State Space Exploration and ASAP: User Perspective

Michael Westergaard
Faculteit Wiskunde & Informatica
Technische Universiteit Eindhoven
m.westergaard@tue.nl

\[
\{ s_0 \} \\
\{ s_0 \} \\
W \neq \emptyset \hspace{2cm} \text{do} \\
\text{let } s \in W \\
W \setminus \{ s \} \\
\text{(s) then} \\
\text{return false} \\
\text{for all } t, s' \text{ such that } s \rightarrow t s' \text{ do} \\
\text{if } s' \notin V \text{ then} \\
V := V \cup \{ s' \} \\
W := W \cup \{ s' \} \\
\text{return true}
\]
Outline

- User perspective
- JoSEL
- Safety Properties
- Simple reduction techniques
- Linear temporal logic (LTL)
User Perspective
User Perspective
User Perspective

Take First

Take Second

Has One

Eating

Done

CHOP

[p = c or else (p + 1) mod n = c, c <> old]

[p = c or else (p + 1) mod n = c]
User Perspective
Verification of a model is done using verification projects consisting of:

- **CPN Models** to be analyzed
- **Queries** expressing the properties we are interested in
- **Reports** reflecting results of the queries
- How to obtain results from models and queries is described using verification jobs.
Example: Dining Philosophers

- **Take First**: $p = c$ or else $(p + 1) \mod n = c$
- **Take Second**: $p = c$ or else $(p + 1) \mod n = c$, $c \neq old$
- **Done**: $1 \rightarrow p++1 \rightarrow (p + 1) \mod n$
- **Eating**: $p$
- **Has One**: $p$
- **Take First**: $p$
- **Has One**: $p$
- **Take Second**: $p$
- **Done**: $1 \rightarrow p++1 \rightarrow (p + 1) \mod n$
Demo: Dining Philosophers

Do a bit of simple simulation
Example: Check for Deadlocks
Demo:
Check for Deadlocks

- Creation of Verification project
- Loading models
- Creating a Verification job from a template
- Executing a job template
- Reporting
No Dead States: false

Error trace:
New_Page.Has_One: 1'(0,0) ++ 1'(1,1) ++ 1'(2,2) ++ 1'(3,3) ++ 1'(4,4)
New_Page.Philosophers: 1'5

Error trace:
New_Page.Has_One: 1'(0,1) ++ 1'(1,2) ++ 1'(2,3) ++ 1'(3,4) ++ 1'(4,0)
New_Page.Philosophers: 1'5

Simulator Console:
- let open JavaExecute in
  case (SafetyChecker.explore true 0 (CPN'Structure'MLExplicitRemoveStorage'4.emptyStorage { init_size = 0 }())) (of [] => execute "result" []
    _ => ()
end
val it = () : unit
No Dead States  false

Error trace  New_Page.Has_One: 1'0(0) ++ 1'1(1) ++ 1'2(2) ++ 1'3(3) ++ 1'4(4)
New_Page.Philosophers: 1'5

Error trace  New_Page.Has_One: 1'0(0) ++ 1'1(1) ++ 1'2(2) ++ 1'3(3) ++ 1'4(4)
New_Page.Philosophers: 1'5

Done
Take Second  
[p = c orelse (p +1) mod n = c,
c <> old]  

Take First  
(p, c)

Has One  
(p, old)

Take Second  
[p = c orelse (p +1) mod n = c,
c <> old]

Eating  
PHIL

Chopsticks  

Done  
1'p++1'((p + 1) mod n)
JoSEL: Background

- ASAP supports a wide range of state space methods
- Depth-first and breadth-first traversal
- On-line and off-line analysis
- Bit-state hashing and hash compaction
- Sweep-line and ComBack methods
- Safety properties, LTL
Applying a state space methods consists of

1. Specifying a model to analyze
2. Making queries expressing desired properties
3. Select method to use for verification
4. Set parameters of and instantiate the selected method
5. Execute the traversal
6. Post-process and interpret the results
JoSEL: Aim

- Develop a high-level language making it possible to tie the model, queries and desired state space method together
- Support research, education and industrial application scenarios
JoSEL: Requirements

- **Abstraction**: Hide details from users
- **Low-level control**: Make it possible to access details when required for performance
  - The hash function used to hash states when storing in a hash table
- **Modularity**: Facilitate construction and use of building blocks (templates) in verification jobs
- **Extensibility**: Allow extension for new methods as needed
JoSEL Overview

Graphical language inspired by object flows and hierarchy of CP-nets

Basic unit is a task

Tasks have typed input and output ports
JoSEL Overview

Graphical language inspired by object flows and hierarchy of CP-nets

Basic unit is a **task**

Tasks have typed input and output **ports**
JoSEL Overview

- Graphical language inspired by object flows and hierarchy of CP-nets
- Basic unit is a **task**
- Tasks have typed input and output **ports**
Tasks

A task represents a single unit of work/operation: instantiating a data type, loading a file, checking a property, ...

