High Level Petri Nets Analysis with Helena

Sami Evangelista

CEDRIC - CNAM Paris,
292, rue St Martin, 75003 Paris
evangeli@cnam.fr

Abstract. This paper presents the high level Petri nets analyzer Helena. Helena can be used for the on-the-fly verification of state properties, i.e., properties that must hold in all the reachable states of the system, and deadlock freeness. Some features of Helena make it particularly efficient in terms of memory management. Structural abstractions techniques, mainly transitions agglomerations, are used to tackle the state explosion problem. Benchmarks are presented which compare our tool to Maria.

Helena is developed in portable Ada and is freely available under the conditions of the GNU General Public License.

1 Introduction

Model checking is an automatic method for the verification of finite state systems. It consists of enumerating all the possible configurations or executions of the system to track the ones which do not match the specification.

Helena (a High LEvel Net Analyzer) is a model checker developed at the CNAM university in Paris. The formalism supported by Helena makes it suitable for the verification of concurrent software. Helena is part of the Quasar project (a tool for the analysis of concurrent Ada programs [1]), but it is also a fully autonomous tool which can be used independently. In its current version, Helena can be used for the verification of state properties, i.e., properties that must hold in all the reachable states of the system, and deadlock freeness. Helena is a command-line oriented tool without any graphical user interface though we consider to include a graphical interactive simulator.

This paper is structured as follows. Section 2 explains the reasons which motivated us to design and implement Helena. The main features of Helena are presented in Section 3. A set of benchmarks which compare our tool to Maria [2] are presented in Section 4. Finally we conclude in Section 5.

2 Motivations

The model checking of concurrent programs involves a translation task from the original programming language to a formalism suitable for the expression of concurrency, e.g., Promela, Petri nets. In order to limit the state explosion

G. Ciardo and P. Darondeau (Eds.): ICATPN 2005, LNCS 3536, pp. 455–464, 2005.

problem of a model checking procedure, the produced model has to remain as small as possible, but still must be an exact translation of the program to guarantee a correct verification process. Quasar is a tool which aims at the automatic verification of concurrent Ada programs based on colored Petri nets. It can not currently handle the whole language, though a large part of the language which is related to concurrency is supported. The main task of Quasar is to perform automatic abstraction (or slicing) of a program and to translate it into a colored Petri net. In its current state, Quasar does not perform reachability analysis but interfaces with another tool which realizes this task. The high level formalism supported by Maria [2] pushed us into choosing this tool, though Quasar can also interface with Prod [3].

After an intensive use of Quasar on various examples, we identified two main factors which limit the efficiency of our tool. Firstly, the translation from the source code to the colored Petri net is not straightforward. As the constructions of high level programming language do not have their exact counterpart in colored Petri nets, even in the Maria formalism, the translation step introduces additional transitions which generate intermediate states that are not relevant for the verification purpose. Secondly, the state vectors exhibited by models obtained from the translation of a program are usually quite large as software make heavy use of structured data types. Thus, though optimized to represent multiplicity in a compact way [4], the encoding scheme of Maria fails to represent efficiently state vectors with large color domains, and many tokens in places.

Helena has been designed from this previous experience to meet these two requirements : enable a straightforward and automatic translation of concurrent programs without resort to the introduction of many useless transitions and intermediate states, and handle state spaces with large state vectors.

3 Main Features of Helena

High Level Description Language. The class of high level Petri nets used in Helena was primarily designed to enable the simulation and verification of concurrent Ada programs. To achieve this, we naturally decided to include in Helena the possibility to define high level data types. There are four categories of data types : numerical type, enumerate type, structured type, and vector, i.e., array type. Another feature of Helena is the possibility for the user to define complex functions written in a pseudo-C syntax. These functions may then be used in arc expressions. Such a possibility allows to automatically translate sub programs and sequences of statements which do not include synchronizations into a single transition, provided that the input programming language can easily be mapped into Helena functions.

Compilation and Execution of the Model. A well known and efficient approach to reduce the execution time of a model checking procedure is to compile the model into a source code which will correspond to the actual reachability analyzer. Compiling and executing the model has numerous advantages over an interpretation of the model. Mainly, the evaluation of the expressions in arc mappings can be drastically fasten. Tools that use this technique include Prod, Spin [5], and Maria. It has been shown in [6] that this technique greatly reduces the execution time even for small models for which we may think that the compilation of the generated code is a too severe overhead (which does not exist if the model is interpreted).

Helena models are also translated into executable code. For performance and portability issues we chose the language C. In order to ease the readability of the generated code, and the debugging of the compilation process, this code is commented, nicely formatted, and divided into several libraries.

