SimGrid MC: Verification Support for a Multi-API Simulation Platform

Stephan Merz, Martin Quinson, Cristian Rosa

SimGrid MC: Verification Support for a Multi-API Simulation Platform

Stephan Merz[1], Martin Quinson[2], and Cristian Rosa[3]

[1] INRIA Research Center Nancy, France stephan.merz@loria.fr
[2] Université Henri Poincaré Nancy 1, Nancy, France martin.quinson@loria.fr
[3] Université Henri Poincaré Nancy 1, Nancy, France cristian.rosa@loria.fr

Abstract. SimGrid MC is a stateless model checker for distributed systems that is part of the SimGrid Simulation Framework. It verifies implementations of distributed algorithms, written in C and using any of several communication APIs provided by the simulator. Because the model checker is fully integrated in the simulator that programmers use to validate their implementations, they gain powerful verification capabilities without having to adapt their code. We describe the architecture of SimGrid MC, and show how it copes with the state space explosion problem. In particular, we argue that a generic Dynamic Partial Order Reductions algorithm is effective for handling the different communication APIs that are provided by SimGrid. As a case study, we verify an implementation of Chord, where SimGrid MC helped us discover an intricate bug in a matter of seconds.

1 Introduction

Distributed systems are in the mainstream of information technology. It has become standard to rely on multiple distributed units that collectively contribute to a business or scientific application. Designing and debugging such applications is particularly difficult: beyond issues common to any parallel (i.e., concurrent or distributed) program, such as race conditions, deadlocks or livelocks, distributed systems pose additional problems due to asynchronous communication between nodes and the impossibility for any node to observe a global system state.

The most common approach to validating distributed applications is to execute them over a given testbed. However, many different execution platforms exist, and it is difficult to assess the behavior of the application on another platform than the one that the programmer has access to. Simulation constitutes another approach, offering the ability to evaluate the code in more comprehensive (even if not exhaustive) test campaigns. It remains however difficult to determine whether the test campaign is sufficient to cover all situations that may occur in real settings. That is why distributed applications are usually only tested on a very limited set of conditions before being used in production.

In recent years the use of formal validation techniques has become more prominent for the evaluation of concurrent and distributed software. Due to their simplicity and the high degree of automation, model checking techniques

have been particularly successful. Relying on exhaustive state space exploration, they can be used to establish whether a system, or a formal model of it, meets a given specification.

Initially, verification techniques were developed for formal modeling languages such as process algebras [1], Petri nets [2], Promela [3] or TLA$^+$ [4], in which algorithms and protocols can be modeled at a high level of abstraction. However, many errors are introduced at the implementation phase, and these can obviously not be detected by formal verification of models. Several authors have considered the application of model checking techniques to source or binary code [5,6,7,8], and our work contributes to this line of research. It promises to catch subtle bugs in actual programs that escape standard testing or simulation and to give non-specialists access to powerful verification methods.

The main impediment to make the approach work in practice is the well-known state explosion problem: the state space of even small programs executed by a few processes is far too large to construct exhaustively. Powerful reduction techniques such as dynamic partial-order reduction (DPOR [9]) must be used to cut down the number of executions that must be explored. DPOR relies on the notion of independent transitions, which must be established for the semantics of real-world programs, which can be a daunting task [10].

The contributions of this article are the following:

- We present SimGrid MC, an extension of the SimGrid simulation framework [11] for the formal verification of properties of distributed applications that communicate by message passing. We believe that by integrating verification capabilities into an existing simulation environment, we are able to close the loop of the development process: developers can assess their implementations for both correctness and performance, using the same overall framework.
- We detail the changes that we made to the main simulation loop to implement the model checking functionality, and how this was eased by the similarities between both tasks.
- We explain how we could implement the DPOR algorithm to support the different communication APIs offered by SimGrid through an intermediate communication layer, for which independence of transitions is established.

This article is organized as follows: Section 2 introduces the SimGrid simulation framework. Section 3 presents the model checker SimGrid MC, its implementation within the SimGrid framework, and our implementation of DPOR for multiple communication APIs available in SimGrid. Section 4 evaluates the resulting tool through several experiments. Finally, Section 5 concludes the paper and discusses future work.

