libDMC: a Library to Operate Efficient Distributed Model Checking

Alexandre Hamez*, Fabrice Kordon and Yann Thierry-Mieg
Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05, France
Alexandre.Hamez@lip6.fr, Fabrice.Kordon@lip6.fr, Yann.Thierry-Mieg@lip6.fr

Abstract

Model checking is a formal verification technique that allows to automatically prove that a system's behavior is correct. However it is often prohibitively expensive in time and memory complexity, due to the so-called state space explosion problem. We present a generic multi-threaded and distributed infrastructure library designed to allow distribution of the model checking procedure over a cluster of machines. This library is generic, and is designed to allow encapsulation of any model checker in order to make it distributed. Performance evaluations are reported and clearly show the advantages of multi-threading to occupy processors while waiting for the network, with linear speedup over the number of processors.

1 Introduction

Software model checking is a formal analysis technique that allows one to verify the behavior of a specification. The basic principle is to exhaustively explore the state space (represented as a finite labeled transition system) of a modeled system. Such an operation is performed by a *model checker* and requires the specification to be finite and formally defined using formal languages like Promela [11], LOTOS [8] or Petri Nets [10].

Model checking is a well accepted approach to analyze specifications of hardware and (network) protocols, and is now increasingly being applied to software systems [11]. There is a particular complexity in the verification of distributed systems due to asynchronous communications. Thus, efficient and very sophisticated techniques like decision diagrams [3], partial order reduction techniques [18], or exploitation of system symmetries [5] have been developed to reduce the complexity of the procedure.

However, they still require the storage of a large number of states in main memory, thus they scale up with difficulty to industrial size problems. For example, the verification of a middleware's core in [12] could not be achieved for large configurations (large number of modeled threads) due to both implementation constraints of the model checker and intensive use of memory and CPU.

In recent years, distribution of model checking appeared to be a solution to increase memory capacity and CPU by taking advantage of a network of workstations. The principle is quite simple: states of the state space have to be distributed over a set of machines that compute them separately. The use of a cluster seems interesting for this because it can be dedicated to the computation and usually enjoys a high bandwidth dedicated network.

This paper presents libDMC, a library to encapsulate a model checker in order to use the capacity of a cluster for state space generation. We aim to produce a way to quickly parallelize a model checker and to use it on a cluster for evaluating *reachability properties* (also called *safety properties*). Such properties are constraints over the state space (for example an invariant). Reachability properties are easier to use than causal properties (expressed by means of temporal logic formulas). Indeed, engineers often use OCL[1] [17] constraints and program asserts (that naturally correspond to reachability properties).

Our library is designed to face the following challenges: it should allow integration of *existing* model checkers. Our goal is to distribute several existing model checkers, not redevelop from scratch distributed versions of existing algorithms. However this requires a generic environment dedicated to distribution of any type of state space construction. Additionnaly we would like to make the best use possible of available resources. We target clusters of multi-core or multiprocessor hosts, thus we want a

* A. Hamez has double affiliation: UPMC/LIP6 and EPITA Reseach and Developpement Laboratory (LRDE).

[1] Objet Constraint Language

parallel (i.e., multi-threaded) implementation. Some further challenges due to the integration of monolithic legacy code in a multi-threaded environment were faced to allow this.

The paper is structured as follows. Section 2 briefly presents the principles of model checking and, in particular, the reachability analysis problem. Then, Section 3 explains the architecture of our distributed model checking library. We provide some details on the implementation of our tool in Section 4 prior to a discussion on its performances in Section 5.

2 Sequential and Distributed Model Checking

The goal is to perform reachability analysis by controlling an invariant in each state as it is visited. To do so, we need to store all states in order to *1)* detect when a state has already been visited and *2)* exhibit a faulty execution path when the property is not verified.

