Model Checking High Level Petri Net Specifications with Helena

Sami Evangelista
evangeli@cnam.fr

Cedric - CNAM Paris

Séminaire MeFoSyLoMa, le 14 octobre 2005
Outline

Background and Motivations

An overview of Helena

State representation in Helena

An example: the load balancing system

Benchmarks

Conclusions and perspectives
Outline

Background and Motivations

An overview of Helena

State representation in Helena

An example: the load balancing system

Benchmarks

Conclusions and perspectives
The Quasar project

- project started in 2002
- Quasar is a platform for the verification of concurrent programs written in Ada
- Quasar performs two main tasks
  - automatic abstraction (slicing) of the code with respect to a given property
  - automatic translation of the code to a colored Petri net
- The model checking part is left to a third part tool
- To be able to verify complex Ada software, we need a model checker which can
  - enable a straightforward and automatic translation of concurrent software to high level Petri nets
  - handle large state vectors of programs
Outline

Background and Motivations

An overview of Helena

State representation in Helena

An example: the load balancing system

Benchmarks

Conclusions and perspectives
What is Helena?

- Helena is a High LEvel Net Analyzer.
- Helena can verify deadlock freeness and state properties on-the-fly.
- It is written in portable Ada and freely available under the term of the GPL.
- Helena provides
  - a specification language to describe high level nets
  - a specification language to describe properties
  - model checking techniques to verify properties on the fly

Net description → Helena → Property

阻挡 Example

Yes

No

Counter-Example
Main features

- Transitions agglomerations

And also...

- High level data types
- Possibility to define high level functions written in an Ada like syntax
- Probabilistic verification methods (bitstate hashing / supertrace and hash compact methods)
- Interfaces with other tools: Lola, Prod, Tina (via unfolding)
Features

Main features

- Transitions agglomerations
- Code generation to speed up the analysis
Features

Main features

- Transitions agglomerations
- Code generation to speed up the analysis
- Stubborn sets computation
Features

Main features

▶ Transitions agglomerations
▶ Code generation to speed up the analysis
▶ Stubborn sets computation
▶ Compact state representation
Features

Main features

- Transitions agglomerations
- Code generation to speed up the analysis
- Stubborn sets computation
- Compact state representation
- A fast simulation mode with an efficient firing rule

And also...

- High level data types
- Possibility to define high level functions written in an Ada like syntax
- Probabilistic verification methods (bitstate hashing / supertrace and hash compact methods)
- Interfaces with other tools: Lola, Prod, Tina (via unfolding)
Features

Main features

- Transitions agglomerations
- Code generation to speed up the analysis
- Stubborn sets computation
- Compact state representation
- A fast simulation mode with an efficient firing rule

And also...

- High level data types
Features

Main features

- Transitions agglomerations
- Code generation to speed up the analysis
- Stubborn sets computation
- Compact state representation
- A fast simulation mode with an efficient firing rule

And also...

- High level data types
- Possibility to define high level functions written in an Ada like syntax
Features

Main features

- Transitions agglomerations
- Code generation to speed up the analysis
- Stubborn sets computation
- Compact state representation
- A fast simulation mode with an efficient firing rule

And also...

- High level data types
- Possibility to define high level functions written in an Ada like syntax
- Probabilistic verification methods (bitstate hashing / supertrace and hash compact methods)
Features

Main features

▶ Transitions agglomerations
▶ Code generation to speed up the analysis
▶ Stubborn sets computation
▶ Compact state representation
▶ A fast simulation mode with an efficient firing rule

And also...

▶ High level data types
▶ Possibility to define high level functions written in an Ada like syntax
▶ Probabilistic verification methods (bitstate hashing / supertrace and hash compact methods)
▶ Interfaces with other tools : Lola, Prod, Tina (via unfolding)
Outline

Background and Motivations

An overview of Helena

State representation in Helena

An example: the load balancing system

Benchmarks

Conclusions and perspectives
In most formalisms, e.g., Petri nets, the transition relation is a deterministic mechanism
⇒ each state $s$ can be directly encoded as couple $(pred, t)$ where
  ▶ $pred$ is a pointer to one of the predecessors of $s$ in the hash table
  ▶ $t$ is the transition such that $next(pred, t) = s$
States are stored in the hash table explicitly or symbolically
  ▶ explicitly: the whole state descriptor is inserted into the hash table
  ▶ symbolically: only the couple $(pred, t)$ is stored
Markings stored symbolically are called $\Delta$-markings

storing a couple $(pred, t)$ instead of the whole state descriptor can lead to important memory savings
The encoding scheme proposed is:

- non ambiguous: the transition relation is deterministic
- but also non canonical: a state may have several predecessors

Comparing a state \( s \) with an encoded representation \( (\text{pred}, t) \) becomes more difficult.

