dmcG: a distributed symbolic model checker based on GreatSPN

Alexandre Hamez, Fabrice Kordon, Yann Thierry-Mieg, and Fabrice Legond-Aubry

Université P. & M. Curie,
LIP6 - CNRS UMR 7606
4 Place Jussieu, 75252 Paris cedex 05, France
Alexandre.Hamez@lip6.fr, Fabrice.Kordon@lip6.fr, Yann.Thierry-Mieg@lip6.fr,
Fabrice.Legond-Aubry@lip6.fr

Abstract. We encountered some limits when using the GreatSPN model checker on life-size models, both in time and space complexity. Even when the exponential blow-up of state space size is adequately handled by the tool thanks to the use of a *canonization* function that allows to exploit system symmetries, time complexity becomes critical. Indeed the canonization procedure is computationally expensive, and verification time for a single property may exceed 2 days (without exhausting memory).
Using the GreatSPN model-checking core, we have built a distributed model-checker, dmcG, to benefit from the aggregated resources of a cluster. We built this distributed version using a flexible software architecture dedicated to *parallel and distributed* model-checking, thus allowing full reuse of GreatSPN source code at a low development cost. We report performances on several specifications that show we reach the theoretical linear speedup w.r.t. the number of nodes. Furthermore, through intensive use of multi-threading, performances on multi-processors architectures reach a speedup linear to the number of processors.

Keywords GreatSPN, Symbolic Reachability Graph, Distributed Model Checking

1 Introduction

If we want model checking to cope with industrial-size specifications, it is necessary to be able to handle large state spaces. Several techniques do help:

i compact encoding of a state space using decision diagrams; these techniques are called symbolic[1] model checking [3,1],
ii equivalence relation based representation of states that group numerous *concrete* states of a system into a *symbolic* state [2]; these techniques are also called *symbolic*,
iii executing the model checker on a distributed architecture [4,5,6,7,8].

[1] The word *symbolic* is associated with two different techniques. The first one is based on state space encoding and was introduced in [1]. The second one relies on set-based representations of states having similar structures and was introduced in [2].

These three techniques can be stacked. This was experimented for *(i)* and *(ii)* in [9].

GreatSPN [10] is a well known tool implementing technique *(ii)* in its model checking kernel thanks to the use of Symmetric Nets [2] for input specifications. We have successfully used it for many studies requiring to analyse complex systems with more than 10^{18} concrete states and an exponential gain between the concrete reachability graph and the symbolic reachability graph [11].

However, these studies and another one dedicated to intelligent transport systems [12] revealed two problems. First, generation of the symbolic reachability graph requires a canonical representation of states to detect already existing ones. This is a known problem that requires CPU time. So, even if memory consumption is reduced, CPU load becomes a problem.

The second problem deals with an implementation constant that prevented us to handle more than 12 million symbolic states in the version we had (in the first study we mentioned, this was the equivalent to 10^{18} concrete states).

So, to handle larger systems, the idea is to use the aggregated resources of a cluster and thus, merge techniques *(ii)* and *(iii)* by implementing a distributed model checker for Symmetric Nets able to generate faster larger state spaces.

This paper presents how we built dmcG, a distributed model checker based on the GreatSPN model-checking core. We did not change the GreatSPN sources files. They were encapsulated in a higher-level program called libdmc [13]. This library is dedicated to parallelization and distribution of model checkers and orchestrates services provided by GreatSPN to enable a distributed execution.

The paper is organized as follow. After a survey of related works in Section 2, Section 3 explains how we built dmcG atop libdmc using GreatSPN. Then, Section 4 presents performances of dmcG and discusses about them.

2 Related Work

Several attempts at proposing a distributed model-checker have been made. In [6] the authors implemented a distributed version of Spin. The problem however, was that the main state space reduction technique of Spin, called partial order reduction, had to be re-implemented in a manner that degrades its effectiveness as the number of hosts collaborating increases. Thus performances, reported up to 4 hosts in the original paper, were reported to actually not scale well on a cluster (see Nasa's case study in [14]). Another effort to implement a distributed Spin is DivSpin [4]. However, they chose to re-implement a Promela engine rather than using Spin's source code. As a result their sequential version is at least twice as slow as sequential Spin in it's most degraded setting with optimizations deactivated. And any further improvements of the Spin tool will not profit their implementation.

