Parallel State Space Generation and Exploration on Shared-Memory Architectures

Milan Češka, Bohuslav Křena, and Tomáš Vojnar

Faculty of Information Technology, Brno University of Technology,
Božetěchova 2, 612 66 Brno, Czech Republic
{ceska, krena, vojnar}@fit.vutbr.cz

1 Introduction

Model checking is a technique for an automated formal verification of correctness of various classes of systems. Compared to the more traditional approaches to this problem based on simulation and testing, the main advantage of model checking is that the desirable behavioural properties of the considered system are checked in such a way that all the reachable states which may affect the given property are guaranteed to be covered. The disadvantage of model checking is that it may have very high computational time and space requirements. That is why various ways of dealing with the computational complexity of model checking are sought.

The systems to be verified may be described using various languages. Here, we consider the systems to be described by the PNtalk language based on Object-Oriented Petri Nets (OOPNs). PNtalk [6] has been developed at the Brno University of Technology as a tool suitable for modelling, prototyping and rapid development of concurrent and distributed software systems.

In the paper, we discuss possibilities of parallel OOPN state space generation and exploration on shared-memory architectures. The goal is to combat the high time complexity of state spaces-based verification methods.

2 Object-Oriented Petri Nets

The OOPN formalism is characterized by a Smalltalk-like object orientation enriched with concurrency and polymorphic transition execution, which allows for message sending, waiting for and accepting responses, creating new objects, and performing computations.

OOPNs are based on active objects whose internal activity is described by high-level Petri nets that are called object nets. The objects communicate via message sending. Asynchronous reactions of objects to incoming message are described by method nets. Method nets are also high-level Petri nets and each of them has a set of parameter places and a return place. Method nets can access places of the corresponding object net. Both object nets and method nets are dynamically instantiable. Objects can also communicate synchronously via synchronous ports that resemble special transitions of objects nets, which can be fired only when they are activated from some classical transition. A simple OOPN model in Fig. 1 demonstrates modelling by OOPNs on a simple example of dealing with the stack data structure.

The latest research in the area of OOPNs shows that due to their features, OOPNs are very suitable for describing open and reflective systems [7].

R. Moreno Díaz et al. (Eds.): EUROCAST 2005, LNCS 3643, pp. 275–280, 2005.

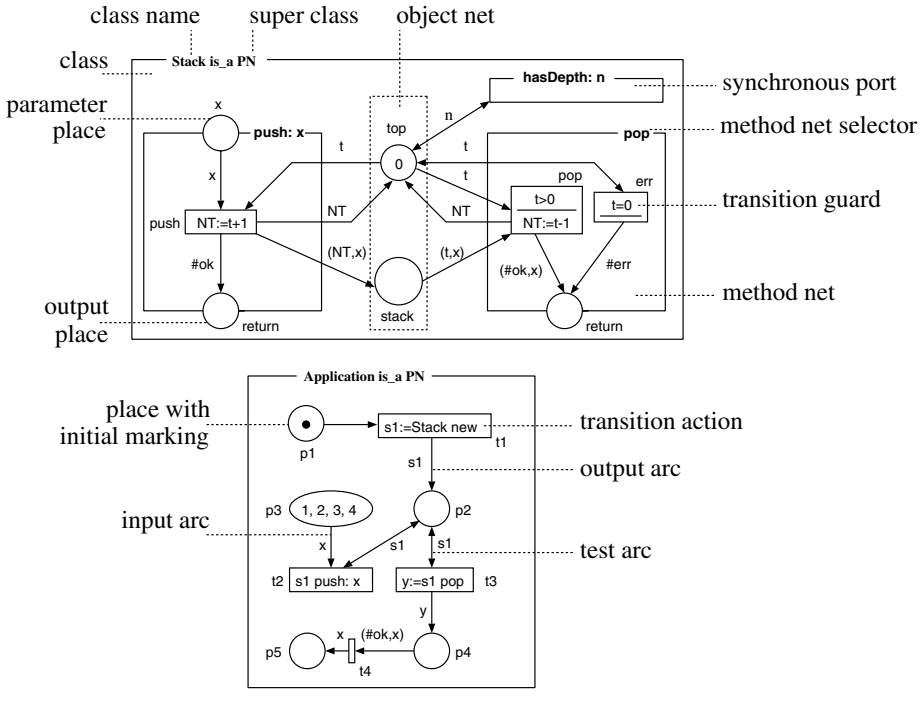

Fig. 1. A simple OOPN model

3 A Parallel OOPN State Space Generator

The main problem of the model checking methods is the so-called *state space explosion*—the number of states grows exponentially with the size of the model. Thus, it is difficult or practically impossible to apply these methods directly to large systems. We can identify two main approaches for dealing with the problem. The first class of these methods consists in sophisticated generation and exploration of the state spaces while the second one tries to exploit more powerful computer architectures. In the context of OOPNs, we have explored the first approach, e.g., in [4,9,5]. Here, we consider the second possibility. Namely, we discuss the techniques behind a parallel state space generator of OOPNs which we have developed for architectures with shared memory using the Java programming language and the JOMP tool [2].