1.1 State of the Art

The idea of applying model checking to actual programs originated in the late 1990s [5,8]. One of the main problems is the representation and storage of system states: C programs freely manipulate the stack and heap, and it becomes

difficult to determine the part that should be saved, and costly to actually store and retrieve it. Godefroid [12] proposed the idea of stateless model checking, in which executions are re-run instead of saving system states. Flanagan and Godefroid also introduced the idea of DPOR [9], although they were not primarily interested in model checking distributed systems. The closest work to ours is probably ISP [13], which is a stateless model checker for MPI applications that also relies on DPOR for effective reductions. ISP is not implemented in a simulation framework, but intercepts the calls to the runtime to force the desired interleavings. MACE [6] is a set of C++ APIs for implementing distributed systems; it contains a model checker geared towards finding dead states. To our knowledge SimGrid MC is the only model checker for distributed applications that supports multiple communication APIs and is tightly integrated with a simulation platform.

2 The SimGrid Framework

2.1 SimGrid Architecture

The SimGrid framework [11] is a collection of tools for the simulation of distributed computer systems. The simulator requires the following inputs:

The application or protocol to test. It must use one of the communication APIs provided in SimGrid, and must be written in one of the supported languages, including C and Java.

Analytical models of the used hardware. These models are used by the simulator to compute the completion time of each application action, taking in account the hardware capacity and the resources shared between application elements and with the external load.

A description of the experimental setup. This includes the hardware platform (hosts, network topology and routing), the external workload experienced by the platform during the experiment, and a description of the test application deployment.

The simulator then executes the application processes in a controlled environment, in which certain events, and in particular communications, are intercepted to evaluate timing and the use of shared resources, according to the models and the description of the setup (links, hosts, etc).

Figure 1 shows the architecture of the SimGrid framework. The analytical models of the resources are provided by the SURF layer, which is the simulation core. On top of this, SIMIX constitutes the virtualization layer. It adds the notion of process, synchronization, communication primitives, and controls the execution of the user processes. Three different communication APIs (or user interfaces) are built on top of the abstractions provided by SIMIX; they are adapted to different usage contexts. MSG uses a communication model that is based on messages exchanged through mailboxes; messages are characterized as tasks with computation and communication costs. GRAS is a socket-based

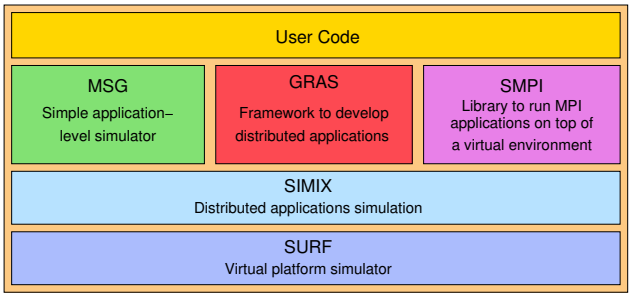

Fig. 1. The SimGrid Architecture.

event loop API designed to be executed in the simulator or deployed in real-life platforms. SMPI stands for Simulated MPI, and allows a user to simulate standard MPI programs without modifications, unlike the other two interfaces that are specific to SimGrid. It implements a subset of MPI that includes two-sided synchronous and asynchronous communication, as well as group operations.

The experimental setup is provided by the user through configuration files that instantiate the models, describing where the processes will be deployed.

Let us point out that SimGrid is a *simulator* and not an *emulator*. Its goal is to compute the timings of the events issued by the system as if it would execute on the virtual platform, irrespective of the speed of the platform on which the simulation is run. In particular, processes always execute at full speed.

2.2 The Simulation Loop

Before we present the model checker for SimGrid, we give some more details about the simulator's main loop, *i.e.* how the simulator controls the execution of the tested application depending on the platform models. This background is necessary in order to understand how SimGrid MC builds upon the infrastructure provided by the simulator.

SimGrid runs the entire simulation as a single process in the host machine by folding every simulated user process in a separate thread. The simulator itself runs in a special distinguished thread called *maestro*, which controls the scheduling.