Enabling Test Algorithm. Verification and simulation tools based on high level Petri nets spend a significant amount of time in determining under which assignments (or bindings) a high level transition is firable. This non trivial problem is known under the term of enabling test. The enabling test algorithm implemented in Helena has been described in [7]. It basically includes two main components.

1. We exploit the locality principle of Petri nets which states that the firing of a transition only affects the status (enabled/disabled) of its neighbor transitions. Thus, a depth first search algorithm can benefit from this locality principle by
   (a) maintaining a set of enabled transitions
   (b) updating this set when a transition instance is fired by only inspecting the neighbor transitions of the fired instance.

   Our implementation of this locality principle is based on the definitions of structural conflict and causality [8]. The basic idea is to translate these relations into equivalent constraints systems [9] before the search algorithm. During the search, the systems built are solved in order to identify disabled and enabled bindings. We illustrate our purpose with an example. Let us consider the net of figure 1.

   From the structure of the net and the arc expressions we can deduce that the firing of an instance $(t, \langle X_t \rangle)$ disables the firing of an instance $(u, \langle X_y, Y_u \rangle)$ if the following constraint is verified : $[X_t = X_u \wedge Y_u = 0]$. During the search, at each firing of an instance of transition t we instantiate this system with the firing binding to identify the instances of u which are disabled. We also observe that the firing of an instance $(t, \langle X_t \rangle)$ enables the firing of an

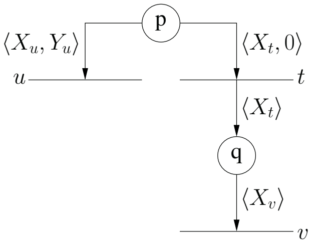

Fig. 1. Translating arc expressions into constraints system

instance $(v, \langle X_v \rangle)$ if the following holds : $[X_v = X_t]$. Consequently, the firing of an instance $(t, \langle X_t \rangle)$ causes the insertion of the instance $(v, \langle X_t \rangle)$ into the set of enabled transitions without any additional check.

2. We showed in [7] that this approach is unfortunately not always sufficient to determine valid transitions assignments at the new marking. In most cases, this must be followed by an unification algorithm. The algorithm we proposed is an improvement of Mäkelä's unification algorithm [10].

State Space Storage. Model checking tools usually represent states in two different forms :

- an expanded form which is convenient for the implementation of the transition relation
- a compressed form in which expanded states are encoded before their insertion into the state space in order to save memory

An interesting feature of Helena is the way it represents compressed states. The method, called Δ-markings method, has been described in [11]. The underlying idea is to store a large set of states in a non explicit way by only storing references on others states. Figure 2 illustrates the principle of the method. Markings met at a depth d such that $d \mod k_\delta = 0$ (with k_δ a user defined parameter) are stored explicitely whereas other markings are represented in the state space by a couple $(pred, (t, c_t))$, where $pred$ is the "address" in the hash table of one of the predecessors of the marking and (t, c_t) is the transition instance which firing leads from the predecessor to the marking. Retrieving the actual value of a marking from its "Δ encoding" can simply be done by following the links which point on the predecessors until a marking stored explicitely is reached. The marking is then obtained by firing the sequence of instances (t, c_t) which label the links followed to reach the explicit marking. For instance, the actual value of marking m can be retrieved by first backtracking to m' and then to m_0 and to apply on it the firing of sequence $(t, c_t).(t', c_{t'})$. For models exhibiting large state vectors, the compression ratios observed are quite impressive, whereas the method becomes less interesting for small models. The price to pay is an acceptable increase of the run time. Both the compression ratio and the run time increase can be influenced by parameter k_δ.

The state collapsing [12] method of Spin is also implemented in Helena. This method is based on the observation that even if the set of syntactical possible values for a token in a place is huge (let us denote it size by m), the set of values really met during the search (which size is denoted by n) is in practice usually much smaller. This is so because the types of the places and transitions of the net are usually over-approximations made by the user of the possible values really met during the search.