2.1 Reachability Analysis

To check whether an invariant P is not preserved (thus a faulty behavior is detected), the verification using reachability analysis can be described as follows [6]. A labeled finite state machine M is defined as a four tuple $M = \langle S, S_0, T, L_P \rangle$ where: S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is a transition relation, $L_P : S \rightarrow \{true, false\}$ is a function that labels with *true* each state that satisfies P. A path in the structure M from a state s is a finite sequence of states $\pi = s_0 s_1 s_2 \dots s_n$ such that $s_0 \in S$ and $(s_i, s_{i+1}) \in T$ holds for all $i \geq 0$. The set of *reachable* states *reach* is defined as $s \in reach \Leftrightarrow \exists \pi = s_0 s_1 s_2 \dots s_n, s_0 \in S_0, s = s_n$, i.e., there exists a path from an initial state s_0 to s.

```
1  foreach $s \in S_0$ do
2      $todo$.push($s$)
3      $reach$.add($s$)
4  while $todo \neq \emptyset$ do
5      $s := todo.pop()$
6      foreach $s' \in trans(s)$ do
7          if $\neg reach.contains(s')$ then
8              $todo.push(s')$
9              $reach.add(s')$
10             if $labels(s', P) = false$ then return $P$ is false.
11 return $P$ is true.
```

Algorithm 1: Reachable state space generation algorithm.

Given S_0, T and L, the verification of a safety property P by model checking consists in determining whether $\nexists s \in Reach$, such that $L_P(s) = false$ (i.e., all reachable states verify the invariant P). To this end, explicit-state based model checkers[2] run algorithm 1 and are usually constructed using:

1. *trans*: a function representing the transition relation, which returns, for a given state s, the set of its successors by T. This function is often quite complex, as its definition depends on the formalism used in the model checker. Furthermore, many state-space reduction techniques are implemented inside this function such as state canonization to exploit symmetries [20, 5], state compression and decompression schemes to obtain compact state signatures for storage [11], optimizations related to accelerating the detection of enabled events, to guiding the procedure by selecting the order in which events are considered, or suppressing some of the enabled events in given states without invalidating the procedure [11], etc.

2. *labels*: a function representing the labeling function L, such that $labels(s, P)$ is true in a state s such that P holds in that state. This labeling function is usually implemented by running some boolean tests on the state, and typically has a low complexity.

3. *reach*: a compact data structure to store the reachability set currently under construction, in a manner that allows a low complexity test *contains* for presence of a state and fast insertion *add* of states. Most commonly, splay trees, hash tables and variants of Bloom filters are found [22].

4. *todo*: a queue (for breadth first search) or stack (for depth-first search) to store states that have yet to be fully explored. *pop* refers to the operation that extracts a state and *push* adds a state.

For verification of safety properties, full storage of the state graph is not required during construction, although parts of the graph may need to be reconstructed *a posteriori* to obtain a witness trace to the undesired state.

2.2 Distributed and Parallel Model Checking

The previous algorithm highlights the characteristics of a generic model checker. Our goal is to implement it in a parallel and distributed context, to allow model checking of larger models. We observe that **computationally**, the most

[2]By opposition to symbolic-based model checkers using decision diagrams as data structures.

expensive treatment is usually the function *trans*, due to its numerous optimizations. When the complexity of *trans* is dominant, we note the possibility of computing successors in parallel instances of *trans* that need not interact, except to avoid treating the same state twice.

Memory-wise, the test *reach.contains*(s') on line 7 is the critical point. If this test requires disk access, the model checking procedure usually does not terminate due to virtual memory swapping. An obvious solution at this level is to implement a distributed hash table scheme, to take advantage of a cluster's memory capacity.

In such a scheme, the call on line 7 first tests if the local host is the *owner* of this state. This is done using a static localization function (e.g., a checksum), that for each state designates a host. This function partitions the full state space over the hosts participating in the computation. Newly reached states are sent to their owner asynchronously. Note that the localization function should have a homogeneous distribution to ensure load balancing. In our case, we experimentally observe that load balancing is related to state distribution.