Input ports represent data required to perform the operation

Output ports represent data produced by the operation

- Waiting Set Exploration
- Model Exploration
- Storage
- Waiting set
Output and input ports can be connected.

A verification job (job) is a set of tasks and their connections.
Output and input ports can be connected.

A verification job (job) is a set of tasks and their connections.
Jobs

- Connections represent flow of information
- Ports can have multiple connections
- Can represent split of information
- Can represent multiple instantiations
Deadlock Checker
Deadlock Checker

Flowchart:
- Input (IFile) → Instantiate Model (Model file) → Model
- Model → No Dead States (Model, Safety property)
- Safety Checker (Model, Answer, Property, Error trace) → Simple Report (Results)
Load a model

Deadlock Checker

Input
IFile

Instantiate Model
Model file
Model

Safety Checker
Model
Answer
Property
Error trace

No Dead States
Model
Safety property

Simple Report
Results
Load a model
...from this file

Deadlock Checker
Deadlock Checker

Load a model
...from this file

Instantiate the “no deadlock” property

Input
IFile

Instantiate Model
Model file
Model

Safety Checker
Model
Answer
Property
Error trace

Simple Report
Results

No Dead States
Model
Safety property
Load a model

...from this file

Instantiate the “no deadlock” property

...check the safety property for the model

Deadlock Checker
Deadlock Checker

Load a model
...from this file

Instantiate the “no deadlock” property

...and dump the results in a report

...check the safety property for the model

Input
IFile

Instantiate Model
Model file Model

Safety Checker
Model Answer
Property Error trace

No Dead States
Model Safety property

Simple Report
Results
Abstraction

The “Safety Checker” is not a single unit of work.

It is in fact a macro representing multiple tasks, such as instantiating a hash table and performing a BFS.
Abstraction

The “Safety Checker” is not a single unit of work

It is in fact a macro representing multiple tasks, such as instantiating a hash table and performing a BFS
Jobs can have exported ports.

Jobs can be represented by macro tasks (macros).
Jobs can have exported ports.

Jobs can be represented by macro tasks (macros).
Jobs can have exported ports

Jobs can be represented by macro tasks (macros)
Safety Checker
Check the given property using the given exploration
Check the given property using the given exploration

Stop after finding at most 10 errors
Stop after finding at most 10 errors

Check the given property using the given exploration

Build error-traces during exploration
Safety Checker

Check the given property using the given exploration

Stop after finding at most 10 errors

Build error-traces during exploration

Technical – just make sure the letters match

On-the-fly Safety Checker
- Exploration ST
- Property
- Model
- Storage
- Build traces
- Maximum of errors

Input
- Boolean
- Integer

Input
- Boolean
- Integer

Answer
- Error Trace

Pipe
- In
- Out

Queue
- Engine
- Waiting set

Waiting Set Exploration
- Model
- Exploration T

Exploration T → ST
- Exploration T
- Exploration ST
Stop after finding at most 10 errors

Build error-traces during exploration

Check the given property using the given exploration

Technical – just make sure the letters match

Exploration algorithm

On-the-fly Safety Checker

Exploration ST

Property

Model

Storage

Build traces

Maximum of errors

Input

Boolean

Input

Integer

Queue

Engine

Storage

Waiting set

Hash Storage

Model

Storage

Waiting set

Exploration T

Model

Exploration T

Exploration ST

Answer

Error Trace

Pipe

In

Out

Exploration algorithm
Safety Checker

Stop after finding at most 10 errors

Check the given property using the given exploration

Temporary storage is a queue

Build error-traces during exploration

Technical – just make sure the letters match

Exploration algorithm

On-the-fly Safety Checker
- Exploration ST
- Property
- Model
- Storage
- Build traces
- Maximum of errors

Input
- Boolean

Queue
- Engine
- Waiting set

Hash Storage
- Model
- Storage

Pipe
- In
- Out

Waiting Set Exploration
- Model
- Exploration TI

Exploration T -> ST
- Exploration T
- Exploration ST

Temporary storage is a queue
Temporary storage is a queue

Exploration algorithm

Technical – just make sure the letters match

Stop after finding at most 10 errors

Build error-traces during exploration

Check the given property using the given exploration

Permanent storage is a hash table

Technical – just make sure the letters match

Exploration algorithm
Safety Checker

Check the given property using the given exploration

Stop after finding at most 10 errors

Build error-traces during exploration

Technical – just make sure the letters match

Temporary storage is a queue

Exploration algorithm

Technical – just make sure the letters match

Permanent storage is a hash table

Technical – allows us to only specify the model once on the level above

On-the-fly Safety Checker

- Exploration ST
- Property
- Model
- Storage
- Build traces
- Maximum of errors

Input
- Boolean

Input
- Integer
Safety Checker
Hash Storage

CPN Tools Hash Function 1
- Model
- Hash function

Hash Table Storage
- Model
- Storage
- Hash function
- Initial size

Pipe
- In
- Out

Input
- Integer

Instantiate the permanent storage
Hash Storage

...using this (built-in) hash function

CPN Tools Hash Function 1

- Model
- Hash function

Hash Table Storage

- Model
- Storage
- Hash function
- Initial size

Pipe

In

Out

Instantiate the permanent storage

Hash Storage
Hash Storage

...using this (built-in) hash function

CPN Tools Hash Function 1
Model Hash function

Input
Integer

Hash Table Storage
Model Storage
Hash function
Initial size

Instantiate the permanent storage

...and initially make room for 1000 states (expands automatically)
Technical – allows us to only specify the model once on the level above