Solution: follow the pointers to predecessors until a state stored explicitly is found and execute the transitions sequence to retrieve the actual representation of the couple \( (\text{pred}, t) \).

This mechanism will be called a state reconstitution, and the transitions sequences will be called a reconstituting sequence.

Checking whether or not a state \( s \) is already in the state space can be considerably slower.
How to limit the time overhead introduced by the method?

- Observation: the computation time introduced directly depends on the lengths of the reconstituting sequences
- To place an upper bound on this length we use the underlying idea of the *stratified caching* strategy:
  Some strata of states are stored explicitly while others are stored symbolically
- We introduce a parameter $k_\delta$
  States met at a depth $d$ such that $d \mod k_\delta = 0$ are stored explicitly.
  Others are stored symbolically
  $\Rightarrow$ the length of a reconstituting sequence is at most $k_\delta - 1$
State space representation

Example of a state space with $k_\delta = 3$
1\textsuperscript{st} optimization: updating $\Delta$-markings predecessor

Idea: update the predecessor of a $\Delta$-marking when a shorter path to an explicit marking is found

\[ m_n \to t_{n-1} \to m_2 \to t_2 \to t_1 \to m_1 \]

\[ m \to t \to t_{n-1} \to m_n \]

↑ reconstitution of $m_n$ and markings of $S$ will be fasten
2nd optimization: backward firing of the reconstituting sequence

- Comparing a marking $m$ with a marking $m'$ encoded symbolically as $(pred, t)$ requires two costly operations:
  - the decoding of an explicit marking $e$
  - the firing of the reconstituting sequence $s$ to retrieve the actual value of $m'$

then the comparison of $m$ and $m'$ becomes trivial

- Idea: these two costly operations can be avoided by performing a backward firing, i.e., an unfiring, of $s$ on $m$

- Let $s = s_1.t.s_2$. If, after the unfiring of $s_2$ on $m$ we reach a marking $m''$ such that $t$ cannot be unfired on $m''$ we can stop the reconstitution since $next(e, s) \neq m$ and therefore $m' \neq m$

- Otherwise, if the unfiring of $s$ on $m$ leads to marking $e$ then $m$ and $m'$ correspond to the same marking

- can avoid useless reconstitutions
Outline

Background and Motivations

An overview of Helena

State representation in Helena

An example: the load balancing system

Benchmarks

Conclusions and perspectives
System specification

We want to specify a system with
- a set of $C$ clients which send requests to servers
- a set of $S$ servers which treat the requests of clients
- a load balancer which
  - routes the requests of clients towards the appropriate server, i.e., the least loaded
  - rebalances the loads of servers when needed
The clients - Algorithm

- send a request to servers
- wait for the answer
- go back to the idle state
The clients - Specification in Helena

constant int C := 7;
type Cid : range 1 .. C;

place Cidle {dom:Cid; init:for(c in Cid) <(c)>;}
place Cwait {dom:Cid;}
place Crequest {dom:Cid;}
place Cack {dom:Cid;}

transition Csend {
  in { Cidle:<(c)>; }
  out { Cwait:<(c)>; Crequest:<(c)>; }
}

transition Creceive {
  in { Cwait:<(c)>; Cack:<(c)>; }
  out { Cidle:<(c)>; }
}
The servers - Algorithm

- wait for a client request and accept it
- notify it to the load balancer and wait for its acknowledgment
- treat the request and acknowledge the client
The servers - Specification in Helena

\[ \text{constant int } S := 2; \]
\[ \text{type } \text{Sid : range 1 .. S;} \]

\[ \text{place } \text{Sidle} \{ \text{dom: Sid;}
\text{init: for(s in Sid) } <(s)>;\} \]
\[ \text{place } \text{Swait} \{ \text{dom: Sid*Cid ;}\} \]
\[ \text{place } \text{Sprocess} \{ \text{dom: Sid*Cid ;}\} \]
\[ \text{place } \text{Snotification} \{ \text{dom: Sid ;}\} \]
\[ \text{place } \text{Snotification_ack} \{ \text{dom: Sid ;}\} \]
\[ \text{place } \text{Srequest} \{ \text{dom: Cid*Sid ;}\} \]
The servers - Specification in Helena

transition S notify {
    in { S idle:<(s)>; S request:<(c,s)>; } 
    out { S waiting:<(s,c)>; S notification:<(s)>; }
}

transition S receive {
    in { S waiting:<(s,c)>; S notification_ack:<(s)>; } 
    out { S processing:<(s,c)>; }
}

transition S send {
    in { S processing:<(s,c)>; } 
    out { S idle:<(s)>; S cack:<(c)>; }
}
The load balancer - Routing algorithm