2.3 Related Work

Our study of the literature shows several attempts at proposing a distributed model checker. In [14] the authors implemented a parallel version of Spin. The problem however, was that the main state space reduction technique of Spin, called partial order reduction, had to be reimplemented 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 [19]). Another effort to implement a distributed Spin is the DivSpin [2] effort. However, they chose to reimplement 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 its 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 [20] for a distributed version of the Murphi verifier from Stanford. Murphi exhibits a costly canonization procedure. The original implementation in [20] was built on top of specific Berkeley NOW hardware[3], which limits portability. A more recent implementation [15] is based on MPI, 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 multiprocessors 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.

An important point is that we wish to reuse existing model checker implementations, not redevelop from scratch distributed versions of existing model checker algorithms. Our team has been maintaining a tool integration platform, that incorporates many explicit state based tools for the analysis of Petri nets. We thus wanted to develop a generic solution such that with as little modification as possible of existing tools (usually quite complex monolithic legacy C code) we could offer the computation power of a cluster.

As a test case for the tool, we wished to implement a distributed version of the tool GreatSPN 2.0 [7], originally developped in Torino and now co-maintained between Torino and LIP6. The part of the tool concerned (the model checking kernel) is over 86 KLOC of C. It implements extremely efficient symmetry based reductions that in favorable cases allows to reduce exponentially the size of the state space [5]. Recent advances [1] have perfected the tool to allow exploitation of partial symmetries, broadening the range of models and properties that can be verified. The compromise is that the transition relation is costly computationally due to a so-called canonization procedure like Murphi, but the algorithm yields a small state space. This setting is favorable to a distributed approach, as shown by the success of the Murphi-based tools. Computation time is a critical issue with GreatSPN as some computations can last two days on a single machine without exhausting memory.

3 Architecture of libDMC

libDMC is a library designed to offer a distributed and parallel implementation of existing, explicit-state based, model checkers. To limit dependences between a given model checker and our library, we chose to define an interface for interaction based on the description of a labeled finite state machine (see Section 2.1).

We can implement such an interface at the model checker kernel level by extracting the primitives related to: determining the initial states set S_0, computing a state's successors *trans*, and labeling states with the truth value of a property *labels*. The internal description of states used by the kernel is thus a black box, of which we only assume that a state is considered as a contiguous segment of memory, and that a state has a unique interpretation on all hosts. Given these constraints, we replace the main loop of a model checker by a distributed controller, and the existing data structures *todo* and *reach* by our own implementations. This architecture allows libDMC to make no hypothesis on the encapsulated model checker data structures.

[3]The Berkeley Network of Workstations

The drawback of such an interface is that no additional informations can be transmitted with a state. At first, we thought that we needed more informations on states in order to have an homogeneous distribution of states. But, as shown in section 5.1, we were able to attain this goal without any additional hints. So, if a model checker designer wants to store extra informations on a state for model checking purposes, he can do it when constructing a state.

We present libDMC's architecture in two steps. First we outline the design of a model checker engine as an assembly of autonomous components which interact only via abstract interfaces. Then we present how this architecture can be distributed and discuss the coordination of the distributed execution.

3.1 A Generic Architecture for State Space Generation

libDMC architecture is structured using three majors components (Figure 1), that represent the main data structures identified in Algorithm 1.

The *NewStatesSet* handles the new states (*todo*). *StateManager* and *FiringManager* share access to it, the *FiringManager* uses pop and *StateManager* uses push on it.

The *FiringManager* (which handles *trans*) pops states from the *NewStatesSet* and invokes the successor computation, which has been extracted from the (existing) model checker. Then each successor is transfered to the *StateManager* to be processed. The *FiringManager* is multithreaded; this multi-threading allows to take advantage of possible multiprocessorss/core computers in a cluster (parallel computation of *trans*). Therefore, each component of libDMC is thread-safe.