...using this (built-in) hash function

CPN Tools Hash Function 1

Model Hash function

Input Integer

Hash Table Storage

Model Storage

Hash function

Initial size

Instantiate the permanent storage

...and initially make room for 1000 states (expands automatically)
Safety Properties

Sometimes we may want to check properties other than absence of deadlocks.

Custom properties are created using SML.

ASAP automatically generates a template formula tailored to a specific model.
Example: Mutual Exclusion
Example: Mutual Exclusion

- We want to check that two adjacent philosophers cannot be eating at the same time.
- I.e., that they are not allowed access to a shared resource (chop-stick) at the same time.
- This is equivalent to checking that if philosopher \( p \) is eating, then philosopher \( p+1 \) is not (mod \( n \)).
A Bit of SML

Check if there is an element “p’ ” in “lst” that satisfies the predicate “f(p’)”:
\[
\text{List.exists (fn p' => f(p')) lst}
\]

Check if “2 + 1 mod 7” belongs to a list, “lst”:
\[
\text{List.exists (fn p' => p' = (2 + 1) mod 7) lst}
\]

Check if “p + 1 mod n” belongs to a list, “lst”:
\[
\text{List.exists (fn p' => p' = (p + 1) mod n) lst}
\]

Check if there is an element “p” in “lst” such that “p + 1 mod n” belongs to “lst”:
\[
\text{List.exists (fn p => List.exists (fn p' => p' = (p + 1) mod n) lst) lst}
\]

Yes, this is inefficient; we can sort “lst” and only compare neighbors
Example: Mutual Exclusion

```plaintext
fun query (state, events) = 
  let
    fun query'New_Page { Waiting, Has_One, Eating, 
                         Philosophers, Initialized, 
                         Chopsticks } = true
    fun query'state { New_Page} = query'New_Page New_Page
  in
    query'state state
  end
```
Example: Mutual Exclusion

```haskell
fun query (state, events) = let
    fun query'New_Page { Waiting, Has_One, Eating,
                         Philosophers,Initialized,
                         Chopsticks } =
        not (List.exists (fn p => List.exists
                           (fn p' => p' =
                           (p + 1) mod (List.hd Philosophers)
                           ) Eating) Eating)
    fun query'state { New_Page} = query'New_Page New_Page
    in
    query'state state
```

Example:

Mutual Exclusion
Demo: Mutual Exclusion

- Create property
- Edit JoSEL job
- Run checker
Example: On-line vs. Off-line
Off-line Safety Checker

\[
V := \{ \, s_0 \, \}
\]
\[
W := \{ \, s_0 \, \}
\]

while \( W \neq \emptyset \) do

Select an \( s \in W \)

\[
W := W \setminus \{ \, s \, \}
\]

for all \( t, s' \) such that \( s \xrightarrow{t} s' \) do

if \( s' \notin V \) then

\[
V := V \cup \{ \, s' \, \}
\]
\[
W := W \cup \{ \, s' \, \}
\]

for all \( v \in V \) do

if \( \neg I(v) \) then

return false

return true

This is off-line analysis; we first generate the state space and then we analyze it.
On-line Safety Checker

\[ V := \{ s_0 \} \]
\[ W := \{ s_0 \} \]

while \( W \neq \emptyset \) do

Select an \( s \in W \)

\[ W := W \setminus \{ s \} \]

if \( \neg I(s) \) then

return false

for all \( t, s' \) such that \( s \rightarrow^t s' \) do

if \( s' \notin V \) then

\[ V := V \cup \{ s' \} \]
\[ W := W \cup \{ s' \} \]

return true

This is on-line analysis; we analyze the state space while we generate it.
<table>
<thead>
<tr>
<th>On-line</th>
<th>Off-line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finds errors faster</td>
<td>Can check additional properties subsequently</td>
</tr>
<tr>
<td>Uses less memory</td>
<td>Can (easier) provide error traces</td>
</tr>
<tr>
<td>Supported by ASAP</td>
<td>Can check more properties</td>
</tr>
<tr>
<td></td>
<td>Supported by Design/CPN, CPN Tools, and ASAP</td>
</tr>
</tbody>
</table>
Demo:
On-line vs. Off-line

- Show safety checker and time spent checking property (maybe crank up size)
- Change to off-line
- Note that top-level has not changed
- Show time spent checking property
Example:
Hash-compaction
Hash-compaction

A problem of the standard method is that we use 1000 bytes per state, and $4 \text{ GB} / 1000 = 4 \cdot 10^6$ states.