- wait for a client request
- choose the least loaded server
- route the request to this server
The load balancer - Load distribution algorithm

- wait for a server to accept a client request
- acknowledge the server
- if the loads are not balanced then remove a request for the most loaded server and give it to the least loaded server
The load balancer - Specification in Helena

type Cno : range 0 .. C;
type load : vector [Sid] of Cno;
constant load empty_load := [0];

// return the least load server's
function least(load l) -> Sid {
    Sid result := Sid 'first;
    for(i in Sid)
        if(l[i] < l[result])
            result := i;
    return result;
}
The load balancer - Specification in Helena

\texttt{transition L\_receive\_server \{ \\
\quad \text{in} \ {\{ \text{Lidle:\langle(l)\rangle}; \ \text{Snotification:\langle(s)\rangle}; \}} \\
\quad \text{out} \ {\{ \text{Lbalancing:\langle(incr(l,s))\rangle}; \\
\quad \quad \text{Snotification\_ack:\langle(s)\rangle}; \}} \\
\text{\texttt{transition L\_no\_balance \{ \\
\quad \text{in} \ {\{ \text{Lbalancing:\langle(l)\rangle}; \}} \\
\quad \text{out} \ {\{ \text{Lidle:\langle(l)\rangle}; \}} \\
\quad \text{guard : is\_balanced(l);} \}} \\
\texttt{transition L\_balance \{ \\
\quad \text{in} \ {\{ \text{Lbalancing:\langle(l)\rangle}; \text{Srequest:\langle(c, least(l))\rangle}; \}} \\
\quad \text{out} \ {\{ \text{Lidle:\langle(incr(decr(l,most(l)), least(l)))\rangle}; \\
\quad \quad \text{Srequest:\langle(c, most(l))\rangle}; \}} \\
\quad \text{guard : not is\_balanced(l);} \}}
Property specification

There is no deadlock state.

```
reject deadlock;
```

The requests are uniformly distributed upon the servers.

```
reject not (
// the load balancer is balancing the requests
 card(Lbalancing) = 1 or

// the difference between the number of requests
// for two servers s1 and s2 is at most 1
forall (s1 in Sid, s2 in Sid : s1 = s2 or
      diff(card(Srequest sr : sr->2 = s1),
           card(Srequest sr : sr->2 = s2)) <= 1));
```
Outline

Background and Motivations

An overview of Helena

State representation in Helena

An example: the load balancing system

Benchmarks

Conclusions and perspectives
Results obtained for the load balancing system

Property verified: requests are uniformly distributed upon the servers.

<table>
<thead>
<tr>
<th>C S</th>
<th>Initial net</th>
<th>Reduced net</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Arcs</td>
</tr>
<tr>
<td>4 2</td>
<td>13 776</td>
<td>46 977</td>
</tr>
<tr>
<td>5 2</td>
<td>99 061</td>
<td>393 253</td>
</tr>
<tr>
<td>6 2</td>
<td>673 814</td>
<td>3 031 863</td>
</tr>
<tr>
<td>7 2</td>
<td>4 397 196</td>
<td>22 023 767</td>
</tr>
<tr>
<td>4 3</td>
<td>43 806</td>
<td>155 673</td>
</tr>
<tr>
<td>5 3</td>
<td>409 581</td>
<td>1 698 438</td>
</tr>
<tr>
<td>6 3</td>
<td>3 766 968</td>
<td>17 604 621</td>
</tr>
<tr>
<td>7 3</td>
<td>32 056 569</td>
<td>165 557 136</td>
</tr>
</tbody>
</table>
Maria vs Helena on some academic models

