Electronic Notes in Theoretical Computer Science 89 No. 1 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 14 pages

Distributed Model Checking: From Abstract Algorithms to Concrete Implementations

Christophe Joubert [1]

Inria *Rhône-Alpes* / Vasy
655, av. de l'Europe
38330 Montbonnot Saint-Martin, France

Abstract

Distributed Model Checking (Dmc) is based on several distributed algorithms, which are often complex and error prone. In this paper, we consider one fundamental aspect of Dmc design: *message passing* communication, the implementation of which presents hidden tradeoffs often dismissed in Dmc related literature. We show that, due to such communication models, high level abstract Dmc algorithms might face implicit pitfalls when implemented concretely. We illustrate our discussion with a generic distributed state space generation algorithm.

Key words: distributed and parallel computing, communication paradigms, message passing, state space generation, model checking

1 Introduction

Nowadays, large computational problems can be solved effectively by using the aggregate power and memory of many computers. A current trend is to use clusters of cheap Pcs or networks of workstations (Nows) as massively parallel machines, since they are widespread architectures in laboratories and companies compared to unaffordable supercomputers. A parallel program aims at performing actions simultaneously on parallel hardware, whereas a distributed program is a parallel program designed for execution on a network of autonomous processors that do not share a main memory. It is natural to try to apply massively parallel machines to computer aided verification, which is very demanding in terms of computing and memory resources.

For the last few years, parallel and distributed model checking (Dmc) have received a lot of interest. Numerous verification tools have been developed,

[1] Christophe.Joubert@inria.fr

ranging from distributed state space generation [11,14,13] to distributed on-the-fly symbolic model checking [4]. Tools like Spin [18], Murϕ [21], and Cadp [13] have been adapted to parallel and distributed machines. Different approaches have been considered: distributed disk-based [16], multithreading [2], state compaction [19], dynamic load balancing [1], and shared and distributed memory [12].

As pointed out by Inggs [15], parallel and distributed model checking is a difficult problem. It requires a double expertise both in formal verification and in distributed/parallel computing. Model checking needs a number of steps, among which creating an abstract model of the application under study, checking this model for satisfaction of correctness requirements, and generating a counter example, if the requirements are violated. In addition, Dmc must address four fundamental aspects related to distribution: resource sharing, computation speedup, reliability, and communication.

There are basically two ways to express communication between processes in a parallel or distributed system: *message passing* or *shared memory*. In the case of shared memory models, synchronisation is the key point for performance. In the case of message passing, communication has to be modeled.

We are interested in investigating the impact of realistic modelization of message passing communications on abstract algorithms.

Most publications on parallel and distributed model checking, that provide explicit algorithms, make use of very abstract communication operations: Send and Recv. However, as we show in this paper, one cannot completely disregard the concrete implementation of these abstract operations.

Implementors of Dmc tools need to pay attention to the interactions between distributed processes. Choosing blocking or non-blocking communication may have diverse implementation consequences, in terms of program correctness and/or performance, such as deadlocks.

The remainder of the paper is organized as follows. Section 2 presents the distributed state space generation context in which we place the discussion. Based on this application, we introduce in Section 3 a taxonomy of asynchronous communication models and present the resulting design tradeoffs. Finally, Section 4 concludes the paper.

2 Distributed state space generation

The problem of generating in a distributed manner a state space has been studied for almost a decade. It was mainly first addressed by Caselli et al. [9,10] and Ciardo [12]. Since then, several implementations and improvements have been made, namely state representation [14], state distribution [3], and load balancing policies [20].