What if we only use, say, 4 bytes per state; then we can store $4 \text{ GB} / 4 = 10^9$ states.

This is the rationale behind hash-compaction.
Observation

For a hash function $h$ (any function, really) we have

$$s = s' \implies h(s) = h(s')$$

We use the terminology

- $s$: full state descriptor (1000 bytes)
- $h(s)$: compressed state descriptor (4 bytes)

We do not have that $h(s) = h(s') \implies s = s'$, but good hash functions ensure that this is mostly true.

If $h(s) = h(s')$ but $s \neq s'$ we say we have a hash collision.
V := \{ s_0 \}
W := \{ s_0 \}

while W ≠ ∅ do
  Select an s ∈ W
  W := W \ { s }
  if ¬I(s) then
    return false
  for all t, s′ such that s →^t s′ do
    if s′ ∉ V then
      V := V ∪ \{ s′ \}
      W := W ∪ \{ s′ \}
  return true

We replace full state descriptors by compressed state descriptors in V
Hash-compaction

\[ V := \{ h(s_0) \} \]
\[ W := \{ s_0 \} \]

while \( W \neq \emptyset \) do

Select an \( s \in W \)

\[ W := W \setminus \{ s \} \]

if \( \neg I(s) \) then

return false

for all \( t, s' \) such that \( s \rightarrow^t s' \) do

if \( h(s') \notin V \) then

\[ V := V \cup \{ h(s') \} \]
\[ W := W \cup \{ s' \} \]

return true

We replace full state descriptors by compressed state descriptors in \( V \).
Hash-compaction

\[ V := \{ h(s_0) \} \]
\[ W := \{ s_0 \} \]

while \( W \neq \emptyset \) do
    Select an \( s \in W \)
    \[ W := W \setminus \{ s \} \]
    if \( \neg I(s) \) then
        return false
    for all \( t, s' \) such that \( s \rightarrow^t s' \) do
        if \( h(s') \notin V \) then
            \[ V := V \cup \{ h(s') \} \]
            \[ W := W \cup \{ s' \} \]
    return true

As long as we encounter no hash collisions, this algorithm works identically to the previous.

We replace full state descriptors by compressed state descriptors in \( V \).
Example
Example
Example

V: h1
W: s1
Example

V: h1
W:
Example

V: h1
W:
Example

V: h1 h2
W: s2
Example

V: h1 h2 h3
W: s2 s5
Example

V: h1 h2 h3
W: s1 s2 s5
Example

V: h1 h2 h3
W: s5
Example

V: h1 h2 h3
W: s5
Example

V: h1 h2 h3
W: s5
Example

V: h1 h2 h3
W: s5
Example

V:  h1 h2 h3
W:
Example

V: h1 h2 h3 h4
W: s4
Example

V: h1 h2 h3 h4
W: s4
Example

V: h1 h2 h3 h4
W: s4
Example

V: h1 h2 h3 h4
W:
Example

V: h1 h2 h3 h4
W:
Example

Incorrect edge

V: h1 h2 h3 h4
W:
Example

Incorrect edge

V: h1 h2 h3 h4
W: Never discovered
Notes on Hash-compaction

- We find most but not all states
- Improve coverage by using larger hash values
- Improve coverage using more than one hash function
- SHA-1 uses 160 bits (20 bytes) per state and has no known collisions
- Uses around as much time as the standard algorithm and space is still $O(\# \text{ nodes})$ but with a smaller factor
Demo: Hash-compaction

