Efficient Model Checking for \textit{Circus} Using FDR

26th May 2015
Introduction

Aims

- Introduce Model Checking
- Talk about efficient Model Checking for Circus...

Topics

1. Model Checking Overview
2. FDR Introduction
3. Circus Introduction
4. Model Checking Problems

Thanks

- Alvaro Miyazawa
- Tom Gibson-Robinson (Oxford)
Model Checking

What is Model Checking?

- Technique for verifying concurrent systems
- Determines if $M$ is a model for a formula $f$
  - $M$ is a transition system
  - $f$ is a temporal logic specification
- That is, $M$ exhibits whatever property $f$ captures
- In traditional Model Checking, $M$ and $f$ are written in different languages
In temporal logic a formula has dynamic truth.
A formula can be true for some of a system's states...
But false for others.

Usually has operators to specify...
What happens next
What happens sometime in the future
Something that happens for all future states
And logic operators like negation, conjunction and disjunction
Model Checking

Transition Systems

- Model a system using...
  - States (static structure)
  - Transitions (dynamic structure)
- Finite transition systems can be expressed as directed graphs

\[ \text{Diagram: } s_0 \rightarrow s_1 \rightarrow s_2 \]
Model Checking

Advantages

- No need to write proofs...
  - Although the model and the specification are needed
- Automatic...
  - After the model and specification are written
- Concurrency errors...
  - Difficult to reproduce with testing
- Fast...
  - Compared to other rigorous methods, like proof checking
- Counterexamples...
  - Invaluable for debugging
Efficient Model Checking

Model Checking

Disadvantages

- Finite state...
  - Automation generally requires finite state systems
- State Explosion...
  - Can be a big problem, most of the problems we address are due to this
Failures-Divergences Refinement

FDR

- Model Checking tool...
  - Although it’s really a refinement checker
- Model and the specification are the same language...
  - CSPm, the machine-readable version of CSP
- Performing a check...
  - Compile both CSP processes into Labelled Transition Systems
  - Check if one is a refinement of the other
 Failures-Divergences Refinement

**Refinement**
- Is a model an implementation of a specification?
- $P \sqsubseteq Q$ if every behaviour of $Q$ is also a behaviour of $P$
  - $Q$ exhibits the property captured by $P$
  - $Q$ implements $P$

**Semantic Models**
- Traces refinement $\sqsubseteq_T \ldots$
  - $P \sqsubseteq_T Q$ if every trace of $Q$ is a trace of $P$
- Failures Refinement $\sqsubseteq_F \ldots$
  - Failure: a trace leading to events that may be refused
- Failures-Divergences Refinement $\sqsubseteq_{FD} \ldots$
  - Divergence: a trace leading to undefined behaviour
Circus and CSPm Introduction

Circus

- Combines...
  - CSP to capture Behaviour
  - Z to capture Data
- Variants...
  - Object Orientation – Oh Circus
  - Time – CircusTime
- Model Checking in FDR...
  - Requires translation to CSPm
  - Different capabilities...
Circus and CSPm Introduction

<table>
<thead>
<tr>
<th>CSPm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events...</td>
</tr>
<tr>
<td>- Model important points in the history of the system</td>
</tr>
<tr>
<td>- May communicate parameters (c!x or c?x or simply c.x)</td>
</tr>
<tr>
<td>- Parameters can be restricted to certain values (c?x:set)</td>
</tr>
<tr>
<td>- Sequenced using prefix (c -&gt; P)</td>
</tr>
<tr>
<td>- May be guarded by a predicate ((guard)&amp; c -&gt; P)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processes in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>- External choice (P [] Q)</td>
</tr>
<tr>
<td>- Interleaving (P</td>
</tr>
<tr>
<td>- Parallel (P [</td>
</tr>
<tr>
<td>- Sequence (P ; Q)</td>
</tr>
</tbody>
</table>

Replication (e.g. ||| s : set @ P(s) )
Model Checking Problems

Problem Classes

- **Data...**
  - FDR struggles with...
    - Large data-types
    - Processes holding several unrelated parameters of large data-types
    - Infinite types

- **Structure...**
  - FDR struggles with...
    - Large numbers of states
    - Large orderings of processes from parallelisms
Data Problems