As we are more interested in the structure of distributed generation algorithms rather than specific optimizations, we will use as a basis for discussion the generic distributed state space generation algorithm presented on Figure 1.

```
V_i := ∅;  E_i := ∅;  T_i := ∅
if h(x_0) = i then V_i := {x_0} endif

while (⋃_{i=1}^n V_i ≠ ∅) ∨ (⋃_{i=1}^n channels_i ≠ ∅)
    if ∃x ∈ V_i then
        V_i := V_i\{x};  E_i := E_i ∪ {x}
        ∀(x -a-> y) ∈ succ(x)
            if h(y) ≠ i then
                SEND (x -a-> y, h(y))
            endif
            []
            if h(y) = i then
                UPDATE (V_i, E_i, T_i, x -a-> y)
            endif
    endif
    []
    RECV (x -a-> y);  UPDATE (V_i, E_i, T_i, x -a-> y)
endwhile

---------------------

procedure UPDATE (V, E, T, x -a-> y)
    if y ∉ E ∪ V then
        V := V ∪ {x}
    endif
    T := T ∪ {x -a-> y}
end
```

Fig. 1. Generic distributed state space generation algorithm

This algorithm illustrates the combination of three main operations needed to build a state space distributed on several machines: sending a transition (`SEND`), computing a transition locally (`UPDATE`), and receiving a transition (`RECV`). The operator `[]` points to places where we can introduce either non-deterministic choice or parallel composition. We will show in the next section how a proper interleaving of operations can be crucial to both the functional behavior of the distributed system and its performance.

In this algorithm, distributed generation takes place on n processes running in parallel. It begins by processing initial state x_0. States are partitionned with respect to a statically known hash function (h), which is assumed to distribute the state space uniformly over the processes, i.e., state s is assigned to process $h(s)$. Three sets are used by each process i to store respectively the visited states (V_i), the explored states (E_i), and the explored transitions (T_i). A general termination condition is sketched in the *while* statement. To

terminate, each of the n processes should have no more state left to explore and no more messages should be in transit inside the communication channels. Hence, when termination is detected, $\cup_{i=1}^{n} E_i$ represents the overall generated state space, and $\cup_{i=1}^{n} T_i$ represents the overall transition relation.

To make such a distributed algorithm work, we need a set of algorithmic components describing the distributed features of the algorithm: task partitioning, load balancing, canonical state representation (visited and explored) and graph representation (transitions) over the processes, communication layer, and termination detection.

The first four components are commonly described in the literature, and many optimizations have been proposed [14,20] in the context of model checking. The last two components are of interest, because they are often omitted in algorithms, especially communication, whose description is often skipped or reduced to a single sentence or a reference to a standard.

3 Message passing paradigms

During the execution of parallel or distributed programs, information has to be distributed and shared by means of communication between processes. In this section, we describe the message passing mechanism, and based on it, three communication paradigms with their respective tradeoffs.

3.1 Message passing mechanism

Message passing is a suitable communication mechanism between distributed processes running on architectures such as clusters of PCs or NOWs. Its purpose is to model interactions between processes within a distributed system. A message sent by a process is received by another process, which then must accept and act upon the message contents. There are typically few restrictions on how much information each message may contain. Thus, message passing can yield high throughput, making it a very effective way to transmit large blocks of data from one process to another. Yet, time overhead in handling each message (latency) may be high. We refer the reader to appendix A, which discusses four message passing mechanisms, and their use in DMC literature.

Direct management of communications by the programmer is the best way to achieve high throughputs and low latencies. But this explicit approach tends to be difficult to implement and debug, as the programmer is responsible for data distribution, synchronization, and message passing communications. Another solution is the introduction of implicit concurrent information by the compiler, like in *High Performance Fortran*. But this approach does not allow a fine control of memory usage. Therefore, it is not suitable for DMC computations, which have dramatic memory consumption.

Therefore, we need operations such that the program can manage commu-

nication at a fine grain level.

3.2 Asynchronous communication paradigms for Dmc

In this subsection, we explain why reliable asynchronous communication that doesn't make use of bounded buffering is error prone, and we then review the communication choices appropriate to Dmc.

In Dmc, we are primarily interested in correctness and efficiency. However, in numerous articles, e.g. [7,4,18,11], communication aspects are reduced to the minimum, although they might have a strong impact on distributed computations.