With the collapse method a token is represented with $log_2(n)$ bits. This "collapsed token" is in fact an index of a table of size n which stores all the "true tokens" already met during the search on $log_2(m)$ bits. This table is initially empty, and filled during the search by the token values met. Thus, we make use of the two following functions to query the tokens table:

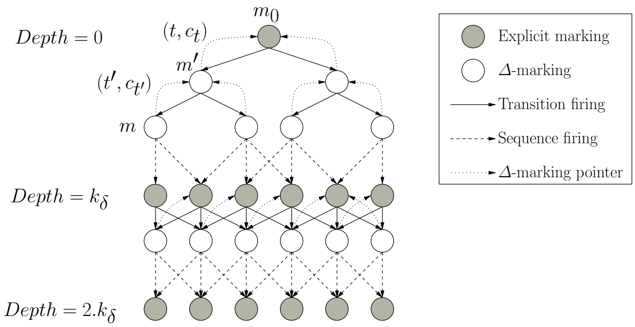

Fig. 2. Illustration of the Δ-markings method

- $index\_of(token\ t) \rightarrow int$ looks for token t into the tokens table and returns the index at which the token has been found. It may also add an entry in the table if the token is not already in it.
- $token\_at(int\ i) \rightarrow token$ returns the token at index i in the tokens table.

Since n can not be known a priori, Helena can detect overflows in this table, i.e., cases where the number of different tokens met is greater than n, and report it to the user who, in turn, can increase it and rerun the search. This restriction may seem quite bothering, but after intensive experiments, we observed that overflows are usually quickly detected during the search, causing only a small run time overhead.

Structural Abstraction Techniques. The efficiency of the explicit model checking approach is seriously limited by the state explosion problem inherent to the concurrent execution of several components. An efficient way to tackle this problem is to perform structural abstractions on the initial model in order to obtain a simpler model which is equivalent to the initial one for to the specified property. For Petri nets and Colored Petri nets, transitions agglomerations are surely the most efficient abstraction techniques. They have been defined by Berthelot in his doctoral thesis [13], and generalized by Haddad to Colored Petri nets in [14]. Transition agglomerations merge consecutive transitions into a virtual atomic one which effect is the composition of the effects of these transitions. It results in a drastic reduction of the combinatory explosion due to the elimination of some intermediate states. In addition, the complexity of these transformations is linear with respect to the size of the net, and their application is totally automatic.

These agglomerations are implemented in Helena. This implementation is based on the symbolic calculus of some structural relations which ensure correct agglomerations in the unfolded net. Though this symbolic computation is not always possible, it usually works fine. To our best knowledge, Helena is currently the only tool that implement structural agglomerations for high level Petri nets.

Stack Representation. Model checking algorithms based on a depth first search, e.g., LTL model checking, store the set of non fully processed states in a stack. For some models, this stack can grow very large, and include almost all the states of the state space. For instance, models for which the state space forms a single strongly connected component usually follow this behavior. Thus, an awkward choice for the representation of this stack can lead to important memory wastes.

In Helena, the stack is represented as a vector of bits, and it uses a nice property of Petri nets : the transition relation is a reversible mechanism, that is, given a marking m and a transition t, there is a single marking m' such that $m'[t\rangle m$. This marking m' is defined by $\forall p \in P, m'(p) = m(p) + W^{-}(p,t) - W^{+}(p,t)$. Instead of representing the search stack as a stack of states we therefore chose to represent it as the sequence of transition bindings which leads from the initial marking to the current marking. When a state has been fully expanded, the binding on top of the stack is "unfired", and the next enabled binding is processed. This representation can naturally be combined with the collapse method to store transition bindings of the stack more compactly.

Expressing State Properties. The query language of Helena is rich enough to express a wide range of state properties. It is based on four basic operations :

- The **count** operation allows to count the number of distinct items in a place which fulfill a given condition.
- The **mult** operation allows to count the cumulated multiplicities of distinct items in a place which fulfill a given condition.
- The **exists_token** operation allows to check that at least one token in a place fulfills a given condition.
- The **forall_token** operation allows to check that all the tokens present in a place fulfills a given condition.

For instance, the property

```
forall_token(p in P : exists_token(q in Q : q->1 = p->2))
```

could be stated in an informal way as : for each token `p` held in place `P` there is a token `q` in place `Q` which is such that the first component of `q` and the second component of `p` are equal.

4 Benchmarks

We have compared our tool with Maria to study the performances of both tools concerning time and memory consumption. We considered these six examples :

- The distributed database system
- The slotted ring protocol
- The dining philosophers

- The sieves of Eratosthene to find prime numbers
- The leader election protocol of Chang and Roberts
- The Peterson algorithm for the mutual exclusion problem

All these examples can be found in the Helena distribution (directory `samples`). Experiments were done on a Pentium 4 with 2.5Ghz and a main memory of 1Gb. To allow a fair comparison, Maria was invoked with option `--compile` which allows to compile arc expressions into a dynamically linkable library. In addition, no capacity constraint was indicated in the Maria model and structural agglomerations were disabled in Helena in order to obtain the same numbers of markings and arcs.