An effort that has met better success is reported in [5] for a distributed version of the Murphi verifier from Stanford. Murphi exhibits a costly canonization procedure. The original implementation in [5] was built on top of specific Berkeley NOW hardware[3],

[2] Formerly known as Well-Formed Nets [2], a class of High-level Petri Nets

[3] The Berkeley Network of Workstations

which limits its portability. A more recent implementation [7] is based on MPI [15], however it is limited to two threads per hosts, one handling the network and one for computation of the next state function. Our work is however comparable to that effort in terms of design goals: reuse of existing code over a network of multi-processor machines, a popular architecture due to its good performance/cost ratio. The good results reported by these Murphi-based tools with slightly sublinear speedup over the number of hosts encourage further experimentation in this direction.

We can also cite [8] which is an effort to distribute the generation of (but theoretically not limited to) LOTOS specifications. They report near linear speedups.

3 Building dmcG

Our strategy is to build a model checker based on the GreatSPN core, thus reusing its implementation of the symbolic reachability graph. So, we let GreatSPN compute successors but its execution is handled by the libdmc library. This one distributes the state space over the nodes of a cluster and manages all the networking and multi-threaded aspects of the state space generation.

To distribute *memory load*, we assign an *owner node* to each state, responsible for storing it. We use a static localization function that, for each state, designates a host. Note that this function should have a homogeneous distribution to ensure *load balancing*: we chose MD5 as it is known to provide a good distribution for any kind of input.

To distribute *CPU load*, each node has an active component that computes successors. This is the computationally most expensive task in the model-checking procedure, particularly because of the canonization algorithms of GreatSPN.

3.1 Architecture of dmcG

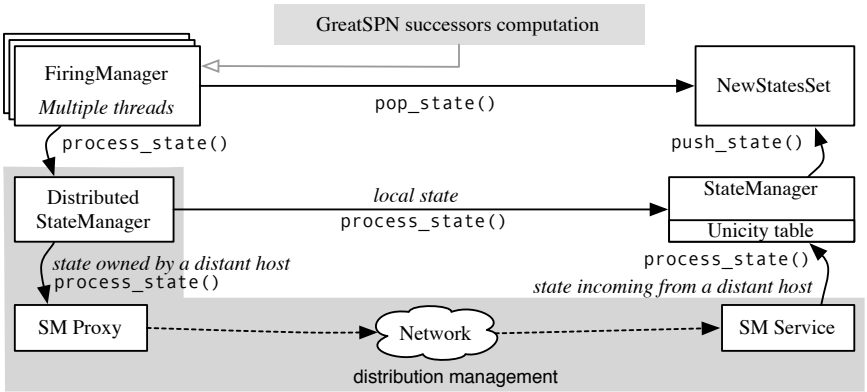

Fig. 1. dmcG architecture

dmcG is designed using a component based approach. Each node instantiates the components shown on figure 1:

1. The *StateManager* is a passive storage component instanciated once per node, that stores the states owned by the node. The behavior of *processState* is to determine whether a state is new or not. If a state is new, it is stored in an unicity table and also pushed into the *NewStatesSet*, otherwise, it is discarded.
2. The *NewStatesSet* is a protected passive state container, used to store states that have not been fully explored. Each node has a single occurrence of this component, shared by the active computing threads.
3. The *FiringManager* represents an active thread used to compute successors by calling GreatSPN successors functions. Several instances of this component are instanciated on each node to allow full use of multi-processor machines, and to overlap network latency. As shown in figure 1, each *FiringManager* instance repeatedly pops a state from the *NewStatesSet*, computes its successors and passes them to the *DistributedStateManager*. One privileged *FiringManager* initiates the computation by processing the initial state(s) on a master node.
4. The *DistributedStateManager* is responsible for forwarding the states to their owner, using the localization function. This component deals with a set of *StateManager Proxies* representing distant *StateManager* (*proxy* design pattern [16]). When a state is not owned by the node which computed it, it is transmitted to its owner through its proxy/service pair. Each node instantiates one *DistributedStateManager* component for localization purposes, one proxy per distant node, and one service to receive incoming states computed on other nodes.