Figure 2 depicts two simulation rounds starting at time t_{n-1} with two user threads T_1 and T_2 running the simulated processes and the maestro thread M. The round starts by calling SURF to compute and advance to the time of the next ending actions, in this case t_n. Next, it passes the list of finished actions to SIMIX that has a table associating them to the blocked threads. Using this table, SIMIX schedules all the unblocked threads. The user threads run without interruption until they block, waiting for a new simulation action, such as a communication. These actions are denoted in Fig. 2 by a and b for threads T_1 and T_2. The simulation round finishes once all the threads were executed until they block. Note that the time advances only between scheduling rounds, thus

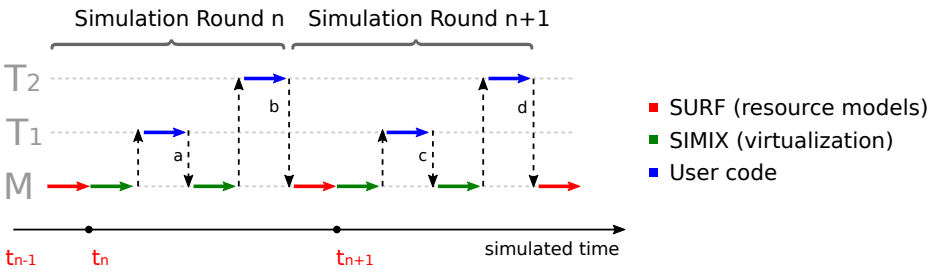

Fig. 2. Simulation Main Loop.

from the simulator's point of view, all the actions performed by the user in a scheduling round happen at the same time.

3 Model Checking Distributed Programs in SimGrid

SimGrid provides us with the capability of simulating distributed programs in a controlled environment. In particular, it includes functionality for managing the control state, memory, and communication requests of simulated processes, which can be run selectively and interrupted at visible (communication) actions. We now outline how we used this framework to implement verification capabilities, and describe the techniques we employed to make them efficient. In this article, we focus only on the C interface.

3.1 SimGrid MC

Unlike simulation, which is mainly concerned with the use of available resources and the performance of a distributed application in a given scenario, verification attempts to exhaustively explore the state space of a system in order to detect corner cases that one would be unlikely to encounter in simulation runs. Typically, the number of process instances will be significantly smaller in verification than in simulation because of the well known problem of state space explosion. We designed SimGrid MC as a complement to the simulator functionality, allowing a user to verify instances of distributed systems, *without requiring any modifications to the program code*. In the schema presented in Fig. 1, SimGrid MC replaces the SURF module and a few submodules of SIMIX with a state exploration algorithm that exhaustively explores the executions arising from all possible non-deterministic choices of the application.

As we explained before, a distributed system in SimGrid consists of a set of processes that execute asynchronously in separate address spaces, and that interact by exchanging messages. In other words, there is no global clock for synchronization, nor shared memory accessed by different processes.

More precisely, the state of a process is determined by its CPU registers, the stack, and the allocated heap memory. The network's state is given by the

messages in transit, and it is the only shared state among processes. Finally, the global state consists of the state of every process plus the network state. The only way a process can modify the shared state (the network) is by issuing calls to the communication APIs, thus the model-checker considers these as the only visible transitions. A process transition as seen by the model checker therefore comprises the modification of the shared state, followed by all the internal computations of the process until the instruction before the next call to the communication API. The state space is then generated by the different interleavings of these transitions; it is generally infinite even for a bounded number of processes due to the unconstrained effects on the memory and the operations processes perform.

Because the global state contains unstructured heaps, and the transition relation is determined by the execution of C program code, it is impractical to represent the state space or the transition relation symbolically. Instead, SimGrid MC is an explicit-state model checker that explores the state space by systematically interleaving process executions in depth-first order, storing a stack that represents the schedule history. As the state space may be infinite, the exploration is cut off when a user-specified execution depth is reached. Of course, this means that error states beyond the search bound will be missed, but we consider SimGrid MC as a debugging tool that is most useful when it succeeds in finding an error. SimGrid MC ensures complete exploration of the state space up to the search bound.

When state exploration hits the search bound (or if the program terminates earlier), we need to backtrack to a suitable point in the search history and continue exploration from that global state. In a naïve implementation, this would mean check-pointing the global system state at every step, which is prohibitive due to the memory requirements and the performance hit incurred by copying all the heaps. Instead, we adopt the idea of stateless model checking [12] where backtracking is implemented by resetting the system to its initial state and re-executing the schedule stored in the search stack until the desired backtracking point. Because global states are not stored, SimGrid MC has no way of detecting cycles in the search history and may re-explore parts of the state space that it has already seen. Note that even if we decided to (perhaps occasionally) checkpoint the system state, dynamic memory allocation would require us to implement some form of heap canonicalization [8,14] in order to reliably detect cycles. In the context of bounded search that we use, the possible overhead of re-exploring states because of undetected loops is a minor concern for the verification of safety properties. It would, however, become necessary for checking liveness properties.