The *StateManager* (which handles *reach*) determines whether a state is new or not. If a state s is new, the *StateManager* inserts s into its unicity table and puts it in the *NewStatesSet*. Otherwise, it is simply discarded.

The implementation of the successor computation component is left to the existing model checkers. A simple interface is defined to obtain: the set of initial states S_0, the successors (through an iterator interface) of a given state, and the labeling function. libDMC is mostly independent from any model checker implementation choices because both the successor computation function and states representation are parameters of our library. States are seen as raw data to be processed, transmitted and stored. Therefore, model checker designers only bring the semantic of their formalisms, our library takes care of the rest.

This separation of concerns is important since designing and implementing a formalism and all associated semantics is a difficult task. This way, model checkers designers can focus on their formalisms without having to bear the burden of networks and threads implementations aspects.

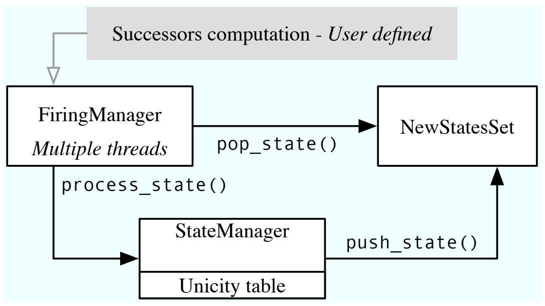


Figure 1. libDMC **architecture in local generation mode**

3.2 A Distributed Architecture for State Space Generation

For the distributed version of libDMC, we dissociate the state space generation from its distribution. We expect this mechanism to greatly simplify future extensibility of libDMC.

The algorithm of local generation is not modified except that the *FiringManager* interacts with a new implementation of the *StateManager* interface, realized by three new dedicated components (strong boxes in Figure 2).

- The *DistributedStateManager* computes the owner of each state (i.e., it says if a state is local or not).
- A set of *StateManager Proxies* represents distant hosts. Each state transmitted to one of these proxies is forwarded to the corresponding *StateManager Service* (proxy design pattern [9]).

FiringManager threads are constantly competing to compute new states, thus communication latency is overlapped by parallel successor computations, allowing full CPU usage.

3.3 Communication Models

Two communication models are used in two different layers:

- *Supervision* handles initialization, execution monitoring and detection of procedure termination, following a master / slave model.
- *State-space generation* is handled in a peer-to-peer manner. Peers exchange states using $n-1$ connections to other peers.

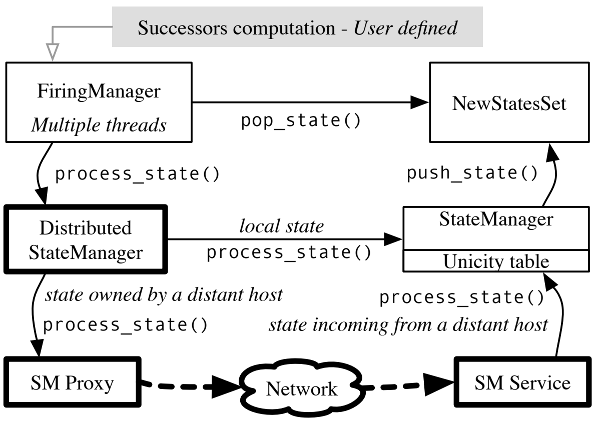


Figure 2. libDMC architecture in distributed generation mode

The supervision layer is handled by a master host. This host starts the program and deploys processes over all the hosts. It also is in charge of detecting termination and stopping the program. Supervision is inexpensive (few communications), thus the master host can also participate in the computation.

Termination is handled by a dedicated monitor running on the master host. It uses a local inactivity measure for a host based on the fact that *1)* all threads are inactive and *2)* its *NewStatesSet* is empty. Peers notify this condition (i.e., it has become active or inactive) to the master when it occurs. When this condition is met on all peers, an additional test is run to check that *3)* no message remains in the network, by checking that all sent states have been received (a simple difference of sent and received states).

