Model Checking High Level Petri Net Specifications with Helena

Sami Evangelista
evangeli@cnam.fr

Cedric - CNAM Paris

Séminaire MeFoSyLoMa, le 14 octobre 2005

Outline

Outline

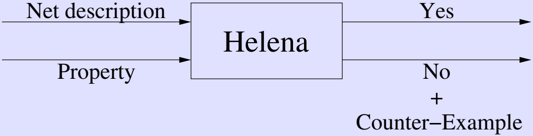
The Quasar project

- 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

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.
- Downloadable at http://helena.cnam.fr
- 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



Features

Main features

- Transitions agglomerations

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

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

General idea (1)

- In most formalisms, e.g., Petri nets, the transition relation is a deterministic mechanism
  $\Rightarrow$ 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 Δ-markings

↗ storing a couple $(pred, t)$ instead of the whole state descriptor can lead to important memory savings

General idea (2)

- 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 $(pred, t)$ 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 $(pred, t)$
- 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

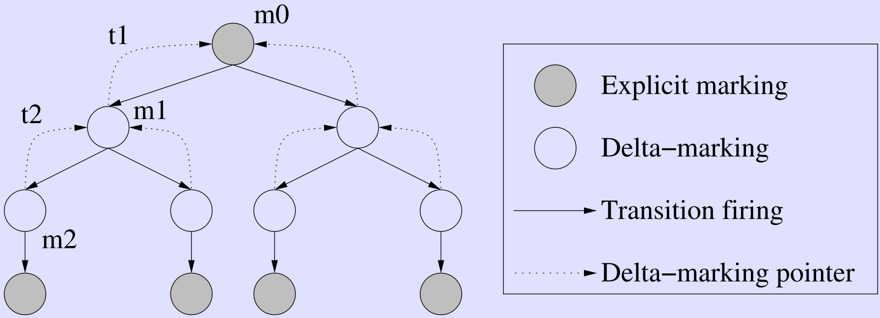
How to limit the time overhead introduced by the method?

- 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_δ
  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 $k_\delta - 1$

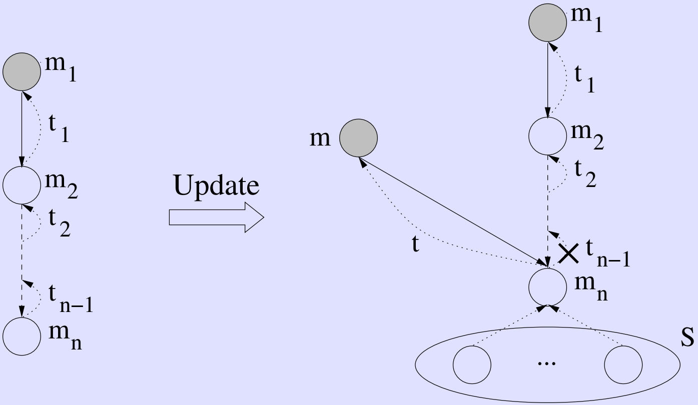
State space representation

Example of a state space with $k_\delta = 3$



1^{st} optimization: updating Δ-markings predecessor

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



↗ reconstitution of m_n and markings of S will be fasten

2^{nd} 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

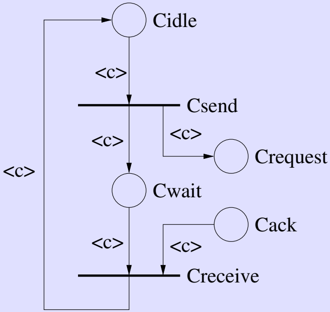
System specification

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

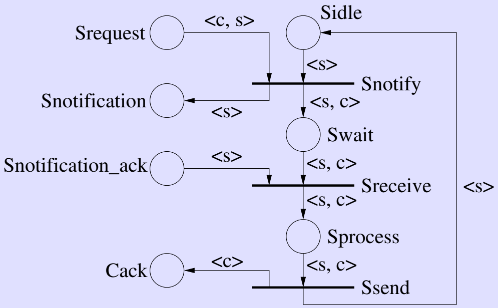


The clients - Specification in Helena

```
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

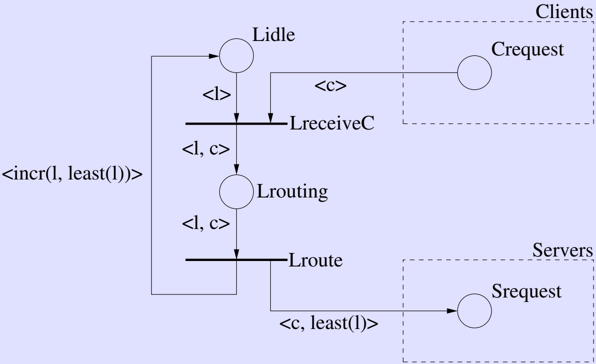
```
constant int S := 2;
type Sid : range 1 .. S;

place Sidle                {dom:Sid;
                            init:for(s in Sid) <(s)>;}
place Swait                {dom:Sid*Cid;}
place Sprocess             {dom:Sid*Cid;}
place Snotification        {dom:Sid;}
place Snotification_ack    {dom:Sid;}
place Srequest             {dom:Cid*Sid;}
```

The servers - Specification in Helena