- Replace storage in standard method
  - We *can* but *should not* compute error traces
  - If we use DFS traversal, computing error traces is no problem
<table>
<thead>
<tr>
<th>Model</th>
<th>Nodes</th>
<th>NodesHC</th>
<th>Mem</th>
<th>MemHC</th>
<th>%</th>
<th>/st</th>
<th>/stHC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP22</td>
<td>39604</td>
<td>39603</td>
<td>23.6</td>
<td>20.8</td>
<td>88</td>
<td>625</td>
<td>550</td>
</tr>
<tr>
<td>DB10</td>
<td>196832</td>
<td>196798</td>
<td>174.0</td>
<td>4.9</td>
<td>3</td>
<td>927</td>
<td>26</td>
</tr>
<tr>
<td>SW7,4</td>
<td>215196</td>
<td>214569</td>
<td>43.0</td>
<td>5.2</td>
<td>12</td>
<td>210</td>
<td>25</td>
</tr>
<tr>
<td>TS5</td>
<td>107648</td>
<td>107647</td>
<td>61.2</td>
<td>45.7</td>
<td>75</td>
<td>596</td>
<td>445</td>
</tr>
<tr>
<td>ERDP2</td>
<td>207003</td>
<td>206921</td>
<td>87.4</td>
<td>5.1</td>
<td>6</td>
<td>443</td>
<td>26</td>
</tr>
<tr>
<td>ERDP3</td>
<td>4277126</td>
<td>4270926</td>
<td>-</td>
<td>113.5</td>
<td>-</td>
<td>-</td>
<td>28</td>
</tr>
<tr>
<td>Model</td>
<td>Nodes</td>
<td>NodesHC</td>
<td>Mem</td>
<td>MemHC</td>
<td>%</td>
<td>/st</td>
<td>/stHC</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>---------</td>
<td>------</td>
<td>-------</td>
<td>----</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>DP22</td>
<td>39604</td>
<td>39603</td>
<td>23.6</td>
<td>20.8</td>
<td>88</td>
<td>550</td>
<td></td>
</tr>
<tr>
<td>DB10</td>
<td>196832</td>
<td>196798</td>
<td>174.0</td>
<td>4.9</td>
<td>3</td>
<td>927</td>
<td>26</td>
</tr>
<tr>
<td>SW7,4</td>
<td>215196</td>
<td>214569</td>
<td>43.0</td>
<td>5.2</td>
<td>12</td>
<td>210</td>
<td>25</td>
</tr>
<tr>
<td>TS5</td>
<td>107648</td>
<td>107647</td>
<td>61.2</td>
<td>45.7</td>
<td>75</td>
<td></td>
<td>445</td>
</tr>
<tr>
<td>ERDP2</td>
<td>207003</td>
<td>206921</td>
<td>87.4</td>
<td>5.1</td>
<td>6</td>
<td>443</td>
<td>26</td>
</tr>
<tr>
<td>ERDP3</td>
<td>4277126</td>
<td>4270926</td>
<td>-</td>
<td>113.5</td>
<td>-</td>
<td></td>
<td>28</td>
</tr>
</tbody>
</table>
Example: Bit-state Hashing
Bit-state Hashing

Hash-compaction uses a hash function to compress state descriptor and stores the compressed vectors.

Bit-state hashing instead uses a hash function to compute an index in an array and sets a bit if a corresponding state has been seen.

We need an array of size $2^{|h(s)|}/8$ bytes, e.g., $2^{32}/8 = 500$ Mb to get same coverage as hash compaction.
Hash-compaction

\[ V := \{ s_0 \} \]
\[ W := \{ s_0 \} \]

while \( W \neq \emptyset \) do

- Select an \( s \in W \)
- \( W := W \setminus \{ s \} \)
- if \( \neg I(s) \) then
  - return false
- for all \( t, s' \) such that \( s \rightarrow^t s' \) do
  - if \( s' \notin V \) then
    - \( V := V \cup \{ s' \} \)
    - \( W := W \cup \{ s' \} \)
- return true

We replace full state descriptors with bit-array access.
Hash-compaction

\[ V := \text{new bool}[2^{|h(s)|}]; V[h(s_0)] := true \]
\[ W := \{ s_0 \} \]
\[ \textbf{while } W \neq \emptyset \textbf{ do} \]
\[ \quad \text{Select an } s \in W \]
\[ \quad W := W \setminus \{ s \} \]
\[ \quad \text{if } \neg I(s) \text{ then } \]
\[ \quad \quad \textbf{return} \text{ false} \]
\[ \quad \textbf{for all } t, s' \text{ such that } s \rightarrow^t s' \text{ do} \]
\[ \quad \quad \text{if } \neg V[h(s')] \text{ then} \]
\[ \quad \quad \quad V[h(s')] := true \]
\[ \quad \quad W := W \cup \{ s' \} \]
\[ \textbf{return} \text{ true} \]

We replace full state descriptors with bit-array access.
Hash-compaction

\[ V := \text{new bool}[2^{|h(s)|}] ; \ V[h(s_0)] := \text{true} \]

\[ W := \{ \ s_0 \} \]

while \( W \neq \emptyset \) do
  Select an \( s \in W \)
  \[ W := W \setminus \{ s \} \]
  if \( \neg I(s) \) then
    return false
  for all \( t, s' \) such that \( s \rightarrow^t s' \) do
    if \( \neg \ V[h(s')] \) then
      \[ \ V[h(s')] := \text{true} \]
      \[ W := W \cup \{ s' \} \]
  return true

This works exactly like hash-compaction with the same hash function.

We replace full state descriptors with bit-array access.

new bool[2^{|h(s)|}]; V[h(s_0)] := true
W := { s_0 }
while W ≠ ∅ do
  Select an s ∈ W
  W := W \setminus { s }
  if ¬I(s) then
    return false
  for all t, s' such that s →^t s' do
    if ¬V[h(s')] then
      V[h(s')] := true
      W := W ∪ { s' }
  return true

We replace full state descriptors with bit-array access.
Bit-state Hashing vs. Hash-compaction

Both allow us to increase the size of the compressed state descriptor to get better coverage, but for bit-state hashing each extra bit doubles memory usage.