3.2 Interaction with GreatSPN

The first step was to identify the core functions of GreatSPN which compute successors of a state. This work was made easier thanks to a previous similar effort aimed at integrating LTL model-checking capacity in GreatSPN, using the Spot library [17]. The interface we defined corresponds to a labeled automata: we extracted functions to obtain the initial state, and a successor function returning the set of successors of a given state (using an iterator). An additional labeling function was extracted to allow labeling of states satisfying a given criterion (deadlock state, state verifying some arbitrary state property...).

This simple interface is defined to minimize dependencies from libdmc to a given formalism. We did not redevelop any algorithms for successor generation in libdmc: this is essential to allow the reuse of existing efficient implementations of state space generators, and their usually quite complex data representation. The (existing) algorithms related to state representation algorithms are cleanly separated from the distribution related algorithms. The use of a canonization function by GreatSPN is thus transparent for libdmc. It manipulates compact state representations, in which each "symbolic state" actually represents an equivalence class of states of the concrete reachability graph.

The states are handled in an opaque manner by libdmc : they are seen as a simple block of contiguous memory, at interaction points between model-checking engine and libdmc, thus reducing dependencies between the model-checker and libdmc. This choice

of retaining a raw binary encoding of states allows a low overhead of the library, but forces to operate over a homogeneous hardware configuration so that all nodes have a common interpretation of the state data. In any case, a layer integrating machine independent state encodings (i.e. XDR) could be added, but has not been implemented.

libdmc is intensely multi-threaded to allow the better use of modern hardware architectures, thus posing some problems when integrating legacy C code such as GreatSPN. GreatSPN is inherently non-reentrant, due to numerous global variables. The solution adopted consists in compiling the tool as a shared library that can be loaded and dynamically linked into. Simply copying the resulting shared object file in different file locations allows to load it several times into different memory spaces. This is necessary as threads usually share memory space. Each successor computation thread is assigned a separate copy of the GreatSPN binary.

We should emphasize the fact that a complete rewriting of GreatSPN was not possible since it is made of complex algorithms stacked in an architecture that has more than ten years of existence. Furthermore It wouldn't have validated the fact that the libdmc can interact with legacy code.

3.3 Verification of safety properties

The verification of safety or reachability properties is an important task since many aspects of modeled systems can be checked by such properties, and it is a necessary basis to handle more complex temporal logics. Finding reachable states that verify or not a property is an easy task once the state space is generated. Each node simply examines the states it has stored. This provides a yes/no answer to reachability queries. However, to let users determine how errors happened in their specifications, we need to provide a witness trace (or 'counter-example') leading to the target state. We provide a minimal counter-example in order to simplify the debuging task for the designer.

During the construction of the reachable state set, arcs are not stored however. And unfortunately, no predecessor function is available in the GreatSPN core. The approach used is to store during the construction in each state its distance to the initial state along the shortest path possible. Due to nondeterminism, the first path found to a given state is not necessarily the shortest, thus the distance may be updated if a state is reached later by a shorter path. In such a case, we have to recompute successors of the state to update their own distance, etc... Since this scheme can be costly, to avoid its occurrence as much as possible, states are popped from each node's *NewStatesSet* in ascending distance. This scheme does not introduce additional synchronization among nodes, and helps to maintain an overall approximative BFS state exploration.

After the generation of the whole state space, we build an index on each node that orders states by depth. It's not a CPU intensive task since we only need to iterate once on the state space (it only takes a few seconds). The drawback is that the size of this index increases with state space depth.

Once this index is built a master node controls the construction of the counter-example which leads to a state s in error at a distance n. The master asks to each slave a predecessor of s at a distance $n-1$. Slaves compute this predecessor by iterating over all states at depth $n-1$, using the index. The first state whose successor is s is sent to the master and the iteration is stopped. Then the master stores the first received predecessor

in the counter-example. This operation is performed until the initial state has been found as predecessor. We then have a minimal counter-example.