- *Circus* has variables but CSP does not...
  - To represent variables in CSP we use processes to control variables
  - e.g. $P(\text{num})$ to control the variable `num`
- FDR evaluates and compiles all possible combinations of states
- Strive to reduce the state space in the model...
Data Problems

Reducing State Space

- Bound large types
  - For example \( \{0..50\} \)

- Use direct function calls to pass large parameters, if possible...
  - Uses the functional language elements of CSPm
    - e.g instead of accumulating the set with
      \[ P(set) = \text{add?x} \rightarrow P(\text{union}(set, x)) \]
    - Call \( P(\{1..100\}) \) to directly pass the set

- Refactor processes that control large state variables into many smaller processes in interleaving...
Data Example One

Use Case

- Model contains a set of data of type value
- We can put values in and take values out
- Attempting to put the same value in twice is not possible
Process Controlling a Set: Bad

\[ P_1(\text{set}) = \]
\[ \text{in?}x: \text{set} \rightarrow P_1(\text{set}) \]
\[ [] \]
\[ \text{in?}x: \text{diff}(\text{value}, \text{set}) \rightarrow P_1(\text{union}(\text{set}, x)) \]
\[ [] \]
\[ \text{out?}x: \text{set} \rightarrow P_1(\text{diff}(\text{set}, x)) \]
Data Example One

Process Controlling a Set: Good

\[ P_2 = \|\| x : \text{value} \odot \text{NotMember}(x) \]

\text{NotMember}(x) =
\begin{align*}
in.x \rightarrow \text{Member}(x)
\end{align*}

\text{Member}(x) =
\begin{align*}
in.x \rightarrow \text{Member}(x) \\
[] \\
\text{out}.x \rightarrow \text{NotMember}(x)
\end{align*}
Data Example One

Comparison

- Since both processes can only add the same number to the set once, they are equivalent...
  - $P_1 \sqsubseteq_T P_2$ and $P_2 \sqsubseteq_T P_1$
  - $P_1 \sqsubseteq_F P_2$ and $P_2 \sqsubseteq_F P_1$
  - $P_1 \sqsubseteq_{FD} P_2$ and $P_2 \sqsubseteq_{FD} P_1$
## Efficient Model Checking

### Data Example One

#### Deadlock Freedom Check

<table>
<thead>
<tr>
<th></th>
<th>value=</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{0..5}</td>
<td>{0..10}</td>
<td>{0..15}</td>
<td>{0..20}</td>
</tr>
<tr>
<td>$P_1$</td>
<td>Compiled</td>
<td>0.02s</td>
<td>0.69s</td>
<td>39.57s</td>
</tr>
<tr>
<td></td>
<td>Checked</td>
<td>0.04</td>
<td>0.02s</td>
<td>0.12s</td>
</tr>
<tr>
<td>$P_2$</td>
<td>Compiled</td>
<td>0.01s</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>Checked</td>
<td>0.06s</td>
<td>0.08s</td>
<td>0.23s</td>
</tr>
</tbody>
</table>

$^1$ excluding $0.01s$ for $P_2$ Compiled

$\sim 2$ hours
Efficient Model Checking

Data Example One

Explanation

- $P_1$ takes longer to compile than $P_2$ but $P_2$ takes longer to check.
- There is a trade-off between the compilation time of a set and the checking time of many interleaved actions.
  - Checking phase can be more easily parallelised.
Data Example Two

Use Case

- Model contains a sequence representing a stack
- We can push and pop a value, and check the top element
- We cannot push more than a max number of values
- We cannot pop a value or check the top value of an empty stack
Data Example Two

Process Controlling a Sequence: Bad

\[ Q_1(\text{sequence}) = \]
\[(\#\text{sequence} < \text{maxStackId}) \land \]
\[ \text{push?x} \rightarrow \]
\[ Q_1(<x>\text{^sequence}) \]

\[
[\]
(\text{not null(sequence)}) \land
(\)
\[ \text{pop} \rightarrow \]
\[ Q_1(\text{tail(sequence)}) \]

\[
[\]
\text{top!head(sequence)} \rightarrow
\]
\[ Q_1(\text{sequence}) \]
Data Example Two

Process Controlling a Sequence: Good

\[ Q_2 = (\| i : \text{value} @ [\text{AlphaFree}(i)] \text{Free}(i) ) \{ l \text{ resume} \} \]
Efficient Model Checking

Data Example Two

Process Controlling a Sequence: Good

Free(id) = push.id?v -> Full(id, v)

Full(id, v) =
    pop.id-> (if id > minStackId then
                resume.id-1 -> Free(id)
        else
                Free(id))
[]
(id < maxStackId) &
    push.id+1?_ -> resume.id -> Full(id, v)
[]
top.id!v -> Full(id, v)
Since this is only a change of structure (once we apply renaming to $Q_2$) these processes are equivalent...