Hash-compaction uses memory proportional to the size of the number of nodes, bit-state hashing uses a constant amount of memory.

Hash-compaction uses memory proportional to the number of hash functions we use, bit-state hashing uses a constant amount of memory.
Bit-state Hashing vs. Hash-compaction

Both allow us to increase the size of the compressed state descriptor to get better coverage, but for bit-state hashing each extra bit doubles memory usage.

Hash-compaction uses memory proportional to the size of the number of nodes, bit-state hashing uses a constant amount of memory.

Hash-compaction uses memory proportional to the number of hash functions we use, bit-state hashing uses a constant amount of memory.
More Hash Functions

Using 2 hash functions require that we have 2 collisions instead of just one.

But we may have a new kind of collisions,
\[ h_1(s_1) = h_2(s_2) \]

Using more hash functions improves coverage to a certain point where the bit-array gets "filled up", so collisions become more common.
Hash-compaction

\[ V := \text{new bool}[2^{h(s)}]; \quad V[h(s_0)] := \text{true} \]

\[ W := \{ s_0 \} \]

\[ \text{while } W \neq \emptyset \text{ do} \]

\[ \quad \text{Select an } s \in W \]

\[ \quad W := W \setminus \{ s \} \]

\[ \quad \text{if } \neg I(s) \text{ then} \]

\[ \quad \quad \text{return false} \]

\[ \quad \text{for all } t, s' \text{ such that } s \rightarrow^t s' \text{ do} \]

\[ \quad \quad \text{if } \neg V[h(s')] \text{ then} \]

\[ \quad \quad \quad V[h(s')] := \text{true} \]

\[ \quad \quad W := W \cup \{ s' \} \]

\[ \quad \text{return true} \]

We simply set and read bits for both (or all) hash functions.
Hash-compaction

\[ V := \text{new bool}[2^{\lvert h(s) \rvert}]; \ V[h(s_0)] := \text{true} \\]
\[ W := \{ s_0 \} \]
\[ \text{while } W \neq \emptyset \text{ do} \]
\[ \quad \text{Select an } s \in W \]
\[ \quad W := W \setminus \{ s \} \]
\[ \quad \text{if } \neg I(s) \text{ then} \]
\[ \quad \quad \text{return } \text{false} \]
\[ \quad \text{for all } t, s' \text{ such that } s \xrightarrow{t} s' \text{ do} \]
\[ \quad \quad \text{if } \neg V[h(s')] \text{ or } \neg V[h_2(s')] \]
\[ \quad \quad \quad V[h(s')] := \text{true}; \ V[h_2(s')] := \text{true} \]
\[ \quad W := W \cup \{ s' \} \]
\[ \text{return } \text{true} \]

We simply set and read bits for both (or all) hash functions.
Double Hashing

Calculating hash functions is actually pretty expensive, so the time complexity grows with the number of hash functions.

Simply using $h_n(s) = n \cdot h_1(s)$ does not work!

It turns out that using $h_n(s) = n \cdot h(s) + h'(s)$ does work; this is called double hashing.

Triple hashing works better but takes more time.

Experiments show that using 15-20 hash functions works well.
Demo:
Bit-state Hashing

Replace storage on standard example
## Numbers

<table>
<thead>
<tr>
<th>Model</th>
<th>Nodes</th>
<th>NodesDH</th>
<th>Mem</th>
<th>MemDH</th>
<th>%</th>
<th>/st</th>
<th>/stDH</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP22</td>
<td>39604</td>
<td>39604</td>
<td>23.6</td>
<td>32.0</td>
<td>135</td>
<td>625</td>
<td>846</td>
</tr>
<tr>
<td>DB10</td>
<td>196832</td>
<td>196832</td>
<td>174.0</td>
<td>12.3</td>
<td>7</td>
<td>927</td>
<td>66</td>
</tr>
<tr>
<td>SW7,4</td>
<td>215196</td>
<td>215196</td>
<td>43.0</td>
<td>12.3</td>
<td>28</td>
<td>210</td>
<td>60</td>
</tr>
<tr>
<td>TS5</td>
<td>107648</td>
<td>107648</td>
<td>61.2</td>
<td>55.4</td>
<td>90</td>
<td>596</td>
<td>540</td>
</tr>
<tr>
<td>ERDP2</td>
<td>207003</td>
<td>207003</td>
<td>87.4</td>
<td>12.3</td>
<td>14</td>
<td>443</td>
<td>62</td>
</tr>
<tr>
<td>ERDP3</td>
<td>4277126</td>
<td>4277125</td>
<td>-</td>
<td>12.1</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
</tbody>
</table>
## More Numbers