Figure 3 illustrates the exploration technique used by SimGrid MC on the example used in Sect. 2.2. The model checker first executes the code of all threads up to, but excluding, their first call to the communication API (actions a and b in this example). The resulting global state S_0 (indicated by a red dot in Fig. 3) is pushed on the exploration stack; it is also stored as the snapshot corresponding to the initial state, and the model checker records the enabled actions. It then chooses one action (say, a) for execution and schedules the associated thread, which performs the communication action and all following local program steps

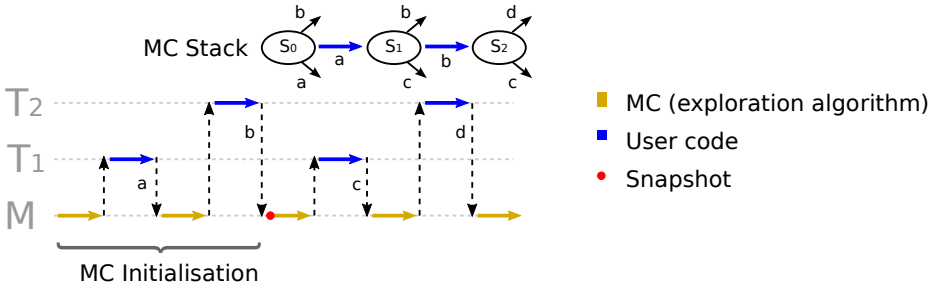

Fig. 3. State Exploration by SimGrid MC.

up to, but excluding, the next API call (c). This execution corresponds to one transition as considered by the model checker, which records the actions enabled at this point and selects one of them (say, b), continuing in this way until the exploration reaches the depth bound or no more actions are enabled; depending on the process states, the latter situation corresponds either to a deadlock or to program termination.

At this point, the model checker has to backtrack. It does so by retrieving the global state S_0 stored at the beginning of the execution, restoring the process states (CPU registers, stack and heap) from this snapshot, and then replaying the previously considered execution until it reaches the global state from which it wishes to continue the exploration. (The choice of backtrack points will be explained in more detail in Sect. 3.2 below.) In this way, it achieves the illusion of rewinding the global application state until a previous point in history.

Figure 4 illustrates the architecture of SimGrid MC. Each solid box labeled P_i represents a thread executing the code of a process in the distributed system being verified. The exploration algorithm is executed by a particular thread labeled MC that intercepts the calls to the communication API (dashed box) and updates the state of the (simulated) communication network. The areas colored blue represent the system being explored, the area colored red corresponds to

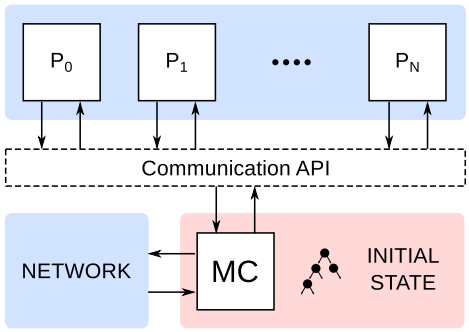

Fig. 4. SimGrid MC Architecture.

the state of the model checker, which holds the snapshot of the initial state and the exploration stack. When a backtracking point is reached, the blue area is reset as described above, but the exploration history is preserved intact.

3.2 Partial Order Reduction for Multiple Communication APIs

The main problem in the verification of distributed programs, even using stateless model checking, lies in the enormous number of interleavings that these programs generate. Usually, many of these interleavings are equivalent in the sense that they lead to indistinguishable global states.