<table>
<thead>
<tr>
<th>Model</th>
<th>States</th>
<th>Maria</th>
<th>Helena</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T</td>
<td>M</td>
<td>T</td>
</tr>
<tr>
<td>Dbm</td>
<td>2 125 765</td>
<td>932 sec.</td>
<td>410 sec.</td>
</tr>
<tr>
<td>Dining</td>
<td>4 126 351</td>
<td>341 sec.</td>
<td>151 sec.</td>
</tr>
<tr>
<td>Eratos</td>
<td>2 028 969</td>
<td>116 sec.</td>
<td>63 sec.</td>
</tr>
<tr>
<td>Lamport</td>
<td>1 914 784</td>
<td>96 sec.</td>
<td>46 sec.</td>
</tr>
<tr>
<td>Leader</td>
<td>1 518 111</td>
<td>150 sec.</td>
<td>70 sec.</td>
</tr>
<tr>
<td>Peterson</td>
<td>3 407 946</td>
<td>134 sec.</td>
<td>57 sec.</td>
</tr>
<tr>
<td>Slotted</td>
<td>3 294 720</td>
<td>197 sec.</td>
<td>99 sec.</td>
</tr>
</tbody>
</table>

410 sec. 328.42 Mo
151 sec. 65.95 Mo
63 sec. 90.33 Mo
46 sec. 32.04 Mo
70 sec. 32.38 Mo
57 sec. 41.93 Mo
99 sec. 41.57 Mo
Results obtained for an Ada client / server program

Property verified : absence of deadlock.

<table>
<thead>
<tr>
<th>No comp.</th>
<th>Collapse</th>
<th>Δ</th>
<th>Δ + Collapse</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>4 clients, 10 running tasks, 34 731 states</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>9.45</td>
<td>1.37</td>
<td>1.42</td>
</tr>
<tr>
<td>T</td>
<td>00:00:02</td>
<td>00:00:03</td>
<td>00:00:03</td>
</tr>
<tr>
<td>V</td>
<td>285.17</td>
<td>41.26</td>
<td>42.71</td>
</tr>
<tr>
<td><strong>5 clients, 12 running tasks, 635 463 states</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>205.63</td>
<td>28.37</td>
<td>21.94</td>
</tr>
<tr>
<td>T</td>
<td>00:00:51</td>
<td>00:01:54</td>
<td>00:01:44</td>
</tr>
<tr>
<td>V</td>
<td>339.31</td>
<td>46.82</td>
<td>36.20</td>
</tr>
<tr>
<td><strong>6 clients, 14 running tasks, 13 805 931 states</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>-</td>
<td>684.41</td>
<td>962.489</td>
</tr>
<tr>
<td>T</td>
<td>-</td>
<td>00:26:04</td>
<td>00:44:20</td>
</tr>
<tr>
<td>V</td>
<td>-</td>
<td>51.98</td>
<td>73.10</td>
</tr>
</tbody>
</table>
Results obtained for an Ada implementation of the sieves of Eratosthene

Property verified: absence of deadlock.

<table>
<thead>
<tr>
<th>No comp.</th>
<th>Collapse</th>
<th>Δ</th>
<th>Δ + Collapse</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N=20, 9 running tasks, 3,599,634 states</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>698.74</td>
<td>214.51</td>
<td>100.72</td>
</tr>
<tr>
<td>T</td>
<td>00:07:10</td>
<td>00:08:05</td>
<td>00:12:05</td>
</tr>
<tr>
<td>V</td>
<td>203.54</td>
<td>62.49</td>
<td>29.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>37.28</td>
</tr>
<tr>
<td><strong>N=25, 10 running tasks, 24,884,738 states</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>-</td>
<td>-</td>
<td>676.24</td>
</tr>
<tr>
<td>T</td>
<td>-</td>
<td>-</td>
<td>01:53:50</td>
</tr>
<tr>
<td>V</td>
<td>-</td>
<td>-</td>
<td>28.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>260.289</td>
</tr>
<tr>
<td><strong>N=30, 11 running tasks, 96,566,610 states</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>T</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>V</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1,026.89</td>
</tr>
</tbody>
</table>

Model Checking High Level Petri Net Specifications with Helena
Sami Evangelista (Cedric - CNAM)
Outline

Background and Motivations

An overview of Helena

State representation in Helena

An example: the load balancing system

Benchmarks

Conclusions and perspectives
Conclusions and Perspectives

Conclusions

Helena is an explicit model checker for high level Petri nets which
▶ targets software specification model checking
▶ enables to define high level data types and functions
▶ is particularly efficient in terms of memory (it can handle state spaces with $10^8$ states)
▶ tackles the state explosion problem by the use of structural abstraction techniques and partial order methods

Perspectives

▶ implementation of extended agglomerations
▶ integrate a LTL model checking module (possibly through an interface with the SPOT library)
▶ support of the Petri Net Markup Language