- $Q_1 \sqsubseteq_T Q_2$ and $Q_2 \sqsubseteq_T Q_1$
- $Q_1 \sqsubseteq_F Q_2$ and $Q_2 \sqsubseteq_F Q_1$
- $Q_1 \sqsubseteq_{FD} Q_2$ and $Q_2 \sqsubseteq_{FD} Q_1$
## Data Example Two

### Deadlock Freedom check

<table>
<thead>
<tr>
<th></th>
<th>value=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{0..5}</td>
</tr>
<tr>
<td></td>
<td>{0..6}</td>
</tr>
<tr>
<td></td>
<td>{0..7}</td>
</tr>
<tr>
<td></td>
<td>{0..8}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Q₁</th>
<th></th>
<th></th>
<th></th>
<th>Q₂</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compiled</td>
<td>0.82s</td>
<td>13.92s</td>
<td>417.83s</td>
<td>21462.05s</td>
<td>Checked</td>
<td>0.02s</td>
<td>0.07s</td>
</tr>
<tr>
<td></td>
<td>Checked</td>
<td>0.02s</td>
<td>0.07s</td>
<td>0.99s</td>
<td>N/A³</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

² ≈ 5.9 hours
³FDR crashed...
Data Example Two

Explanation

- $Q_1$ compiles slower due to checking the length of the sequence.
- $Q_2$ compiles faster because each process only controls one variable.
- Trade-off in terms of checking...
  - $Q_1$ controls a sequence, but is sequential.
  - $Q_2$’s sub-processes only control one variable each but being are parallel.
Process Structure Problems

**Large Possible Orderings of Behaviour**
- FDR explores all the possible states
  - Before any hiding or parallel restrictions can occur
- This can cause FDR to use a lot of memory
Use Case
- Waiting for many subordinate processes to signal readiness...
- Which may happen in any order
For a process...
R(x) = ready.x -> SKIP

Simple Replicated Interleave...
Interleave = ||| x : value @ R(x)

... Becomes Replicated Sequential Composition
Sequential = ; x : seq(value) @ R(x)
Comparison

- $\text{Interleave} \sqsubseteq_T \text{Sequential}$ but not the other way around...
  - Because $\text{Interleave}$ can perform $\text{ready}.x$ events in any order, whereas in $\text{Sequential}$ the order is fixed
- Refinement does not hold in any other semantic models
- $\text{Sequential}$ can be said to implement $\text{Interleave}$
### Deadlock Freedom check

<table>
<thead>
<tr>
<th>Intr</th>
<th>value=</th>
<th>0..15</th>
<th>0..20</th>
<th>0..25</th>
<th>0..30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiled</td>
<td></td>
<td>0.01s</td>
<td>0.01s</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td>Checked</td>
<td></td>
<td>0.17s</td>
<td>1.99s</td>
<td>77.34s</td>
<td>3455.14s$^4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Seq</th>
<th>value=</th>
<th>0..15</th>
<th>0..20</th>
<th>0..25</th>
<th>0..30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiled</td>
<td></td>
<td>0.00s</td>
<td>0.00s</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td>Checked</td>
<td></td>
<td>0.05s</td>
<td>0.05s</td>
<td>0.04s</td>
<td>0.05s</td>
</tr>
</tbody>
</table>

$^4 \sim 57.5 \text{ minutes}$
**Explanation**

- \( R(x) \) is a simple process so both versions are fast
- *Interleave* is slower to check because the events may happen in any order
- *Sequential* is much quicker because FDR chooses an arbitrary order for the events to happen in
- However, these processes are not equivalent
Summary

Model Checking

- Verification technique useful for concurrent systems
- Usually automatic and quick
- However, has some general limitations
  - State explosion being the most common
Summary

Efficient Model Checking in FDR

- Problems can occur when using data or parallelism yield a large number of state
- Solutions...
  - Use the functional elements of CSPm
  - Refactor processes controlling variables
  - Possibly refactor interleavings
- Mileage may vary...