4 Experimentations

Performances results have been measured on a cluster of 22 dual Xeons hyper-threaded at 2.8GHz, with 2GB of RAM and interconnected with Gigabit ethernet. We focused our evaluations on two parameters: states distribution and obtained speedups.

The following parametric specifications have been selected:

- the Dining Philosophers, a well known academic example; it is parameterized with the number of philosophers.
- a Client-Server specification with asynchronous send of messages and acknowledgments; it is parameterized with the number of clients and servers.
- the model of the PolyORB middleware kernel [11]; it is parameterized with the number of threads in the middleware. Let us note that the analysis of this model could not be achieved for more than 17 threads in sequential generation (it took more than 40 hours).

4.1 State distribution

A homogeneous distribution of the states over the hosts in the cluster is an important issue since it is related to load balancing of dmcG. The goal is to avoid the situation were some hosts are overloaded while others are idle. This is required to reach a linear speedup.

Therefore, we measured the number of states owned by each host, in order to validate the choice of MD5 as a localization function. We then compared these results to the theoretical mean value and noted the variation. These measures are summarized in Table 1. In this table, column 1 represents the parameter that scales up the model, column 2 the number of involved hosts, column 3 the total number of symbolic states, column 4 the theoretical mean value, column 5 the standard deviation and column 6 the standard deviation expressed in percentage of the mean value.

We obtain standard deviations that are from less than 0.1% to less than 5% of the mean values. Such results are obtained for every model we analyzed in any configuration (model parameters and number of nodes). So our expectations are met since the state space is evenly distributed over the whole cluster. However, we observe that, as a possible side effect of the MD5 checksum, it is preferable to have a number of nodes that is a power of 2, in which case the standard deviation is smaller than 1% of the mean value. Additional experiments showed that the usage of a more complex checksum like SHA-1 doesn't seem to improve or to decrease the quality of the states distribution.

4.2 Speedups

Figures 2, 3 and 4 show the compared time of sequential and distributed generation needed for the three specifications we analyzed. They also show the speedups we obtain

Parameter	Hosts	Symb. States	Mean	Std. deviation	Percentage
Dining Philosophers					
12 philosophers	4	347 337	86 834	427	0.5%
15 philosophers	4	12 545 925	3 136 481	792	< 0.1%
12 philosophers	22	347 337	15 788	689	4.4%
15 philosophers	22	12 545 925	570 269	24 179	4.2%
Client-Server					
100 processes	4	176 851	44 213	202	0.5%
400 processes	4	10 827 401	2 706 850	1327	< 0.1%
100 processes	20	176 851	8 843	316	3.57%
400 processes	20	10 827 401	541 370	17 438	3.22%
PolyORB middleware					
11 threads	16	3 366 471	210 404	402	0.2%
17 threads	16	12 055 899	753 494	1131	0.2%
11 threads	20	3 366 471	168 324	5393	3.2%
17 threads	20	12 055 899	602 795	19 557	3.2%
25 threads	20	37 623 267	1 881 163	60 540	3.7%

Table 1. States distribution for the Philosophers, Client-Server and PolyORB specifications

for the distributed generation of these models. Speedups are compared to the runtime of the standard GreatSPN tool running the same example.

dmcG has a low overhead: execution for the mono-threaded, single host version of dmcG takes within 95% to 105% of the time needed by the standard GreatSPN version (these variations are due to implementation details concerning the storage of states). The local but multi-threaded version is truly twice as fast on a dual-processor machine.

