Model-checking Distributed Applications with GRAS

Cristian Rosa, Martin Quinson, Stephan Merz

Model-checking Distributed Applications with GRAS

Cristian Rosa
Nancy University / LORIA
Cristian.Rosa@loria.fr

Martin Quinson
Nancy University / LORIA
Martin.Quinson@loria.fr

Stephan Merz
INRIA Nancy-Grand Est / LORIA
Stephan.Merz@loria.fr

ABSTRACT

In this paper, we present our work to add a model-checker to the GRAS framework. This tool provides a development environment allowing debug real applications within the simulator before deploying them on real platforms. We discuss the main difficulties to integrate such a model-checker, and present our approach to solve them, as well as a prototype implementation.

Keywords

Model Checking, Distributed Application Development.

1. INTRODUCTION

Distributed systems are in the mainstream of information technology. They form the core of many important businesses and scientific applications, as multiple distributed units may work simultaneously at multiple parts of a problem. The first motivation to build such systems is to cope with intrinsically distributed applications, such as banks, airline reservation systems or collaborative work over the Internet. Another reason for the success of these systems is that they allow to leverage the computational power of several machines, to speedup the applications or to deal with bigger instances of the problems. This is of particular importance for example in financial or scientific simulations.

Designing and debugging such applications is particularly difficult because the programmer has to cope with issues shared by any parallel program (either concurrent or distributed), such as race conditions, deadlocks, live locks, as well as some other issues specific to distributed settings, such as asynchronous communications and the difficulty to define a global state for the system.

Several approaches were proposed to develop and debug such application. The most natural one is the direct execution of the program over a representative testbed. Unfortunately, this approach suffers from several drawbacks. First, it is very time consuming since the programmer may face issues in the testbed system not directly related to its application (such as loss of connexion). Then, its is intrinsically limited to the testbed the programmer has access to, making it difficult to assess the performance of the application on another kind of platform. These reasons explain why most of the time, distributed applications are only tested on a very limited set of conditions before being used in production.

Another approach is to rely on simulation for fine tuning and debugging distributed applications. It offers the ability to test their code in more comprehensive (even if not exhaustive) test campaigns. The main drawback of this approach is that in most cases, the prototype tested onto the simulator has to be completely rewritten to produce the application. This manual translation process unfortunately leaves space to bug introduction.

GRAS (Grid Reality and Simulation) [7] is an API for the implementation of distributed algorithms on heterogeneous platforms, in particular for the Grid infrastructure and P2P networks. It can be used in two modes: for the simulation of programs (via SimGrid – [1]) as well as for native execution on real platforms. It thus provides a *"write once, deploy twice"* approach, where the user can test their code within the simulator and then deploy them on real platform without any change to the source code.

In recent years several formal methods techniques have been proposed for the verification of concurrent and distributed software. Model checking is one of the most prominent of these techniques, mainly because of its simplicity and its ability to be run automatically. Model-checking is used to establish whether a model meets a given specification, and proceeds by doing an automatic and exhaustive exploration of the model's state space. Generally the models are written in an abstract specification language like Promela for Spin [2], or TLA+ for TLC [4], but there is also widespread interest in applying model checking to source code; this is the case of MACE [3], CHESS [6] or CMC [5].

In this work we explore the possibility of adding a model-checker mode to the GRAS framework. We believe that having such a tool can be extremely beneficial for the development process. It is known that many bugs are introduced during the translation to code, even after the model was shown correct. Also, many bugs are only visible at implementation level because models are usually too abstract to capture them. As many authors suggest [8], these are strong arguments to also model-check the implementation produced by programmers in addition of formal models built out of them. Working at C level allows non specialists to use this methodology without requiring them to learn a specification language, which is an impediment of many model checkers.

The rest of this paper is organized as follows: Section 2 summarizes the main issues to solve in order to develop a model-

checker for C programs. Section 3 presents our approach to solve these issues and introduces our first prototype, while Section 4 concludes this paper and discusses the envisioned future work.

2. MODEL-CHECKING DISTRIBUTED APPLICATIONS

In this section, we identify four main issues to solve in order to implement a model-checker at application level.

Capturing the global state of the system. As mentioned in previous section, model-checking proceeds by exploring all the possible states of the model. It is thus mandatory to capture this state, however this problem is known to be very difficult in distributed settings. It received a lot of attention in the domain, and some algorithms to globally checkpoint a distributed application exist, but they remain extremely complex and their implementations are error prone.

Identifying decision points. Another problem to solve is to determine which are the decision points that lead to new states. Every state is determined by the trace of execution of all the interacting processes that lead to it. Thus in order to explore all the state space, the model-checker must execute all the possible interleavings of the atomic instructions blocks. Each point between two atomic instruction blocks is a decision point. In concurrent settings (ie, applications where memory is shared among several processes), the biggest execution unit that can be considered atomic is a single instruction. This results in an extremely large amount of possible execution interleavings, known as state explosion.

Rewinding the application. In order to explore different execution traces, the model-checker has to rewind the application to a previous execution point, and restart it from there taking a different execution path. This rollback operation is performed each time the model-checker reaches a decision point, and it depends on variables like the actual depth and the number of processes that are ready to run. This functionality requires the model-checker to be able to save and restore the state at each point.

Expressing properties. The last obstacle to cope with is how to express properties over C programs. When model checking is applied to abstract models, properties are generally expressed in temporal logic such as LTL, and they can reason about all the elements on it. However, when dealing with C programs, properties are expressed as C functions, which are limited to reason about the objects in scope. This tends to put most variable describing the state of the process as globals, even if it violates the data encapsulation.

3. OUR APPROACH

In this section, we detail how we plan to address the issues identified in the previous section.