Algorithm 1 Depth-first search with DPOR

```
 1: q := initial state
 2: s := empty
 3: for some p ∈ Proc that has an enabled transition in q do
 4:    interleave(q) := {p}
 5: end for
 6: push(s,q)
 7: while |s| > 0 do
 8:    q := top(s)
 9:    if |unexplored(interleave(q))| > 0 ∧ |s| < BOUND then
10:       t := nextinterleaved(q)
11:       q' := succ(t, q)
12:       for some p ∈ Proc that has an enabled transition in q' do
13:          interleave(q') := {p}
14:       end for
15:       push(s,q')
16:    else
17:       if ∃ i ∈ dom(s): Depend(tran(s_i), tran(q)) then
18:          j := max({i ∈ dom(s): Depend(tran(s_i), tran(q))})
19:          interleave(s_{j-1}) := interleave(s_{j-1}) ∪ {proc(tran(q))}
20:       end if
21:       pop(s)
22:    end if
23: end while
```

(Dynamic) Partial-Order Reduction [9] has proved to be efficient for avoiding the exploration of equivalent interleavings, and we also rely on this technique in SimGrid MC. The pseudo-code of the depth-first search algorithm implementing DPOR appears in Algorithm 1. With every scheduling history q on the exploration stack is associated a set *interleave*(q) of processes enabled at q and whose successors will be explored. Initially, an arbitrary enabled process p is selected for exploration. At every iteration, the model checker considers the history q at the top of the stack. If there remains at least one process selected for exploration at q, but which has not yet been explored, and the search bound has not yet been reached, one of these processes (t) is chosen and scheduled for execution,

resulting in a new history q'. The model checker pushes q' on the exploration stack, identifying some enabled process that must be explored. Upon backtracking, the algorithm looks for the most recent history s_j on the stack for which the transition $tran(s_j)$ executed to generate s_j is dependent with the incoming transition $tran(q)$ of the state about to be popped. If such a history exists, the process executing $tran(q)$ is added to the set of transitions to be explored at the predecessor of s_j, ensuring its successor will be explored during backtracking (if it has not yet been explored). Our algorithm is somewhat simpler than the original presentation of DPOR [9] because it assumes that transitions remain enabled until they execute, which is the case for SimGrid.

The effectiveness of the reductions achieved by the DPOR algorithm is crucially affected by the precision with which the underlying dependency relation can be computed. The two extremes are to consider all or no transitions as dependent. In the first case, the DPOR algorithm degenerates to full depth-first search. In the second case, it will explore only one successor per state and may miss interleavings that lead to errors. A sound definition of dependence must ensure that two transitions are considered independent only if they commute, and preserve the enabledness of the other transition, at any (global) state where they are both enabled. Because processes do not share global memory in SimGrid, memory updates cannot contribute to dependence, and we need only consider the semantics of the communication actions. However, their semantics must be described formally enough to determine (in)dependence.

In the case of the SimGrid Simulation Framework, the programs can be written using one of its three communication APIs, which lack such formal specification, as they were not designed for formal reasoning. Palmer et al. [10] have given a formal semantics of a substantial part of MPI for use with DPOR, but this is a tedious and daunting task, which would have to be repeated for the other APIs in SimGrid.

Instead, our implementation of DPOR in SimGrid relies on the definition of a minimal internal networking API, for which we have given a fully formal semantics and for which we have proved independence theorems in our previous work [15]. The three communication APIs provided by SimGrid are implemented on top of this basic API, and the DPOR-based model checker presented in Algorithm 1 operates at the level of these elementary primitives.

The communication model used by this set of networking primitives is built around the concept of "mailbox". Processes willing to communicate queue their requests in mailboxes, and the actual communication takes place when a matching pair is found. The API provides just the four operations *Send*, *Recv*, *WaitAny* and *TestAny*. The first two post a send or receive request into a mailbox, returning a communication identifier. A *Send* matches any *Recv* for the same mailbox, and vice versa. The operation *WaitAny* takes as argument a set of communication identifiers and blocks until one of them has been completed. Finally, *TestAny* also expects a set of communication identifiers and checks if any of these communications has already completed; it returns a Boolean result and never blocks.

Listing 3.1. Inefficient WaitAll

```
1 void WaitAll(comm_list[])
2 {
3   while(len(comm_list) > 0){
4     comm = WaitAny(comm_list);
5     list_remove(comm, comm_list);
6   }
7 }
```

Listing 3.2. Efficient WaitAll

```
1 void WaitAll(comm_list[])
2 {
3   for(i=0;i<len(comm_list);i++){
4     WaitAny(comm_list[i]);
5   }
6 }
```

We specified these primitives formally in TLA$^+$ [4], and for every pair of communication operations we formally proved conditions ensuring their independence. Moreover, it was surprisingly easy to implement SimGrid's communication APIs in terms of these primitive operations.