4 Implementation

libDMC implements the principles of the previous section. This section lists some technical aspects of this experimentation.

Language. To implement libDMC, we chose C++ for its support of object-oriented programming capabilities and its efficiency. Moreover, interface with C is easy to achieve; it is important since most model checkers are developed in C.

Encapsulating model checkers. We successfully experimented the encapsulation of two model checkers coming from different research teams: GreatSPN [7] (Univ. Torino) and CheckPN [4] (Univ. P. & M. Curie). So far, we only exploit the reachability analysis capabilities of these tools.

Communications support. As MPI 1.1 was not clearly thread-safe and as we couldn't find a complete implementation of MPI 2 (which theoretically adds support for multiple threads) at the time of the design of libDMC, we used TCP to handle network communications. This solution is portable and offers a low overhead compared to higher-level communications libraries.

Making encapsulated model checkers thread-safe. One challenge was to multi-thread existing model checker implementations. GreatSPN for instance 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.

State balancing algorithm. A key-point of distributed model checking is the localization function, which returns the host that owns a given state (i.e., it is stored in its local memory). To be generic, we had to chose a function independent from state representation, that ensures a uniform distribution of states over the cluster hosts. We chose MD5 [16] (after testing some others) as it is able to compute a checksum on raw data and is known to have a uniform distribution.

4.1 Interfaces for encapsulated model checkers

The general interface presented in Section 3 for the state space generation is described through the C++ abstract interface shown at Figure 3.

```
typedef struct
{
  void* state_content;
  size_t state_size;
} state;

class abstract_formalism
{
public:

  virtual state*
    get_initial_state() = 0;

  virtual abstract_successors_iterator*
    get_succ_iterator(state* s) = 0;
};
```

Figure 3. Interfaces of the `abstract_formalism` **class**

Designers of model checkers just have to implement this interface (and some others, which are necessary for implementation details, such as the ones described below) to describe how the state space of a model is generated.

The `get_initial_state` method returns the initial state (lines 11-12) through a `state` data structure (lines 1-5). This structure contains a pointer to the state content and its size. As one can see, states are completely opaque to libDMC.

The `get_succ_iterator` method returns an iterator on successors of a state `s` (lines 14-15). This iterator has to implement the `abstract_successors_iterator` interface.

5 Performances

Performances 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: state balancing and obtained speedup.

Performances have been computed using GreatSPN. The following parametric specifications have been selected:

- the Dining Philosophers, a well known academic example; its complexity grows with the number of philosophers,
- the model of the PolyORB middleware kernel [12]; its complexity grows with the number of threads in the middleware,
- the specification of a telephone commutator [21]; its complexity grows with the number of subscribers.

The first model is academic and served as a validation example for our encapsulated model checkers. The two others correspond to industrial case studies and their associated specification is much more complex. Moreover, these models were developed independently from this project and thus serve as "fair" benchmarks.

Let us note that the analysis of the PolyORB kernel could not be achieved for more than 17 threads (12 055 899 symbolic states, the equivalent of 6.57×10^{17} concrete states without symmetry reductions). Computation then took more than 40 hours. Such a size is quite common when analyzing industrial systems. Parallelization of model checking indeed allows to apply it on larger models.