The main task in programming parallel applications for shared-memory architectures is to achieve a good load balance among multiple computation threads taking into account their need to synchronize when accessing shared data. In our case, we need to deal with three shared data structures—(1) a hash table, which is used for storing the state space such that fast searching of states is possible when we need to know whether a certain state has already been found or not, (2) a list of non-explored states that contains states whose successors have not been explored yet, and (3) a marking pool which is a special pool for storing decomposed markings.

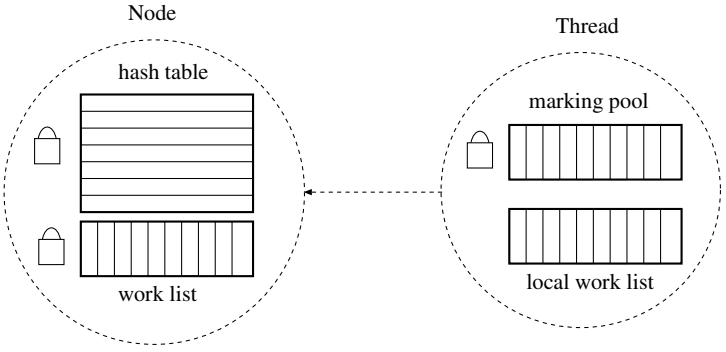

Fig. 2. Data Distribution

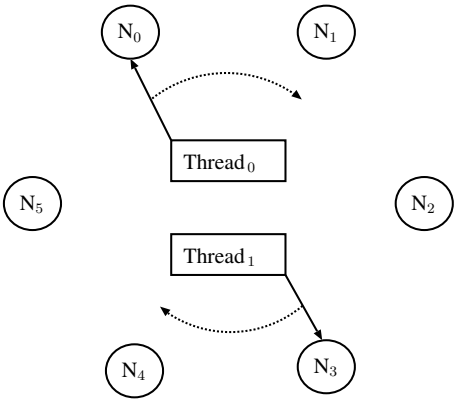

Fig. 3. Work Seeking

In order to minimize conflicts among threads trying to access the hash table, we have divided the table into several parts. In addition, we have divided also the list of non-explored states (also called as the work list) in such a way that a newly generated state is put to the unique part of the work list associated with the part of the hash table to which the state is stored. Moreover, we let the thread that is responsible for handling a certain part of the hash table use a private work list when processing the states to be explored. Finally, to further improve the performance, we create a copy of the marking pool for each thread. The distribution of data we thus obtain is shown in Fig. 2.

As we have already said, one of the crucial issues for efficiency of parallel applications is the quality of load balance among threads. We have started with one data node (i.e. a fraction of the hash table) for each thread. However, this did not work well. At first, we are not able to divide the hash table to parts with the same number of states in advance. Moreover, for a good load balance, it is necessary that each thread has some work (i.e. states to explore) during the whole production run, which is impossible to predict and guarantee in advance too.

Therefore, we have divided the hash table and the work list to more parts than we have threads. We find by experiments that the best performance is reached when three times more nodes than threads are created. Then, threads switch among nodes and try to find some work to do as it is shown in Fig. 3 for the case of two threads.

4 Experimental Results

In this section, we show parallel speedups achieved using all the optimizations discussed in the previous section. For our experiments and measurements, we used two servers, namely, a Sun Fire 15k server (52 processors UltraSPARC III, heartbeat 900 MHz, and 52 GB of memory, provided by the Edinburgh Parallel Computing Centre) and a Sun Enterprise 450 server (4 processors Sun UltraSPARC-II, heartbeat 400 MHz, and 4 GB of memory, provided by the Brno University of Technology).

In the following tables, we show average values for three successive production runs in order to statistically minimize the measuring error. The simple OOPN model called *life model* [8] was used in all the measurements discussed here.

The average execution times for the server Sun Fire 15k are listed in Tab. 1 while the corresponding speedups can be found in Tab. 2. The values in the left columns were obtained using the sequential garbage collector (SGC) while the values in the right ones were obtained using the parallel garbage collector (PGC). The average execution times for the server Sun Enterprise 450 are listed in Tab. 3 while the corresponding speedups can be found in Tab. 4.

Table 1. Average execution times for the Sun Fire 15k server

Threads	Number of Unique States			
	102 340		400 995	
	SGC	PGC	SGC	PGC
1	26,50 s		105,68 s	
2	20,08 s	20,85 s	78,73 s	90,07 s
4	13,61 s	13,50 s	59,53 s	58,04 s
8	9,82 s	9,20 s	49,50 s	38,88 s
12	9,21 s	8,80 s	42,76 s	32,52 s
16	8,77 s	8,31 s	44,57 s	32,30 s

Table 2. Average speedups for the Sun Fire 15k server

Threads	Number of Unique States			
	102 340		400 995	
	SGC	PGC	SGC	PGC
1	1,00		1,00	
2	1,32	1,27	1,34	1,17
4	1,95	1,96	1,78	1,82
8	2,70	2,88	2,14	2,72
12	2,88	3,01	2,47	3,25
16	3,02	3,19	2,37	3,27