The results of our experiments are reported in table 1. For each model, row T reports the times observed for the exploration of the state space in seconds, row S reports the size of the state space in kilobytes, and row V reports the average size of the state vector in bytes. Compilation times of the models are included. Helena was invoked with several values of parameter k_δ within the set $\{1, 5, 10, 20, 50\}$. Let us recall that k_δ is the parameter of the compression method implemented in Helena. The higher this parameter is, the more compressed the state space

Table 1. Results of the Comparison of Helena with Maria

	Maria	**Helena**				
		$k_\delta = 1$	$k_\delta = 5$	$k_\delta = 10$	$k_\delta = 20$	$k_\delta = 50$
The distributed database system, N=12, 2 125 765 states						
T	900	452	492	533	778	782
S	303 409	336 330	80 111	52 496	16 493	15 997
V	146.15	162.01	38.59	25.29	7.95	7.71
The dining philosophers, N=12, 4 126 351 states						
T	360	306	425	474	549	647
S	57 992	67 947	35 679	31 587	29 527	28 290
V	14.39	16.86	8.85	7.84	7.33	7.02
The sieves of Eratosthene, N=40, 2 028 969 states						
T	116	90	121	151	206	368
S	82 056	92 868	35 274	28 046	24 432	22 294
V	41.41	46.87	17.80	14.15	12.33	11.25
The leader election protocol, N=14, 1 518 111 states						
T	155	103	141	181	264	354
S	29 367	33 258	15 930	13 667	12 786	11 616
V	19.81	22.43	10.75	9.22	8.62	7.84
The Peterson algorithm, N=4, 3 407 946 states						
T	136	91	133	154	188	315
S	35 899	43 366	28 287	26 511	25 630	25 094
V	10.78	13.03	8.50	7.97	7.70	7.54
The slotted ring protocol, N=8, 3 294 720 states						
T	214	164	245	276	325	401
S	37 622	42 760	26 370	23 456	23 342	22 730
V	11.39	13.29	8.20	7.57	7.25	7.06

will be, and the more slow the search will be. If $k_\delta = 1$ then no compression is performed. In addition, the state collapsing method was not helpful for these simple models and therefore disabled.

We observe that the results obtained by Helena without any compression technique are comparable to the ones obtained by Maria. The encoding scheme of Maria gives a slight advantage to Maria in terms of memory usage, while Helena usually explores the state space in less time. When using the Δ-markings method, Helena outperforms its competitor concerning the size of the state space, especially for models with large state vectors (the distributed database system, and the sieves of Eratosthene). The best compression ratio is obtained for the distributed database system, with $k_\delta = 50$. For this case, the size of the state space is divided by almost 19. The drawback is an increase of the execution time. This one is acceptable for low values of k_δ but tends to grow with it. For models with small vectors, e.g., the slotted ring protocol, our storage method is clearly less interesting though a reduction factor of 2, which is the average reduction factor observed for these models, can still be helpful for systems with large numbers of states. Finally, we observe that with our storage method, the average size of the state vector could be reduced to approximatively 10 bytes, whatever the model is.

5 Conclusions and Perspectives

The validation of software is a difficult problem. Few tools based on high level Petri nets have been designed to face this challenge as most of them focus on the representation of control, e.g. tools based on Well Formed Petri nets such as GreatSPN [15], over data. Thanks to multiple features, Helena can handle the validation of software systems.

The possibility to define high level data types and functions enables to model concurrent software written in high level programming languages such as Ada in a succinct way.

The Δ-markings method [11] is implemented in Helena. This one is particularly efficient when dealing with state spaces with large state vectors. This is why we believe that it should be adapted within the scope of the verification of software as these make heavy use of structured data types and usually exhibit large state vectors.

Lastly, structural abstractions help to tackle the state explosion problem.

We plan to extend Helena in the following ways :

- At the current implementation stage, Helena supports the verification of state properties and the deadlock freeness. We plan to extend the field of properties that Helena can verify by including a module for the verification of linear time temporal logic properties (LTL). An interface with the extensible library SPOT [16] is currently under study. The main interest of using SPOT is that it relies on transition-based generalized Büchi automata and allows translation of LTL formula to smaller automata and thus, smaller synchronized products.

- The current version of Helena tackles the state explosion problem by the use of structural abstraction techniques. We envisage to combine these with the stubborn set method of Valmari [17, 18, 19]. However, defining an algorithm which computes good stubborn sets for colored Petri nets is a difficult task. To our best knowledge, the only tool that compute stubborn sets for colored Petri nets (without unfolding) is CPN tools [20]. The algorithm has been described in [19].
- New transitions agglomeration rules have been recently defined for ordinary and colored Petri nets in [21, 22]. An implementation of these agglomerations in Helena is scheduled.