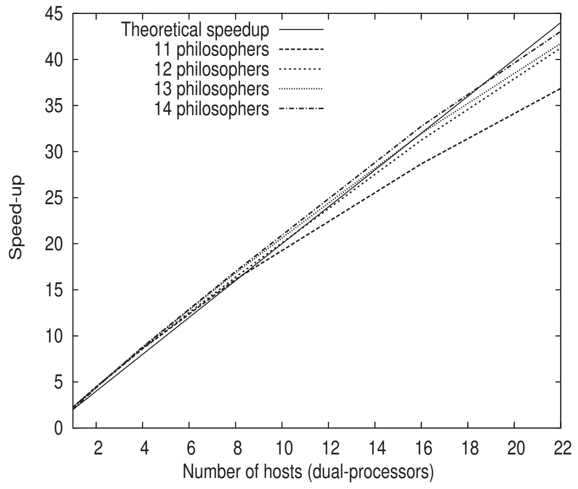
5.1 Load Balancing

The fact that states are homogeneously distributed over the involved hosts in the cluster is an important issue since it is related to load balancing of the application. 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. We then compared these results to the theoretical mean and noted the variation. These measures are summarized in Table 1. In this table, column 1 represents the parameter that scale up the model, column 2 the number of involved hosts, column 3 the total number of states, column 4 the theoretical mean, column 5 the standard deviation and column 6 the standard deviation expressed in percentage.

The measures show an even state distribution up to 16 hosts. However we observed that for 16 hosts and over, there are two groups of state distribution. This pushes the standard deviation up by an order of magnitude. This problem may be related to the use of MD5, and requires further investigation.

5.2 Speedup



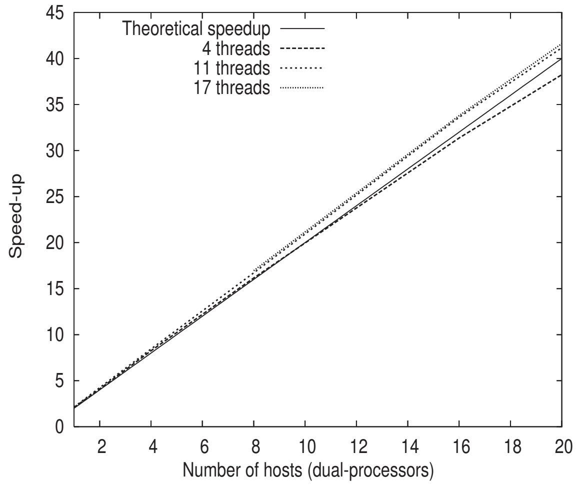


Figure 4. Speedups for Dining Philosophers (top) and PolyORb (bottom).

Figure 4 shows the speedup curve we obtain for the Philosophers and the PolyORB specifications. Distributed

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%
PolyORB middleware					
11 threads	4	3 366 471	841 618	1021	< 0.1%
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%
Telephone Service					
4 subscribers	4	12 544 968	3 136 242	1138	< 0.1%
4 subscribers	8	12 544 968	1 568 121	969	< 0.1%
4 subscribers	16	12 544 968	784 060	647	< 0.1%
4 subscribers	20	12 544 968	627 248	20 055	3.2%

Table 1. States distribution for the Philosophers, PolyORB and Telephone Service models

execution time is compared to the standard version of GreatSPN (i.e., not plugged to libDMC). Our library induces a low overhead: execution for the mono-threaded, single host version linked with libDMC within 95% to 105% of the standard version. The local but multi-threaded version is truly twice as fast on a bi-processor machine.

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 (of a few percents). We observed this in near all our experiments on several models with various parameters. We attribute this to hyper-threading since the supra-linear acceleration factor was not observed on dual core PowerPC 970, which doesn't have hyper-threading. This hypothesis seems to be correlated by [13]. Apparently, the multi-threading implementation enables an intensive use of all the processor units:

- all I/O and mutexes are overlapped by other threads,
- multi-threading probably increases a simultaneous use of dedicated hardware functions in the processors's pipe-line,
- shared code (between threads) and access to common data may introduce a better use of caches.

We also observe that the larger the state space, the more efficient encapsulated model checkers are. This is because there are more chances for a state to have at least a successor that is then distributed to another host, leading *todo* queues handled by the *NewStatesSet* to never be empty during the computation (which would make idle hosts).