The communication layer available on clusters of Pcs and Nows, usually offers a mechanism to transfer messages from network to application and vice versa. Two options are possible: *synchronous* operations, blocking the calling process, and *asynchronous* operations.

- A sending call is said to be *synchronous* if the sender process is blocked until a reception acknowledgement of its message (*rendez-vous*). This acknowledgement means that the destination application has taken into account the message, and
- A receiving call is said *synchronous* if the destination process explicitly calls a primitive for receiving a message and possibly blocks until the message arrives.

On the contrary,

- An *asynchronous* operation (sending or receiving call) implies that it continues to take place in parallel with other operations of the same process.

At this point, we should clearly distinguish *non-blocking* operations from *asynchronous* operations, since they are often confused. A non-blocking operation is simply one that does not block the calling process at system level.

Figure 2 gives a clear picture of where communication aspects take part in the overall distributed computing taxonomy.

Synchronous operations lead to applications that are simpler to verify and thus operations are simpler to use by the programmer. Asynchronous operations lead to more flexible applications, in which the programmer can choose himself the waiting points. Globally, it is better to prefer asynchronous communication, because it is easy to emulate synchronous operations with asynchronous ones, and because it is well adapted to distributed systems, which are inherently asynchronous.

In the following paragraphs, we attempt to define the conditions under which sending calls can be used safely, depending on the type of asynchronous communication operations available in the system. The discussion is intended to be independent from the choice of asynchronous message passing system, and is relevant for receiving calls as well.

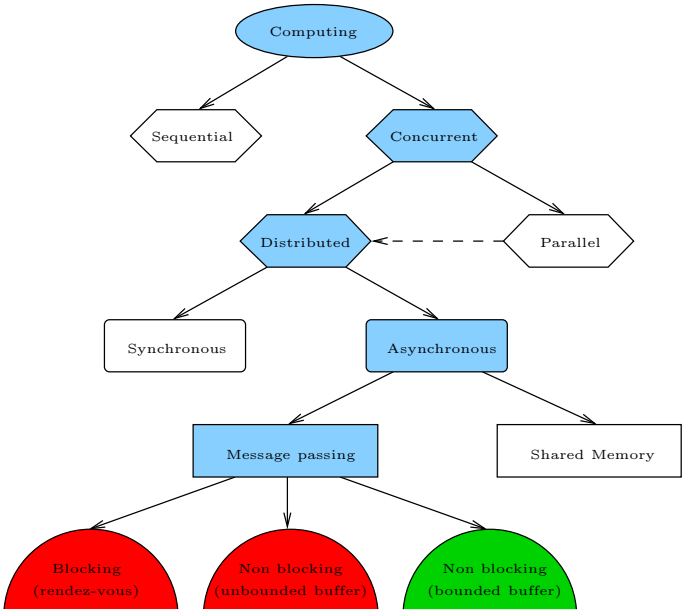

Fig. 2. Distributed computing taxonomy and communication choices

Blocking communication

Accomplishing the same efficient use of the network resources than with non-blocking communication is possible with blocking operations, but requires very careful ordering of computations, and hence much more difficult programming than in the non-blocking case. This is caused by synchronization delays when sender and receiver must cooperate (rendez-vous). If the partner doesn't react quickly, a delay results. Hence, there is a performance tradeoff caused by reacting quickly, since it requires devoting CPU and memory resources to check for communication operations to do.

Based on the following small distributed system example with only two processes, we highlight which problems can arise with blocking communication and what are the possible solutions.

Process 1	Process 2
`SEND(data A, process 2)`	`SEND(data B, process 1)`
`RECV(data B, process 2)`	`RECV(data A, process 1)`

This program is not guaranteed to always terminate. It depends on the size of the messages (`data`), the particular platforms (clusters, NOWs), and even the environment (e.g., free swap space). For short messages, the program will almost always work, particularly if the sender and receiver OS can handle the message in kernel buffer. For larger messages, it will fail, since the messages must be buffered somewhere outside the program itself. Thus, the two processes will enter a deadlock, each process waiting for the other to execute the `RECV` call. This may seem unusual, but programs that process large amounts of data can easily exceed the amount of available kernel or user buffering.