Availability. Helena is a free software available under the conditions of the GNU General Public License. It can be downloaded at the following URL : `http://helena.cnam.fr`.

References

1. Evangelista S., Kaiser C., Pradat-Peyre J.F., and Rousseau P. Quasar : a new tool for analyzing concurrent programs. In *Proceedings of the Ada-Europe International Conference on Reliable software technologies*, volume 2655 of *LNCS*, pages 166–181. Springer, 2003.
2. Mäkelä M. Maria : modular reachability analyser for algebraic system nets. In *Proceedings of the 23th International Conference on Application and Theory of Petri Nets*, volume 2360 of *LNCS*, pages 434–444. Springer, 2002.
3. Varpaaniemi K. PROD 3.4.00 — an advanced tool for efficient reachability analysis. Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland, 2004. Software.
4. Marko Mäkelä. Condensed storage of multi-set sequences. In *Practical Use of High-Level Petri Nets*, number 547 in DAIMI report PB, pages 111–125. University of Århus, Denmark, 2000.
5. Holzmann G.J. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
6. Mäkelä M. Applying compiler techniques to reachability analysis of high-level models. In *Workshop on Concurrency, Specification & Programming 2000*, number 140 in Informatik-Bericht, pages 129–142. Humboldt-Universität zu Berlin, Germany, 2000.
7. Evangelista S. and Pradat-Peyre J.F. An efficient algorithm for the enabling test of colored petri nets. In *Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, number 570 in DAIMI report PB, pages 137–156. University of Århus, Denmark, 2004.
8. Dutheillet C. and Haddad S. Conflict sets in colored petri nets. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models*, pages 76–85, 1993.
9. Evangelista S. Syntactical rules for colored petri nets manipulation. Technical Report 641, Cedric, CNAM, http://cedric.cnam.fr, 2004.
10. Mäkelä M. Optimising enabling tests and unfoldings of algebraic system nets. In *Proceedings of the 22th International Conference on Application and Theory of Petri Nets*, volume 2075 of *LNCS*, pages 283–302. Springer, 2001.

11. Evangelista S. and Pradat-Peyre J.F. Efficient state space storage in explicit model checking. Technical Report 682, Cedric, CNAM, http://cedric.cnam.fr/, 2004.
12. Visser W. Memory efficient state storage in spin. In *Proceedings of the Second Spin Workshop*, 1996.
13. Berthelot G. Transformations and decompositions of nets. In *Advances in Petri Nets*, volume 254 of *LNCS*, pages 359–376. Springer, 1986.
14. Haddad S. A reduction theory for colored nets. In *Advances in Petri Nets*, volume 424 of *LNCS*, pages 399–425. Springer, 1989.
15. Chiola G., Franceschinis G., Gaeta R., and Ribaudo M. Greatspn 1.7: graphical editor and analyzer for timed and stochastic petri nets. *Performance Evaluation*, 24(1-2):47–68, 1995.
16. Alexandre Duret-Lutz and Denis Poitrenaud. Spot: an extensible model checking library using transition-based generalized Büchi automata. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'04)*, pages 76–83. IEEE Computer Society Press, 2004.
17. Valmari A. On-the-fly verification with stubborn sets. In *Proceedings of the 5th International Conference on Computer-Aided Verification*, volume 697 of *LNCS*, pages 397–408. Springer.
18. Valmari A. The state explosion problem. *Lectures on Petri Nets I : Basic Models*, 1491:429–528, 1998.
19. Lars Michael Kristensen and Antti Valmari. Finding stubborn sets of coloured petri nets without unfolding. In *Proceedings of the 19th International Conference on Application and Theory of Petri Nets*, volume 1420 of *LNCS*, pages 104–123. Springer, 1998.
20. Michel Beaudouin-Lafon, Wendy E. Mackay, Mads Jensen, Peter Andersen, Paul Janecek, Michael Lassen, Kasper Lund, Kjeld Mortensen, Stephanie Munck, Anne Ratzer, Katrine Ravn, Soren Christensen, and Kurt Jensen. CPN/tools: A tool for editing and simulating coloured petri nets. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 574–pp. Springer, 2001.
21. Haddad S. and Pradat-Peyre J.F. New powerfull Petri nets reductions. Technical report, Cedric, CNAM, http://cedric.cnam.fr/, 2003.
22. Evangelista S., Haddad S., and Pradat-Peyre J.F. New coloured reductions for software validation. In *Proccedings of the 7th International workshop on discrete event systems*, pages 355–360, 2004.