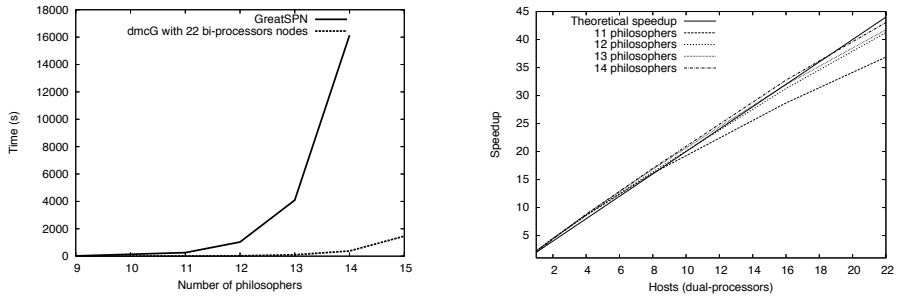

Fig. 2. Generation time (left) and speedups (right) for the Dining Philosophers specification

The main observation is that, in many cases, the observed speedup is over the theoretical one based on the number of *processors* (two per host): we have a supra-linear acceleration factor, up to 50 with 20 bi-processors nodes (fig. 3). We observed this in near all our experiments on several models with various parameters. We attribute this to

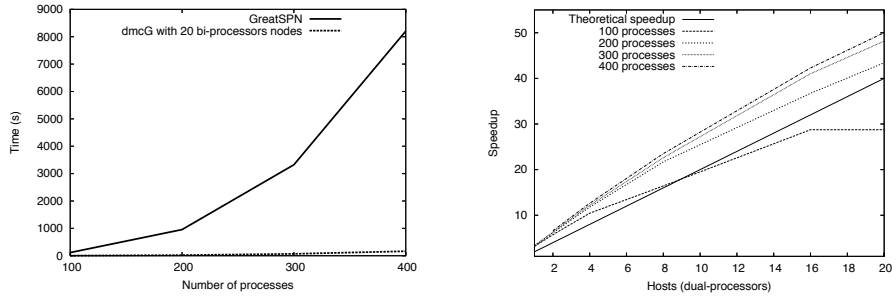

Fig. 3. Generation time (left) and speedups (right) for the Client-Server specification

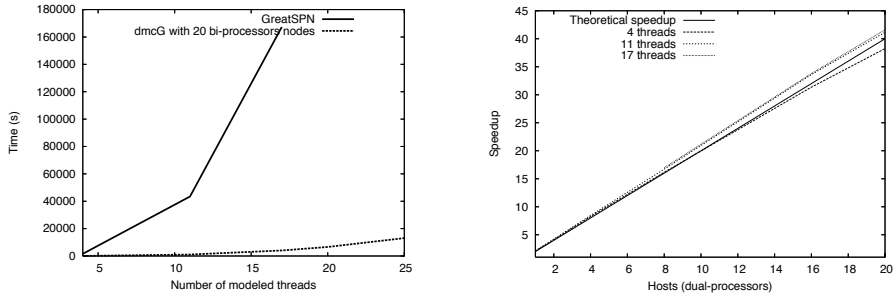

Fig. 4. Generation time (left) and speedups (right) for the PolyORB specification

hyper-threading since the supra-linear acceleration factor was not observed on classic dual processors (without hyper-threading). This hypothesis is confirmed by [18].

The stall observed in figure 3 for the Client-Server specification parameterized with 100 processes is simply due to the fact that the execution time is very small (<5s): the specification becomes too simple to compute for a number of nodes superior to 16.

4.3 Other considerations

Another measure of interest is the network bandwidth consumption: measured bandwidths per node are from 230 KB/s for the PolyORB specification to 1.5 MB/s for the Client-Server specification, in a configuration with 20 nodes. That means that we do not use more than 30 MB/s of total bandwidth in this configuration, which any modern switch should handle easily.

We also observe that the larger the state space, the more efficient dmcG is. We impute this to the fact that there are more chances for a state to have at least a successor that is then distributed to another host, leading the *NewStatesSet* of each node to never be empty during the computation (which would make idle hosts).

Finally, a preliminary campaign on a larger cluster with 128 dual-processors nodes shows that dmcG scales up very well: we continue to observe a growing linear speedup with large models, as well as a homogeneous distribution.

5 Conclusion

In this paper, we presented dmcG, a distributed model checker working on a symbolic state space. The goal is to stack two accelerating techniques for model checking in order to get a more powerful tool. As a basis for the core functions of this distributed model checker, we used GreatSPN implementation without changing it. It was connected to a library dedicated to the distribution of model checking: libdmc.