A static ordering of blocking communications can be a solution to this problem, but for Dmc algorithms (Fig. 1), it is a hard task to find an ordering that ensures an efficient overlapping of computation (`UPDATE`) and communications (`SEND` and `RECV`). Another solution could be to defer the ordering at run time by interleaving sending and receiving operations on each process. This corresponds to the use of multithreading. There exists a recent work [6] using multithreaded Dmc approach on distributed memory parallel architectures. However, the complexity of static ordering operations disappears, but is replaced by the complexity of synchronization operations.

Non-blocking communication with unbounded buffer

In this section, we consider non-blocking communication, which can achieve maximal overlapping between communication and computation.

Many publications on Dmc assume constant successful transmission of messages and they do no test possible message buffer overflow [21,17]. In [6] for instance, an explicit use of unbounded asynchronous channels is made. However, they don't justify if it is a reasonable assumption or not. Even by using synchronization barriers between processes, there might be computational steps where the amount of messages to be sent by a process cannot fit in system communication resources. The work done by [18] refers to this problem when using Pvm (the same applies to Mpi) library, which buffers messages in memory without limitations.

With this mechanism, a sending call adds a new message to the buffer and returns control immediately. The approach gives maximum flexibility with unbounded buffer capacity, since executing a sending call will take no delay, and will always succeed.

In this paradigm, we assume that the buffer is unbounded. But, to implement an unbounded buffer, we have to consider system resource limitations. It is possible that a process may send more messages than can be actually stored in memory. If nothing is planned, it may cause the system to crash, and hence to abort the distributed computation.

In model checking, memory is a major bottleneck. Having no bound on the amount of memory used by buffers, overlaps states storage. Such approach cannot be acceptable for Dmc systems, which require controlled memory consumption by careful allocation.

Non-blocking communication with bounded buffer

With bounded buffers, whatever the memory size allocated for message buffering, an overflow is always possible unless a suitable flow control mechanism is used. Flow control consists in testing return values of communication operations, in knowing if buffers are full or not, and in managing computations so that communication eventually succeeds.

With finite buffer capacity, programming non-blocking communication is

tricky and difficult. This is one of the main reasons why it is not considered in most Dmc algorithms. For each sending call (resp. receiving call), the result or returned values must be checked in order to know if the call succeeded. If not, the program will postpone the call, resume other interleaved computations and will attempt to make the same call later again. This is similar to context switches, where a process has to save its environment when interrupted. For Dmc problems like state space generation in Figure 1, non-blocking communication with bounded buffers can be done in two different ways:

- If the program is monothreaded, one needs a careful ordering between sending, receiving and computing operations. Such an ordering may be to perform receiving calls before sending calls, or at least to give priority to receptions over other computations.
- If the program is multithreaded, one can use either blocking communications or non-blocking communications with synchronization barriers, so as to block processes until some buffer space becomes available.

4 Conclusion

Parallel algorithms published for Dmc are usually highly simplified and remain at a very abstract level. In particular, the precise effect of communication operations is often left unspecified. Designers should be aware that this lack of definition makes implementations error prone. There are many different possibilities to implement reliable Send and Recv operations (blocking, non-blocking, synchronous, asynchronous with respect to the taxonomy presented in Section 3). The message of this paper is that it is not sufficient to write Dmc algorithms without being concerned about communication model pitfalls, like possible failure of Send calls and no means to recover from such a situation. To be implemented in practice, an algorithm should be detailed enough so as to check the results of communication calls, and should deal with issues such as multithreading, non-determinism, as well as buffer management and flow control.

The goal of Dmc is to use hundreds or even thousands of computing processes to solve problems of ever increasing complexity. Since the number of exchanged messages is proportional to the problem size, a scalable solution must focus on efficient communication means. Based on the above discussion, we believe that non-blocking communication with bounded buffers is the most efficient scheme provided that the user develops an appropriate flow control mechanism.