<table>
<thead>
<tr>
<th>Model</th>
<th>Nodes</th>
<th>MemHC</th>
<th>MemDH</th>
<th>/stateHC</th>
<th>/stateDH</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP22</td>
<td>39604</td>
<td>20.8</td>
<td>32.0</td>
<td>550</td>
<td>846</td>
</tr>
<tr>
<td>DB10</td>
<td>196832</td>
<td>4.9</td>
<td>12.3</td>
<td>26</td>
<td>66</td>
</tr>
<tr>
<td>SW7,4</td>
<td>215196</td>
<td>5.2</td>
<td>12.3</td>
<td>25</td>
<td>60</td>
</tr>
<tr>
<td>TS5</td>
<td>107648</td>
<td>45.7</td>
<td>55.4</td>
<td>445</td>
<td>540</td>
</tr>
<tr>
<td>ERDP2</td>
<td>207003</td>
<td>5.1</td>
<td>12.3</td>
<td>26</td>
<td>62</td>
</tr>
<tr>
<td>ERDP3</td>
<td>4277126</td>
<td>113.5</td>
<td>12.1</td>
<td>28</td>
<td>3</td>
</tr>
</tbody>
</table>
Comparing the Top-levels
Example: Drawing SS Graphs
Demo:
Drawing SS Graphs

- Change safety checker to draw SS graph
- Change model size to 2 philosophers
- Play with layouts
- Export to DOT and GML
Example:
Simple Protocol
Example:
Simple Protocol

10 packets
Example:
Simple Protocol

10 packets
Example:
Simple Protocol

3000 nodes

10 packets
Demo: Error Traces

- Displaying error trace
- Displaying multiple error traces in a single window
Linear Temporal Logic

Until now we have only dealt with safety properties (i.e., what happens in one state).

Temporal logics allow us to talk about several states.
Propositional Logic

Atomic Propositions

\[ \text{AP} = \{ p, q, r, \ldots \} \]

Syntax

\[ \varphi ::= p \mid \neg \varphi \mid \varphi \rightarrow \psi \]

We call all such formulas Prop

A formula \( \varphi \) holds for a system if it holds in all reachable states

We use SML functions for atomic propositions

We have not really used connectors until now
Example

AP = \{•, •, •\}

APs holding in a state are shown inside the state
Example

AP = \{ \cdot, \cdot, \cdot \}

APs holding in a state are shown inside the state

\phi = \cdot \lor \cdot
AP = \{\bullet, \bullet, \bullet\} 

APs holding in a state are shown inside the state 

\[ \phi = \bullet \lor \bullet \]
AP = \{ \bullet, \bullet, \bullet \}

APs holding in a state are shown inside the state.

\[ \varphi = \bullet \lor \bullet \]

\[ \varphi' = \bullet \]
AP = \{ •, •, •, • \}

APs holding in a state are shown inside the state

\( \varphi = • \lor • \)

\( \varphi' = • \)

Example
Example

AP = \{ \bullet, \bullet, \bullet, \bullet \}

APs holding in a state are shown inside the state

φ' = \circ \text{ does not hold}
Example

AP = \{ \bullet, \bullet, \bullet \}

APs holding in a state are shown inside the state

\phi' = \bullet \text{ does not hold}

...but after executing “some” transitions, it does...
Example: Dining Philosophers

\[ p = \text{philosopher 1 eats} \]

Philosopher 1 always eats: \( \phi = p \)

Philosopher 1 never eats: \( \phi' = \neg p \)
Example: Dining Philosophers

p = philosopher 1 eats

Philosopher 1 may eat?
Example: Dining Philosophers

p = philosopher 1 eats

Philosopher 1 may eat?

...we can actually check this property:

• check \( \varphi' = \neg p \)
• answer is the opposite
Example: Dining Philosophers

$\varphi' = false \iff$

$\varphi'$ does not hold in all states $\iff$

$\neg p$ does not hold in all states $\iff$

$p$ holds in at least one state

$p = \text{philosopher 1 eats}$

Philosopher 1 may eat?

...we can actually check this property:

- check $\varphi' = \neg p$
- answer is the opposite
Example: Dining Philosophers

\[ p = \text{philosopher 1 eats} \]

Philosopher 1 will always eat at some point
Example: Dining Philosophers

$p = \text{philosopher 1 eats}$

Philosopher 1 will always eat at some point...we cannot check this (unless "eat at some point" is an atomic proposition)
PLTL Syntax

Atomic Propositions

AP = \{ p, q, r, ... \}

Syntax

\( \phi ::= p \mid \neg \phi \mid \phi \to \psi \mid X \phi \mid \phi U \psi \)

Add some syntactical sugar

\( F\phi \equiv \text{true} U \phi \) (also written \( \Diamond \phi \))

\( G\phi \equiv \neg F\neg \phi \) (also written \( \Box \phi \))
PLTL Syntax

Atomic Propositions

\[ AP = \{ p, q, r, \ldots \} \]

Syntax

\[ \phi ::= p \mid \neg \phi \mid \phi \rightarrow \psi \mid X \phi \mid \phi U \psi \]

Add some syntactical sugar

\[ F\phi \equiv \text{true} U \phi \text{ (also written } \Diamond \phi) \]

\[ G\phi \equiv \neg F\neg \phi \text{ (also written } \Box \phi) \]
PLTL Syntax

Atomic Propositions

\( \text{AP} = \{ p, q, r, \ldots \} \)

