
type Receiver = {NextRec : INT.cs ms}
type Network = {}
type Sender = {NextSend : INT.cs ms}

type Protocol =
    {A : NOxDATA.cs ms,  
     B : NOxDATA.cs ms,  
     C : INT.cs ms,       
     D : INT.cs ms,       
     Limit : UNIT.cs ms,  
     Packets_To_Send : NOxDATA.cs ms,  
     Data_Received : STRING.cs ms,  
     Network : Network, Receiver : Receiver, Sender : Sender}

type state = { Protocol : Protocol }
```
Event Representation

datatype event
   Network'Transmit_Ack of
      int * {n : INT.cs, success : BOOL.cs}
   | Network'Transmit_Packet of
      int * {d : DATA.cs, n : INT.cs, success : BOOL.cs}
   | Receiver'Receive_Packet of
      int * {d : DATA.cs, data : DATA.cs,
             k : INT.cs,  n : INT.cs}
   | Sender'Receive_Ack of int * {k : INT.cs,  n : INT.cs}
   | Sender'Send_Packet of int * {d : DATA.cs, n : CPN'ColorSets.IntCS.cs}
end