Acknowledgement

We are grateful to Hubert Garavel and Radu Mateescu (Inria Vasy) for their insightful comments about the content of the paper.
We would like to thank the anonymous referees for their constructive remarks. Last but not least, we address special thanks to Nicolas Descoubes, who gave the opportunity to discuss many technical points.

References

[1] S. Allmaier, S. Dalibor, and D. Kreische. Parallel graph generation algorithms for shared and distributed memory machines. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97 (Bonn, Germany)*, volume 12, pages 581–588. Elsevier, North-Holland, 1997.

[2] S. Allmaier, M. Kowarschik, and G. Horton. State space construction and steady-state solution of gspns on a shared-memory multiprocessor. In *Proceedings of the 7th IEEE International Workshop on Petri Nets and Performance Models PNPM'97 (Saint Malo, France)*, pages 112–121. IEEE Computer Society Press, 1997.

[3] S. C. Allmaier and G. Horton. Parallel shared-memory state-space exploration in stochastic modeling. In G. Bilardi, A. Ferreira, R. Lüling, and J. Rolim, editors, *Proceedings of the International Conference on Solving Irregularly Structured Problems in Parallel*, volume 1253 of *LNCS*, pages 207–218. Springer Verlag, 1997.

[4] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, (FMCAD'00)*, volume 1954 of *LNCS*, pages 390–404. Springer Verlag, November 2000.

[5] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[6] S. Blom and S. Orzan. A distributed algorithm for strong bisimulation reduction of state spaces. In L. Brim and O. Grumberg, editors, *Proceedings of the Workshop on Parallel and Distributed Model Checking PDMC'02 (Brno, Czech Republic)*, 2002.

[7] Benedict Bollig, Martin Leucker, and Michael Weber. Parallel model checking for the alternation free mu-calculus. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'01*, volume 2031 of *LNCS*, pages 543–558. Springer Verlag, 2001.

[8] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[9] S. Caselli, G. Conte, F. Bonardi, and M. Fontanesi. Experiences on simd massively parallel gspn analysis. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 794 of *LNCS*, pages 266–283. Springer Verlag, 1994.

[10] S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for gspn models. In G. De Michelis and M. Diaz, editors, *Proceedings of the 16th International Conference on Applications and Theory of Petri Nets (Torino, Italy)*, volume 935 of *LNCS*, pages 181–200. Springer Verlag, June 1995.

[11] G. Ciardo. Distributed and structured analysis approaches to study large and complex systems. In E. Brinksma, H. Hermanns, and J.-P. Katoen, editors, *First EFF/Euro Summer School on Trends in Computer Science (Berg en Dal, The Netherlands)*, volume 2090 of *LNCS*, pages 344–374. Springer Verlag, July 2001.

[12] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state space generation of discrete-state stochastic models. *INFORMS Journal on Computing*, 10(1):82–93, 1998.

[13] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, volume 2057 of *LNCS*, pages 217–234, Berlin, May 2001. Springer Verlag. Revised version available as INRIA Research Report RR-4341 (December 2001).

[14] B. R. Haverkort, A. Bell, and H. C. Bohnenkamp. On the efficient sequential and distributed generation of very large markov chains from stochastic petri nets. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models PNPM'99 (Zaragoza, Spain)*, pages 12–21. IEEE Computer Society Press, September 1999.

[15] Cornelia P. Inggs and Howard Barringer. On the parallelisation of model checking. In *Proceedings of the 2nd Workshop on Automated Verification of Critical Systems (AVOCS'02)*. Technical report, University of Birmingham, April 2002.

[16] W. J. Knottenbelt and P. G. Harrison. Distributed disk-based solution techniques for large markov models. In B. Plateau, W. J. Stewart, and M. Silva, editors, *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains NSMC'99 (Zaragoza, Spain)*, pages 58–75. Prensas Universitarias de Zaragoza, September 1999.