However, it was not clear a priori that this approach would result in a satisfactory degree of reduction, as the implementations of two high-level operations in terms of the lower-level ones might falsely be considered dependent. Moreover, the implementation of the higher-level APIs may introduce additional non-determinism, generating spurious interleavings during model checking. For example, consider the implementation of listing 3.1: it expects a set of communications identifiers and repeatedly uses *WaitAny* for all unfinished communications, until no one is left. While correct, such an implementation would introduce a non-deterministic choice among the finished communications, which is irrelevant to the semantics of *WaitAll* but would be considered by the model checker. For our purposes, it is therefore better to issue *WaitAny* operations in sequence for all the communication operations as shown in listing 3.2.

4 Experimental Results

In this section we present a few verification experiments using two of the APIs supported by SimGrid. We thus illustrate the ability of our approach to use a generic DPOR exploration algorithm for different communication APIs through an intermediate communication layer. Each experiment aims to evaluate the effectiveness of the DPOR exploration at this lower level of abstraction compared to a simple DFS exploration. We use a depth bound fixed at 1000 transitions (which was never reached in these experiments), and run SimGrid SVN revision 9888 on a CPU Intel Core2 Duo T7200 2.0GHz with 1GB of RAM under Linux.

4.1 SMPI Experiments

The first case study is based upon two small C programs using MPI that are designed to measure the performance of our DPOR algorithm.

The first example, presented in Listing 4.3, shows an MPI program with N+1 processes. The process with rank 0 waits for a message from each of the other processes, while the other processes send their rank value to process 0. The property to verify is coded as the assertion at line 5 that checks for the incorrect

Listing 4.3. Example 1

```
1  if (rank == 0){
2    for (i=0; i < N-1; i++){
3      MPI_Recv(&val, MPI_ANY_SOURCE);
4    }
5    MC_assert(val == N);
6  } else {
7    MPI_Send(&rank, 0);
8  }
```

Listing 4.4. Example 2

```
1  if (rank % 3 == 0) {
2    MPI_Recv(&val, MPI_ANY_SOURCE);
3    MPI_Recv(&val, MPI_ANY_SOURCE);
4  } else {
5    MPI_Send(&rank, (rank / 3) * 3);
6  }
```

assumption of a fixed message receive order, where the last received message will be always from the process with rank N.

Table 1(a) shows the timing and the number of states visited before finding a violation of the assertion. In this case, the number of processes does not have a significant impact on the number of visited states because the error state appears early in the visiting order of the DFS. Still, using DPOR helps to reduce the number of visited states by more than 50% when compared to standard DFS.

Table 1. Timing, number of expanded states, and peak memory usage (a) to find the assertion violation in Listing 4.3; (b) for complete state space coverage of Listing 4.3 and (c) for complete state space coverage of Listing 4.4.

(a)

#P	DFS			DPOR		
	States	Time	Peak Mem	States	Time	Peak Mem
3	119	0.097 s	23952 kB	43	0.063 s	23952 kB
4	123	0.114 s	25008 kB	47	0.064 s	25024 kB
5	127	0.112 s	26096 kB	51	0.072 s	26080 kB

(b)

#P	DFS			DPOR		
	States	Time	Peak Mem	States	Time	Peak Mem
2	13	0.054 s	21904 kB	5	0.046 s	18784 kB
3	520	0.216 s	23472 kB	72	0.069 s	23472 kB
4	60893	19.076 s	24000 kB	3382	0.913 s	24016 kB
5	-	-	-	297171	84.271 s	25584 kB

(c)

#P	DFS			DPOR		
	States	Time	Peak Mem	States	Time	Peak Mem
3	520	0.247 s	23472 kB	72	0.074 s	23472 kB
6	>10560579	>1 h	-	1563	0.595 s	26128 kB
9	-	-	-	32874	14.118 s	29824 kB

Table 1(b) shows the effectiveness of DPOR for a complete state space exploration of the same program (without the assertion). Here, the use of DPOR reduces the number of visited states by an order of magnitude.

The second example, presented in Listing 4.4, shows the relevance of performing reduction dynamically. This time the number of processes in the system should be a multiple of 3. Every process with a rank that is a multiple of three will wait for a message from the next two processes, thus process 0 will receive from processes 1 and 2, process 3 from processes 4 and 5, etc. It is quite obvious that each group of three processes is independent from the others, but standard static reduction techniques would not be able to determine this. Again, no property is verified, as we try to compare the reductions obtained by DPOR.