Performances on several models, including industrial-like case study (the verification of a middleware's core) are almost optimal. We observe a nearly linear speedup with, in some favorable cases, a supra-linear speedup (due to the intensive use of both distribution and multi-threading). Using dmcG, we can push memory and CPU capacity of our model-checker one to two orders of magnitude further.

So far, our model checker basically manages safety properties and, when a property is not verified, provides an execution path to the faulty state. It takes as input AMI-nets or native GreatSPN format models, with additional parameters to specify properties. The multi-node version still requires some skill to install and configure, but we plan to make it available soon. The multi-threaded single node version is immediately useful to owners of a bi-processor machine.

We are currently challenging libdmc to handle the verification of temporal logic formulae.

References

1. Burch, J., Clarke, E., McMillan, K.: Symbolic model checking: 10^{20} states and beyond. Information and Computation (Special issue for best papers from LICS90) **98**(2) (1992) 153–181
2. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In Jensen, K., Rozenberg, G., eds.: Procedings of the 11th International Conference on Application and Theory of Petri Nets (ICATPN'90). Reprinted in High-Level Petri Nets, Theory and Application., Springer-Verlag (1991)
3. Ciardo, G., Luettgen, G., Siminiceanu, R.: Efficient symbolic state-space construction for asynchronous systems. In Nielsen, M., Simpson, D., eds.: Application and Theory of Petri Nets. Volume 1825 of Lecture Notes in Computer Science., Springer-Verlag (2000) 103–122
4. Barnat, J., Forejt, V., Leucker, M., Weber, M.: DivSPIN - a SPIN compatible distributed model checker. In Leucker, M., van de Pol, J., eds.: 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05), Lisbon, Portuga (2005)
5. Stern, U., Dill, D.L.: Parallelizing the Murφ verifier. In: Proceedings of the 9th International Conference on Computer Aided Verification, Springer-Verlag (1997) 256–278
6. Lerda, F., Sisto, R.: Distributed-memory model checking with SPIN. In: Proc. of the 5th International SPIN Workshop. Volume 1680 of LNCS., Springer-Verlag (1999)
7. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in eddy. In Valmari, A., ed.: SPIN. Volume 3925 of Lecture Notes in Computer Science., Springer (2006) 108–125
8. Garavel, H., Mateescu, R., Smarandache, I.: Parallel State Space Construction for Model-Checking. Volume 2057. (2001)
9. Thierry-Mieg, Y., Ilié, J.M., Poitrenaud, D.: A symbolic symbolic state space representation. In: 24th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'04), Springer Verlag, LNCS 3235 (2004) 276–291

10. : GreatSPN V2.0, http://www.di.unito.it/~greatspn/index.html (2007)
11. Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Baarir, S., Vergnaud, T.: On the Formal Verification of Middleware Behavioral Properties. In: 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04), Elsevier (2004) 139–157
12. Bonnefoi, F., Hillah, L., Kordon, F., Frémont, G.: An approach to model variations of a scenario: Application to Intelligent Transport Systems. In: Workshop on Modelling of Objects, Components, and Agents (MOCA'06), Turku, Finland (2006)
13. Hamez, A., Kordon, F., Thierry-Mieg, Y.: libDMC: a library to Operate Efficient Distributed Model Checking. In: Workshop on Performance Optimization for High-Level Languages and Libraries - associated to IPDPS'2007, Long Beach, California, USA, IEEE Computer Society (2007) to be published
14. Rangarajan, M., Dajani-Brown, S., Schloegel, K., Cofer, D.D.: Analysis of distributed spin applied to industrial-scale models. In Graf, S., Mounier, L., eds.: SPIN. Volume 2989 of Lecture Notes in Computer Science., Springer (2004) 267–285
15. : Message Passing Interface http://www.mpi-forum.org/ (2007)
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
17. Duret-Lutz, A., Poitrenaud, D.: 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), Volendam, The Netherlands, IEEE Computer Society Press (2004) 76–83
18. Koufaty, D., Marr, D.T.: Hyperthreading Technology in the Netburst Microarchitecture. IEEE Micro **23**(2) (2003) 56–65