```
transition Snotify {
  in  { Sidle:<(s)>; Srequest:<(c,s)>; }
  out { Swaiting:<(s,c)>; Snotification:<(s)>;}}
transition Sreceive {
  in  { Swaiting:<(s,c)>; Snotification_ack:<(s)>;}
  out { Sprocessing:<(s,c)>;}}
transition Ssend {
  in  { Sprocessing:<(s,c)>;}
  out { Sidle:<(s)>; Cack:<(c)>;}}
```

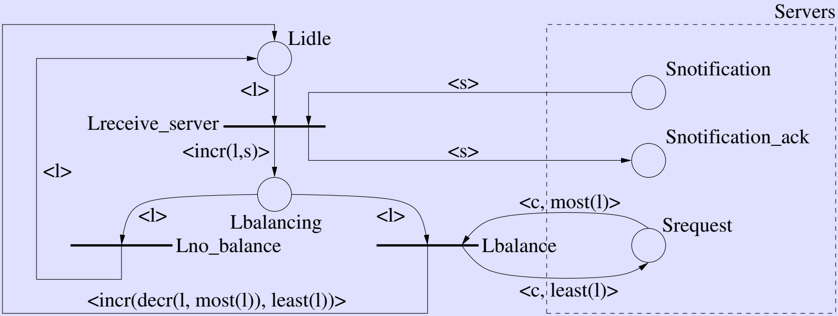
The load balancer - Routing algorithm

- 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

```
transition Lreceive_server {
  in  { Lidle:<(l)>; Snotification:<(s)>;}
  out { Lbalancing:<(incr(l,s))>;
        Snotification_ack:<(s)>;}
transition Lno_balance {
  in  { Lbalancing:<(l)>;}
  out { Lidle:<(l)>;}
  guard : is_balanced(l);}
transition Lbalance {
  in  { Lbalancing:<(l)>; Srequest:<(c,least(l))>;}
  out { Lidle:<(incr(decr(l,most(l)),least(l)))>;
        Srequest:<(c,most(l))>;}
  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

Results obtained for the load balancing system

Property verified : requests are uniformly distributed upon the servers.

		Initial net			Reduced net		
C	S	States	Arcs	Time	States	Arcs	Time
4	2	13 776	46 977	0	6 420	22 533	0
5	2	99 061	393 253	1	41 456	171 128	1
6	2	673 814	3 031 863	13	260 744	1 230 207	5
7	2	4 397 196	22 023 767	104	1 574 530	8 344 591	36
4	3	43 806	155 673	0	16 938	61 569	0
5	3	409 581	1 698 438	6	139 836	595 638	3
6	3	3 766 968	17 604 621	85	1 232 262	5 897 781	28
7	3	32 056 569	165 557 136	5 465	9 613 008	51 203 400	295

Maria vs Helena on some academic models

Model	States	Maria		Helena	
		T	M	T	M
Dbm	2 125 765	932 sec.	296.30 Mo	410 sec.	328.42 Mo
Dining	4 126 351	341 sec.	56.63 Mo	151 sec.	65.95 Mo
Eratos	2 028 969	116 sec.	80.13 Mo	63 sec.	90.33 Mo
Lamport	1 914 784	96 sec.	26.56 Mo	46 sec.	32.04 Mo
Leader	1 518 111	150 sec.	28.68 Mo	70 sec.	32.38 Mo
Peterson	3 407 946	134 sec.	35.06 Mo	57 sec.	41.93 Mo
Slotted	3 294 720	197 sec.	34.51 Mo	99 sec.	41.57 Mo

Results obtained for an Ada client / server program

Property verified : absence of deadlock.

	No comp.	Collapse	Δ	Δ + Collapse
4 clients, 10 running tasks, 34 731 states				
M	9.45	1.37	1.42	0.30
T	00:00:02	00:00:03	00:00:03	00:00:04
V	285.17	41.26	42.71	9.10
5 clients, 12 running tasks, 635 463 states				
M	205.63	28.37	21.94	4.98
T	00:00:51	00:01:54	00:01:44	00:01:52
V	339.31	46.82	36.20	8.21
6 clients, 14 running tasks, 13 805 931 states				
M	-	684.41	962.489	167.85
T	-	00:26:04	00:44:20	00:48:59
V	-	51.98	73.10	12.75

Results obtained for an Ada implementation of the sieves of Eratosthene

Property verified : absence of deadlock.

	No comp.	Collapse	Δ	Δ + Collapse
N=20, 9 running tasks, 3 599 634 states				
M	698.74	214.51	100.72	37.28
T	00:07:10	00:08:05	00:12:05	00:14:11
V	203.54	62.49	29.34	10.86
N=25, 10 running tasks, 24 884 738 states				
M	-	-	676.24	260.289
T	-	-	01:53:50	02:07:02
V	-	-	28.49	10.97
N=30, 11 running tasks, 96 566 610 states				
M	-	-	-	1 026.89
T	-	-	-	12:25:40
V	-	-	-	11.15

Outline

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