[17] W. J. Knottenbelt, M. A. Mestern, P. G. Harrison, and P. Kritzinger. Probability, parallelism and the state space exploration problem. In R. Puigjaner, N. N. Savino, and B. Serra, editors, *Proceedings of the 10th*

International Conference on Computer Performance Evaluation - Modelling, Techniques and Tools TOOLS'98 (Palma de Mallorca, Spain), volume 1469 of *LNCS*, pages 165–179. Springer Verlag, September 1998.

[18] F. Lerda and R. Sisto. Distributed-memory model checking with spin. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking SPIN'99*, volume 1680 of *LNCS*, pages 22–39. Springer Verlag, July 1999.

[19] P. Marenzoni, S. Caselli, and G. Conte. Analysis of large gspn models: A distributed solution tool. In *Proceedings of the 7th IEEE International Workshop on Petri Nets and Performance Models PNPM'97 (Saint Malo, France)*, pages 122–131. IEEE Computer Society Press, 1997.

[20] D. Nicol and G. Ciardo. Automated parallelization of discrete state-space generation. *Journal of Parallel and Distributed Computing*, 47(2):153–167, 1997.

[21] U. Stern and D. Dill. Parallelizing the murφ verifier. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification CAV'97 (Haifa, Israel)*, volume 1254 of *LNCS*, pages 256–267. Springer Verlag, June 1997.

[22] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[23] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[24] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[25] David W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, 1994.

A Message passing systems

Traditional communication mechanisms for distributed memory architectures fall in one of these four categories: Tcp/Udp sockets over Ip, Rpc, Mpi or Pvm, and Active Message. A comparison of these message passing mechanisms should be useful to designers and users of parallel and distributed systems. Designers need to know which approach offers the highest efficiency with respect to their needs. This is possible by an accurate understanding of message passing's strengths and weaknesses. For each message passing system,

we make a link with corresponding Dmc research, and we justify the appropriateness of communication models to Dmc tools according to experiments and performance analyses present in Dmc literature.

A.1 Sockets and Tcp/Udp Ip

In a distributed environment, channels are often implemented with support from the kernel. The Unix socket interface [23] is by far the most common low-level network interface. Most standard network hardware is designed to support at least two types of socket protocols: datagram Udp and connected Tcp. These socket types are the basic network software interface for most of the portable, higher-level, parallel processing software, like Pvm, which uses a combination of Udp and Tcp. Another good feature is that Tcp provides reliable connection-oriented transport service, meaning that messages will not get lost or corrupted and will be delivered in the correct order.

However layered protocols suffer from quite low level of abstraction, memory to memory copy, poor code locality, and heavy functional overhead. Starting from Tcp/Ip protocols, a user will have to program a software communication layer in order to have efficient cluster computing. Indeed, network performances are exposed to costly buffer management, namely in the case of asynchronous messages. But it is worth noting that Unix socket interface provides a solution much like the Mpi non-blocking operations, though somewhat less convenient for the user. By checking return values, the careful users are allowed to avoid deadlock in their applications.

Lerda and Sisto [18] first tried to use Pvm (see below) to parallelize the Spin model checker, because Pvm was, at the time, a widely used package namely in the context of heterogeneous architecture. To satisfy a need for flow control over communication operations and due to high overhead introduced by using this library, they finally decided to use the socket interface. Moreover, in order to make the messages portable on different architectures, they set up an Xdr (eXternal Data Representation) layer on top of it. This case study shows very well that unbounded buffered asynchronous communication can lead to uncontrolled costly parallel programs. In [13], they also chose Unix sockets in order to use fine-tuned buffers and to set up efficient flow control in the context of state space generation.

A.2 Rpc

Remote Procedure Call [5] is an enhanced general purpose network abstraction atop socket. It is a simple concept mainly considered as a standard for distributed client-server applications. A client sends request by passing arguments to remote procedure, and blocks itself until it gets a response (reply) from the server. Besides high level of abstraction and sequential like programming, thanks to calling procedures, it offers canonical format (marshalling)

across different systems connected to the network in heterogeneous environment.