6 Conclusion

In this paper, we presented libDMC, a library dedicated to the encapsulation of model checkers to distribute them. Our work is focused on state space generation and evaluation of reachability (or safety) properties, which is also a limit of most other existing implementations of distributed model checkers.

We have implemented and used libDMC to encapsulate two model checkers. We also measured performances on model checking of some large models. Our performance results are better than those of [20] (that report a near linear speedup over the number of hosts), with supra-linear to the number of processors speedup reported. Moreover:

- our solution offers a framework to ease the integration of existing model checkers,
- our solution relies on a portable architecture (sockets, pthread library) that takes maximum advantage of modern cluster characteristics,
- libDMC is the only library that multi-threads the generation of the state space, thus leading to excellent results on multiple processors architectures.

We succeeded in analyzing specifications that could not be computed before, due to limitation of memory and CPU. If several days of computation may remain acceptable for the development of critical components, many projects could benefit from the linear speedup in the evaluation of a safety property. The gain in the size of the problems that can be treated is appreciable in any context.

In the future, we intend to extend the functionalities of libDMC to handle properties expressed in temporal logic (LTL in particular). We also intend to plug Spin into libDMC to further validate our framework genericity, and run more comparisons with existing solutions.

References

[1] S. Baarir, C. Dutheillet, S. Haddad, and J.-M. Ilié. On the use of exact lumpability in partially symmetricalwell-formed nets. In *Second International Conference on the Quantitative Evaluaiton of Systems (QEST 2005), 19-22 September 2005, Torino, Italy*, pages 23–32. IEEE Computer Society, 2005.

[2] J. Barnat, V. Forejt, M. Leucker, and M. Weber. DivSPIN - a SPIN compatible distributed model checker. In M. Leucker and J. van de Pol, editors, *4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, Lisbon, Portuga, 2005.

[3] J. Burch, E. Clarke, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation (Special issue for best papers from LICS90)*, 98(2):153–181, 1992.

[4] CheckPN is a simple model-checker for Petri Nets. `http://spot.lip6.fr/wiki/CheckPn/`.

[5] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, 1993.

[6] E. Clarke, O.Grumberg, and A. Peled. *Model Checking*. MIT Press, 2000.

[7] G. G. Editor, A. for Timed, and S. P. Nets. `http://www.di.unito.it/~greatspn/`.

[8] P. V. Eijk and M. Diaz, editors. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., New York, NY, USA, 1989.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] C. Girault and R. Valk. *Petri Nets for Systems Engineering*. Springer Verlag, 2002.

[11] G. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2004.

[12] J. Hugues, Y. Thierry-Mieg, S. Baarir, F. Kordon, T. Vergnaud, and L. Pautet. On the formal verification of middleware behavioral properties. In T. Arts and W. Fokkink, editors, *Proc. Ninth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 04)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[13] D. Koufaty and D. T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

[14] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.

[15] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in eddy. In A. Valmari, editor, *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2006.

[16] J. G. Myers and M. Rose. The Content-MD5 Header Field. Technical report, Internet draft standard RFC 1864, 1995.

[17] OMG. *OCL 2.0 Specification - Version 2.0 ptc/2005-06-06*. OMG, June 2005.

[18] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. 6th Int. Conf. on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390, Stanford, USA, June 1994. Springer Verlag.

[19] M. Rangarajan, S. Dajani-Brown, K. Schloegel, and D. D. Cofer. Analysis of distributed spin applied to industrial-scale models. In *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 267–285. Springer, 2004.

[20] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In *Proceedings of the 9th International Conference on Computer Aided Verification*, pages 256–278. Springer-Verlag, 1997.

[21] A. Vernier and E. Paviot-Adet. *Vérification et mise en œuvre de réseaux de Petri*, chapter Modélisation et vérification de l'interopérabilité de services de télécommunication, pages 233–252. Hermès Science, 2003.

[22] Wikipedia. Hash table — wikipedia, the free encyclopedia, 2007. [Online; accessed 22-January-2007].