Table 3. Average execution times for the Sun Enterprise 450 server

Threads	Number of Unique States				
	23 426	**176 851**	**585 276**	**1 373 701**	**2 667 126**
1	12,5 s	81,7 s	264,6 s	656,8 s	2 571 s (42 min 51 s)
2	8,1 s	51,0 s	180,0 s	436,8 s	1 874 s (31 min 14 s)
3	6,7 s	39,6 s	140,3 s	340,7 s	1 642 s (27 min 22 s)
4	6,6 s	35,8 s	126,3 s	304,5 s	1 517 s (25 min 17 s)

Table 4. Average speedups for the Sun Enterprise 450 server

Threads	Number of Unique States				
	23 426	**176 851**	**585 276**	**1 373 701**	**2 667 126**
1	1,00	1,00	1,00	1,00	1,00
2	1,54	1,60	1,47	1,50	1,37
3	1,87	2,06	1,89	1,92	1,57
4	1,89	2,28	2,10	2,16	1,69

Table 5. Load balance among threads (in number of states)

Threads	Thread 0	Thread 1	Thread 2	Thread 3
1	23 426	0	0	0
	88 401	0	0	0
2	11 871	11 555	0	0
	43 881	44 520	0	0
3	8 011	7 573	7 742	0
	29 685	28 702	30 014	0
4	5 946	6 501	5 078	5 901
	22 516	23 428	20 735	21 722

The parallel speedups that we have currently achieved are satisfactory despite we had expected a little bit better results (especially efficiency). However, the load balance among particular threads is nearly ideal. This is promising for the future development because the good load balance is a basic condition for achieving a good parallel speedup. We illustrate the quality of our algorithm regarding the load balance in Tab. 5 and Tab. 6 where the values for the Sun Enterprise 450 server and a small number of states are showed. An upper number in each cell corresponds with a number of unique states while the lower one corresponds with a number of generated states. We show here the values for the model with small number of states due to the load balance is even better when models with bigger number of states are considered.

5 Conclusions and Future Work

We have discussed possibilities of exploiting parallel architectures with shared memory for combating the high computational complexity of model checking. We have used Java and JOMP [2] as programming tools and considered OOPNs as a modelling formalism. Proposing a parallel algorithm with good load balance among threads is an important result of our work.

Table 6. Load balance among threads (in percentage)

Threads	Thread 0	Thread 1	Thread 2	Thread 3
1	100,0	0	0	0
	100,0	0	0	0
2	50,7	49,3	0	0
	49,6	50,4	0	0
3	34,2	32,3	33,5	0
	33,6	32,5	34,0	0
4	25,4	27,8	21,7	25,2
	25,5	26,5	23,5	24,6

We have also presented experimental results that we have achieved. To sum up them, we have reached parallel speedups up to 3.3 using 16 processors on the Sun Fire 15k server and speedups up to 2.3 using 4 processors on the Sun Enterprise 450 server.

The main problem that prevents us to achieve an even better parallel speedup is the memory management system (especially garbage collecting) due to the memory operations being performed sequentially. We have tried to exploit parallel garbage collectors too—the results are, however, not satisfactory. Another possible approach to this problem is to implement a user-specific memory management system like in [1] to reduce the number of dynamically created objects.

Acknowledgement. This work was supported by the Czech Grant Agency under the contracts 102/04/0780 and 102/03/D211 and it was also supported by the European Community within the "Access to Research Infrastructure Action of the Improving Human Potential Programme" under the contract HPRI-CT-1999-00026.

References

1. S. C. Allmaier and G. Horton. Parallel Shared-Memory State-Space Exploration in Stochastic Modelling. In Proc. of IRREGULAR '97, LNCS 1253, 1997. Springer.
2. M. Bull and M. E. Kambites. JOMP—an OpenMP-like Interface for Java. In Proc. of the ACM 2000 conference on Java Grande, 2000. ACM Press.
3. E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking. The MIT Press, Cambridge, Massachusetts, London, England, 2000.
4. M. Češka, V. Janoušek, and T. Vojnar. Towards Verifying Distributed Systems Using Object-Oriented Petri Nets. In Proc. of EUROCAST'99, LNCS 1798, 2000. Springer.
5. M. Češka, L. Haša, and T. Vojnar. Partial Order Reduction in Model Checking of Object-Oriented Petri Nets. In Proc. of EUROCAST'03, LNCS 2809, 2003. Springer.
6. V. Janoušek. Modelling Objects by Petri Nets. PhD. thesis, Brno University of Technology, Brno, Czech Republic, 1998.
7. B. Kočí. Methods and Tools for Implementing Open Simulation Systems. PhD. thesis, Brno University of Technology, Brno, Czech Republic, 2004.
8. B. Křena. Analysis Methods of Object Oriented Petri Nets. PhD. thesis, Brno University of Technology, Brno, Czech Republic, 2004.
9. T. Vojnar. Towards Formal Analysis and Verification over State Spaces of Object-Oriented Petri Nets. PhD. thesis, Brno University of Technology, Brno, Czech Republic, 2001.