Although this mechanism is well adapted to autonomous distributed systems, the overhead to perform Rpc is very high compared to sockets. The emphasis on hiding network communication from the calling process leads the system to a loss of performance that is not favorable for Dmc tools. To our knowledge, no Dmc research made use of Rpc mechanism.

A.3 Pvm *and* Mpi

They are general purpose systems for message passing and parallel program management on distributed platforms at the application level, based on available Ipc (Interprocess Communication) mechanisms.

Pvm (*Parallel Virtual Machine*) [22] is a freely-available software package from Oak Ridge National Laboratories that permits an heterogeneous collection of Unix and/or Windows computers linked together by a network to be used as a single large parallel computer. It is a portable message passing library generally implemented on top of sockets and it is established as a main standard for message passing cluster parallel computing. Most of the ideas in Pvm and other basic message passing systems are incorporated in a relatively new official standard *Message Passing Interface* (Mpi). This important development tackles basic point to point, and collective communication.

Mpi [25] is a standardized message passing library defined by a wide community of scientific and industrial experts. Its design provides access to advanced parallel hardware for end users as well as library writers and tool developers. One of the goals of Mpi is to provide a clearly defined set of routines that can be implemented efficiently on many types of platforms and particularly on heterogeneous networks of cluster.

Due to a set of automatically handled issues (coordination, polling, interrupt, delivering messages), Mpi and Pvm message passing calls generally add significant overhead to standard socket operations, which already had high latency. Adding to that, Mpi and Pvm do not address issues such as distributed computing and wide-area networks, but focus on parallel problems. Finally, heterogeneity of processors is also a drawback in writing parallel programs with current versions of Mpi and Pvm. Indeed, although these libraries are portable, they need a specific executable on each machine of the node group when running the parallel application. Another research trend is to use Java to solve this disadvantage. A Java version of Mpi has already been developed [8].

The majority of Dmc research has been using either Mpi or Pvm. Caselli et al. [10] made the first attempt to use message passing to parallelize the *Tangible Reachability Graph* generation problem. Like most of early Dmc projects [12,19,1,17,16], they were using Pvm interface. Numerous Dmc tools

are now working with the standard Mpi [12,19,20,17,16] and with different implementations of this standard, like Mpich [14] or Lam/ Mpi 6.5.6 [6]. Problems encountered with Mpi in Dmc context are mainly starvation phenomenon [14] and memory management [6], where a proposed solution was to build specific layers to deal with both message size and message content to be sent.

A.4 Active Message

The active message approach [24] is a one-sided communication mechanism, which avoids buffer management and stalled processors pitfalls by requiring the address of a message handling routine to be embedded in the header of each message. In this way, it is possible to overlap computation and communication because the receiving process does not have to perform a synchronous receive, and buffering is eliminated because the message is handled immediately upon receipt via the handling routine. Compared to Mpi and Pvm, active message is a fast message passing library in that the time to handle an incoming message at a node is very small.

One of the main drawbacks is that it is a very unsafe approach. It is possible but unreasonable to achieve safety in active message, since it would require adding new layers to the communication operations and it would increase the message passing overhead, which is already high. Nevertheless, it was used by Stern and Dill [21] to parallelize the Murϕ verifier, and it was combined with aggregation methods to decrease the time consumed in send and receive routines.

Much of message passing system overhead is usually caused by repeated copying of messages to and from buffers located in the OS kernel address space. Globally, higher level systems (Pvm, Mpi, Rpc, Active Message) focus on the standardization for interoperation and portability than efficient use of resources, like lower level systems (sockets).

Thus, it is not surprising to see recent works in Dmc [18,13] using Unix socket interface, since it is a good candidate, thanks to its flexibility, for asynchronous communication with bounded buffer, which appears to be the most appropriate communication paradigm to Dmc tools.