Table 1(c) shows the experimental results for a complete exploration of the state space. In this case the DFS with 6 processes was interrupted after one hour, and up to that point it had visited 320 times more states than the complete state space exploration of the same program for 9 processes with DPOR enabled.

4.2 MSG Experiment: CHORD

As our second case study we consider an implementation of Chord [16] using the MSG communication API. Chord is a well known peer-to-peer lookup service, designed to be scalable, and to function even with nodes leaving and joining the system. This implementation was originally developed to study the performance of the SimGrid simulator.

The algorithm works in phases to stabilize the lookup information on every node that form a logical ring. During each phase, nodes exchange messages to update their knowledge about who left and joined the ring, and eventually converge to a consistent global vision.

Listing 4.5 shows a simplified version of Chord's main loop. In MSG, processes exchange *tasks* containing the messages defined by the user. Each node starts an asynchronous task receive communication (line 3), waiting for petitions from the other nodes to be served. If there is one (the condition at line 4 is true), a handler is called to reply with the appropriate answer using the same received task (line 5). Otherwise, if the delay for the next lookup table update has passed, it performs the update in four steps: request the information (lines 7-9), wait for the answer (lines 12-14), update the lookup tables (line 19), and notify changes to other nodes (line 22).

Running Chord in the simulator, we occasionally spotted an incorrect task reception in line 14 that led to an invalid memory read, producing a segmentation fault. Due to the scheduling produced by the simulator, the problem only appeared when running simulations with more than 90 nodes. Although we thus knew that the code contained a problem, we were unable to identify the cause of the error because of the size of the instances where it appeared and the amount of debugging information that these generated.

We decided to use SimGrid MC to further investigate the issue, exploring a scenario with just two nodes and checking the property $task == update\_task$ at line 15 of listing 4.5. In a matter of seconds we were able to trigger the bug and

Listing 4.5. Main loop of CHORD (simplified).

```
1  while(1) {
2    if (!rcv_comm)
3      rcv_comm = MSG_task_irecv(&task);
4    if (MSG_comm_test(rcv_comm)) {
5      handle(task);
6    } else if(time > next_update_time) {
7      /* Send update request task */
8      snd_comm = MSG_task_isend(&update_task);
9      MSG_task_wait(snd_comm);
10
11     /* Receive the answer */
12     if(rcv_comm == NULL)
13       rcv_comm = MSG_task_irecv(&task);
14     MSG_task_wait(rcv_comm);
15
16     MC_assert(task == update_task); /* <-- Assertion verified by the MC */
17
18     /* Update tables with received task */
19     update_tables(task);
20
21     /* Notify some nodes of changes */
22     notify();
23   } else {
24     sleep(5);
25   }
26 }
```

could understand the source of the problem by examining the counter-example trace, which appears in listing 4.6. It should be read top-down and the events of each node are tabulated for clarity. The Notify task sent by node 1 in line 22 of listing 4.5 is incorrectly taken by node 2 at line 14 as the answer to the update request sent by it line 8. This is due to an implementation error in the line 12: the code reuses the variable *recv_comm*, incorrectly assuming this to be safe because of the guard of that branch, but in fact the condition may change after the guard is evaluated.

Listing 4.6. Counter-example

```
 #line          Node 1                    #line          Node 2

 3: rcv_comm = MSG_task_irecv(&task)
 4: MSG_comm_test(rcv_comm) == FALSE
 8: snd_comm =
      MSG_MSG_task_isend(&update_task)
                                           3: rcv_comm = MSG_task_irecv(&task)
 9: MSG_task_wait(snd_comm)
                                           4: MSG_comm_test(rcv_comm) == TRUE
                                           5: handle(task)
                                           3: rcv_comm = MSG_task_irecv(&task)
                                           4: MSG_comm_test(rcv_comm) == FALSE
14: MSG_task_wait(recv_comm)
22: Notify()
 3: rcv_comm = MSG_task_irecv(&task)
                                           8: snd_comm =
                                                MSG_MSG_task_isend(&update_task)
                                           9: MSG_task_wait(snd_comm)
                                          14: MSG_task_wait(recv_comm)
```

The Chord implementation that was verified has 563 lines of code, and the model checker found the bug after visiting just 478 states (in 0.280 s) using DPOR; without DPOR it had to compute 15600 states (requiring 24 s) before finding the error trace. Both runs had an approximate peak memory usage of 72 MB, measured with the `/usr/bin/time` program provided by the operating system.