Syntax

\[ \phi ::= p \mid \neg \phi \mid \phi \rightarrow \psi \mid X \phi \mid \phi U \psi \]

Add some syntactical sugar

\[ F\phi \equiv \text{true U } \phi \text{ (also written } \Diamond \phi) \]

\[ G\phi \equiv \neg F \neg \phi \text{ (also written } \Box \phi) \]

In the next state \( \phi \) holds

\( \phi \) holds until \( \psi \) holds (and \( \psi \) holds eventually)
Atomic Propositions

AP = \{ p, q, r, \ldots \}

Syntax

\( \phi ::= p \mid \neg \phi \mid \phi \rightarrow \psi \mid X \phi \mid \phi U \psi \)

Add some syntactical sugar

\( F\phi \equiv \text{true U } \phi \) (also written \( \Diamond \phi \))

\( G\phi \equiv \neg F\neg \phi \) (also written \( \square \phi \))

\( \phi \) holds at some point (future, eventually, possibly)

In the next state \( \phi \) holds

\( \phi \) holds until \( \psi \) holds (and \( \psi \) holds eventually)
Atomic Propositions

AP = { p, q, r, … }

Syntax

φ ::= p | ¬φ | φ → ψ | X φ | φ U ψ

Add some syntactical sugar

Fφ ≣ true U φ (also written ⃟φ)

Gφ ≣ ¬F¬φ (also written ⃤φ)

φ holds at some point (future, eventually, possibly)

In the next state φ holds

φ holds until ψ holds (and ψ holds eventually)

φ holds in all states (everywhere, globally, necessarily)
Example
Example
Example
Example
For a property to hold for a state space, it must hold along all (infinite) paths.
Example
Example: Dining Philosophers

p = philosopher 1 eats

Philosopher 1 will always eat at some point
Example: Dining Philosophers

$p = \text{philosopher 1 eats}$

Philosopher 1 will always eat at some point

- $\phi'' = F \ p$
- $\phi''' = G \ F \ p$
LTL Examples

- **Safety** (nothing bad happens): $G \neg \text{bad}$

- **Liveness** (something good happens): $F \text{good}$

- **Response** (requests are eventually serviced): $G(\text{request} \rightarrow F \text{serviced})$

- **Reactiveness** (infinite number of requests means an infinite number are serviced): $GF \text{sent} \rightarrow GF \text{received}$
Checking LTL

- $L(M)$ language of a model $M$ (i.e., all possible executions of $M$)
- $L(\varphi)$ language of a formula $\varphi$ (i.e., all traces satisfying $\varphi$)

We want to check that

$L(M) \subseteq L(\varphi) \iff L(M) \cap L(\varphi)^c = \emptyset$

$\iff L(M) \cap L(\neg \varphi) = \emptyset$
Checking LTL (2)

We want to check that \( L(M) \cap L(\neg \phi) = \emptyset \)

We can construct a Büchi automaton \( A_{\neg \phi} \)
such that \( L(\neg \phi) = L(A_{\neg \phi}) \)

The state space \( SS_M \) is essentially a Büchi automaton representing \( L(M) \)

We thus check whether \( L(SS_M \times A_{\neg \phi}) = \emptyset \)

The product is (essentially) equal to the product construction for finite automata
Checking LTL (3)

A Büchi automaton is a finite automaton but in order for a word to be accepted, we must go thru an accept state infinitely often.

As it is finite, this means we must visit (at least) one accept state infinitely often.

This is only possible if we can find a loop containing the accept state from the initial state.
We can find such “accepting” loops by nested depth-first search:

Do DFS from the initial state until an accepting state

Do DFS from the accepting state to see if we can reach the state again
Checking LTL (5)

\( F \phi \)

LTL formula → Büchi automaton → State space → Product automaton → NDFS → Accepting cycle

Model
LTL Checking in ASAP
We return a model (generator/transition relation) rather than an automaton.

LTL Checking in ASAP
We return a model (generator/transition relation) rather than an automaton.

LTL Checking in ASAP
Example: Dining Philosophers

\[ p = \text{philosopher 1 eats} \]

Philosopher 1 will always eat at some point

- \( \varphi'' = F \ p \)
- \( \varphi''' = G \ F \ p \)
Example: Dining Philosophers

\[ p = \text{philosopher 1 eats} \]

Philosopher 1 will always eat at some point

- \( \phi'' = Fp \)
- \( \phi''' = GFp \)

How do we reasonably enter formulas in the tool?
Entering Formulas

- We already said that our atomic propositions are functions.
- It is possible to build a complex data structure representing the formulas and APs as functions.
- ...which we do internally and immediately and hide from users :-)

Demo: LTL

- Show JoSEL task
- Create formulas $F \ p$ and $G \ F \ p$
- Mapping can be reused
- Show error traces
- (Draw Büchi automaton)