As GRAS, our prototype uses the SimGrid simulation framework to emulate the hosts and networks, and it really executes the C code of the application. This solves the issue of ***capturing the global state*** of the distributed algorithm, because from the SimGrid point of view, the entire system is running in the same address space. When operating in *mc* mode, each simulated process is executed in its own isolated heap. For that, we implemented a modified version of malloc that performs the allocations into a private memory region. This isolation greatly simplifies the control over the application's state.

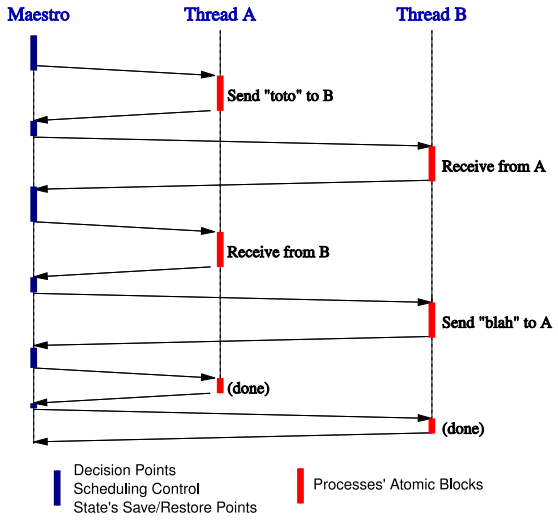

Figure 1: Synchronous Execution Model.

Since our prototype targets distributed systems, the issue of ***identifying decision points*** is somehow easier than for arbitrary parallel code. Indeed, each parallel process can be thought as executed in a different address space, thus all the communication among them is performed by message exchange, making possible to consider atomic the code executed between the send and receive of messages.

In SimGrid, the entire simulation runs as a single system process in the host machine, and each simulated process runs as a user thread. The model-checker runs as a special *maestro* thread, and is responsible of deciding the scheduling. The execution of the simulation is synchronous, only one thread is in running at any time. When one thread wants to communicate with another, it setups a message and yields back the control to maestro, which decides what happen next. Figure 1 depicts the execution model. Since every simulated process is stopped when the maestro runs, it gives us the ability to save the state (heap, registers, and stack) of the simulated threads, and ***rewind the application*** to a previous point if requested by the model-checking algorithm.

In the version presented as our prototype, ***properties are expressed*** as boolean functions that are evaluated at each execution. If any of them evaluates to false in an execution path, the model checking process is aborted and the execution trace that lead to the invalid state is dumped. This provides a nice way to assertions. Our prototype still misses

a way to declare classical temporal properties, and we plan to add them in the near future.

As said in the previous section, since properties can only reason on the variable defined in their scope, one usually rely on global variables to express the process state. Unfortunately, the use of classical globals is forbidden to the users in GRAS. This is because the implementation of such construct is troublesome depending on the way the code is executed. If the user wanted their global to be visible from the whole application, this would be almost impossible to implement when running in distributed settings over a real platform. If s/he wanted the global to be only visible from the process which declare it, this would be difficult to implement when folding the system as a multi-threaded application. That is why GRAS request the user to declare the set of variables defining the process state in a specific way, so that they can be dealt with properly both in distributed and in folded settings. We plan to use this feature to allow the user to express global properties over the state of each processes in future work.

4. CONCLUSION AND FUTURE WORK

In this paper, we introduced an extension of the GRAS framework aiming at allowing the model-checking of distributed applications. This new mode comes in addition to the existing simulation and real execution ones already provided by the tool. We detailed the main issues we envision to build such a model-checker, and provided some insight on how we solved them.

The prototype we built is already able to model-check unmodified GRAS applications written in C, and was successfully used to identify an error in an intentionally crafted example. This work in progress can be retrieved from the svn source repository of the SimGrid project[1].

This prototype however still requires some more work. At interface level, we first plan to allow the user to express temporal properties using a specifically crafted formalism. Then, we plan to ease the expression of properties over every processes using the GRAS virtualization mechanism presented in Section 3. We also plan to better integrate this new execution mode into the GRAS framework, provide some examples of use and improve the user documentation.

Internally, the biggest obstacle to cope with when model-checking is the famous state explosion problem, that is exacerbated when the technique is applied to software. This is because of the level of detail in a real software is naturally higher than in abstract models: new memory can be allocated, threads can be created or finished, and different ordering of interleaving concurrent threads generates different states.

Many techniques have been developed to reduce the amount of states to explore, and a great number of them tries to profit from the symmetries in the structure of the state. This is the case of *partial-order reductions*, that exploits the fact that it is not necessary to consider the execution order of independent processes. Another technique in this category is *heap canonicalization*, that uses the fact that many different heaps might be equivalent from the program point of view. It is possible to obtain a unique representative for each class and reduce the amount of states to consider as different. We plan to add these optimizations to our tool as part our future research.

Despite all the effort invested into reducing the state space, the scale limiting factor still is the size. We explore the ability to distribute the model-checking process among several computers to allow the execution of larger instances of the problem. This mechanism could reveal useful any SimGrid users, since it would also allow to run distributed simulations, thus removing the main scalability issue of the main framework.

[1] `http://gforge.inria.fr/scm/?group_id=12`

5. REFERENCES

[1] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, March 2008.

[2] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[3] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 179–188, New York, NY, USA, 2007. ACM.

[4] Leslie Lamport. *Specifying Systems.* Addison-Wesley, Boston, Mass., 2002. `http://research.microsoft.com/users/lamport/tla/tla.html`.

[5] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[6] Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 362–371, New York, NY, USA, 2008. ACM.

[7] Martin Quinson. GRAS: a Research and Development Framework for Grid and P2P Infrastructures. In *The 18th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2006.

[8] Willem Visser and Klaus Havelund. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12, 2000.