5 Conclusions and Future Work

We have presented SimGrid MC, a model checker for distributed C programs that may use one of three different communication APIs. Like similar tools, SimGrid MC is based on the idea of stateless model checking, which avoids computing and storing the process state at interruptions, and relies on dynamic partial order reduction in order to make verification scale to realistic programs. One originality of SimGrid MC is that it is firmly integrated with the pre-existing simulation framework provided by SimGrid [11], allowing programmers to use the same code and the same platform for verification and for performance evaluation. Another specificity is the support for multiple communication APIs. We have implemented sensibly different APIs in terms of a small set of elementary primitives, for which we could provide a formal specification together with independence theorems with reasonable effort, rather than formalize three complete communication APIs. We have been pleasantly surprised by the fact that this approach has not compromised the degree of reductions that we obtain, which are roughly on a par with those reported in [10] for a DPOR algorithm specific to MPI.

The integration of the model checker in the existing SimGrid platform has been conceptually simple, because simulation and model checking share core functionality such as the virtualization of the execution environment and the ability to execute and interrupt user processes. However, model checking tries to explore all possible schedules, whereas simulation first generates a schedule that it then enforces for all processes. SimGrid benefitted from the development of SimGrid MC in that it led to a better modularization and reorganization of the existing code. The deep understanding of the execution semantics gained during this work lets us envision an efficient parallel simulation kernel in future work.

SimGrid MC is currently restricted to the verification of safety properties such as assertion violations or the detection of deadlock states. The verification of liveness properties would require us to detect cycles, which is currently impossible due to the stateless approach. For similar reasons, state exploration is limited by a (user-definable) search bound. We intend to investigate hybrid approaches between stateful and stateless model checking that would let us overcome these limitations.

Acknowledgment

The helpful comments of the anonymous reviewers are gratefully acknowledged. This work is partially supported by the ANR project USS SimGrid (08-ANR-SEGI-022).

References

1. Hennessy, M.: Algebraic Theory of Processes. MIT Press (1988)
2. Reisig, W.: A Primer in Petri Net Design. Springer (1992)
3. Holzmann, G.J.: The model checker Spin. IEEE Trans. Softw. Eng. **23**(5) (1997) 279–295
4. Lamport, L.: Specifying Systems. Addison-Wesley, Boston, Mass. (2002)
5. Visser, W., Havelund, K.: Model checking programs. In: Automated Software Engineering Journal. (2000) 3–12
6. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: language support for building distributed systems. In: Proc. ACM SIGPLAN Conf. Programming language design and implementation (PLDI 2007), San Diego, CA, USA, ACM (2007) 179–188
7. Musuvathi, M., Qadeer, S.: Fair stateless model checking. In: Proc. ACM SIGPLAN Conf. Programming language design and implementation (PLDI 2008), Tucson, AZ, USA, ACM (2008) 362–371
8. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: A pragmatic approach to model checking real code. In: Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI 2002). (2002)
9. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. SIGPLAN Not. **40**(1) (2005) 110–121
10. Palmer, R., Gopalakrishnan, G., Kirby, R.M.: Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In: Proc. ACM Wsh. Parallel and distributed systems: testing and debugging (PADTAD 2007), London, UK, ACM (2007) 43–53
11. Casanova, H., Legrand, A., Quinson, M.: SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In: 10th IEEE International Conference on Computer Modeling and Simulation. (March 2008)
12. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles of programming languages (POPL 1997), Paris, France, ACM (1997) 174–186
13. Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R.M., Thakur, R.: Formal verification of practical MPI programs. SIGPLAN Not. **44**(4) (2009) 261–270
14. Iosif, R.: Exploiting heap symmetries in explicit-state model checking of software. In: Proc. 16th IEEE Intl. Conf. Automated software engineering (ASE 2001), Washington, DC, USA, IEEE Computer Society (2001) 254–261
15. Rosa, C., Merz, S., Quinson, M.: A simple model of communication APIs – Application to dynamic partial-order reduction. In: 10th Intl. Wsh. Automated Verification of Critical Systems, Düsseldorf, Germany (2010) 137–152
16. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM Comput. Commun. Rev. **31** (August 2001) 149–160