VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software

Patrice Godefroid

Bell Laboratories
Lucent Technologies
1000 E. Warrenville Road
Naperville, IL 60566, U.S.A.
god@bell-labs.com
http://www.bell-labs.com/~god

Abstract. VeriSoft is a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary code written in full-fledged programming languages such as C or C++. It can automatically detect coordination problems between concurrent processes. Specifically, VeriSoft searches the state space of the system for deadlocks, livelocks, divergences, and violations of user-specified assertions. An interactive graphical simulator/debugger is also available for following the execution of all the processes of the concurrent system.

1 Introduction

State-space exploration techniques are increasingly being used for analyzing the correctness of *concurrent reactive systems*. These techniques consist of exploring a directed graph, called the *state space*, representing the combined behavior of all concurrent components in a system. In the case of software systems, existing state-space exploration tools can compute automatically a state space from an abstract description of such a system, specified in a *modeling language*. Examples of such tools are CAESAR [FGM+92], COSPAN [HK90], CWB [CPS93], MURPHI [DDHY92], SMV [McM93], SPIN [Hol91], and VFSMvalid [FHS95], among others. In many cases, analyses of complex concurrent systems using state-space exploration techniques were able to reveal quite subtle design errors (for instance, see [Rud92, CGH+93, BG96]).

VeriSoft extends the previous results by being able to directly analyze the implementation of a concurrent reactive software system, rather than a hand-written model of it. Specifically, VeriSoft is a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary code written in full-fledged programming languages such as C or C++. It can automatically detect coordination problems between concurrent processes. An interactive graphical simulator/debugger is also available for following the execution of all the processes of the system.

In the next section, we define the state space of a concurrent system composed of processes executing arbitrary code. Then, we present the properties that can be checked with VeriSoft. We conclude with a brief presentation of the tool itself.

2 Concurrent Systems and Dynamic Semantics

We consider a concurrent system composed of a finite set $\mathcal{P}$ of *processes* and a finite set of *communication objects.* Each process $P_i \in \mathcal{P}$ executes a sequence of *operations*, that is described in a sequential program written in a programming language such as C or C++ for instance. Such programs are deterministic: every execution of the program on the same data performs the same sequence of operations. We assume that processes communicate with each other by performing operations on communication objects. Examples of communication objects are shared variables, semaphores, and FIFO buffers. At any time, at most one operation can be performed on a given communication object (operations on a same communication object are mutually exclusive). Operations on communication objects are called *visible operations*, while other operations are by default called *invisible.* The execution of an operation is said to be *blocking* if it cannot be completed. We assume that only executions of visible operations may be blocking.

The concurrent system is said to be in a *global state* when the next operation to be executed by every process in the system is a visible operation. We assume that every process in the system always eventually attempts to execute a visible operation. This implies that initially, after the creation of all the processes of the system, the system may reach a first and unique global state s_0, called the *initial global state* of the system. We define a *transition* as a visible operation followed by a finite sequence of invisible operations performed by a single process. A transition whose visible operation is blocking in a global state s is said to be *disabled* in s. Otherwise, the transition is said to be *enabled* in s. A transition t that is enabled in a global state s can be *executed* from s. Once the execution of t from s is completed, the system reaches a global state s', called the *successor* of s by t. The *state space* of the concurrent system is composed of the global states that are reachable from the initial global state s_0, and of the transitions that are possible between these.

All operations on objects are deterministic, except one special operation "VS_toss", which is used to express a valuable feature of modeling languages, not found in programming languages: *nondeterminism.* Indeed, we consider here closed concurrent systems, where the environment of one process is formed by the other processes in the system. This implies that, in the case of a single "open" reactive system, the environment in which this system operates has to be represented, possibly using other processes. In practice, a complete representation of such an environment may not be available, or may be very complex. It is then convenient to use a simplified representation (software stub) for the environment to simulate its observable behavior. Another reason for providing a specific representation of the environment is to test the system under specific external constraints (test driver). The operation VS_toss takes as argument a positive integer n, and returns an integer in $[0, n]$. The operation is visible and nondeterministic: the execution of a transition starting with VS_toss(n) may yield up to $n + 1$ different successor states, corresponding to different values returned by VS_toss.

3 Properties

In [God97], it is shown that *deadlocks* and *assertion violations* can be detected by exploring only the global states of a concurrent system as defined in the previous section. Deadlocks are states where the execution of the next operation of every process in the system is blocking. Assertions can be specified by the user with the special operation "VS_assert". This operation can be inserted in the code of any process, and is considered visible. It takes as its argument a boolean expression that can test and compare the value of variables and data structures local to the process. When "VS_assert(expression)" is executed, the expression is evaluated. If the expression evaluates to false, the assertion is said to be *violated*.

In addition to deadlocks and assertion violations, VeriSoft also checks for *divergences* and *livelocks*. A "divergence" occurs when a process does not attempt to execute any visible operation for more than a given (user-specified) amount of time, while a "livelock" occurs when a process has no enabled transition during a sequence of more than a given (user-specified) number of successive global states. Note that these definitions of divergence and livelock differ from the standard definitions for these notions, which correspond to *liveness* properties, i.e., properties that can only be violated by *infinite* sequences of operations or transitions [Lam77, MP92]. In contrast, our notions of divergence and livelock can be violated by *finite* sequences of operations or transitions, and therefore are actually *safety* properties. (See [God97] for details.)

4 Systematic State-Space Exploration using VeriSoft

VeriSoft is a tool for systematically exploring the state spaces of concurrent systems as defined in Section 2. In a nutshell, every process of the concurrent system to be analyzed is mapped to a UNIX process. The execution of the system processes is controlled by an external process, called the *scheduler*. This process observes the visible operations performed by processes inside the system, and can suspend their execution. By resuming the execution of (the next visible operation of) one selected system process in a global state, the scheduler can explore one transition between two global states in the state space of the concurrent system. By reinitializing the system, the scheduler can explore alternative paths in the state space.

The scheduler also contains an implementation of a new search algorithm, introduced in [God97], that makes it possible to systematically and efficiently explore the state spaces of such systems without storing any intermediate states in memory. This algorithm is built upon existing state-space pruning techniques known as partial-order methods [God96]. For finite acyclic state spaces, this algorithm is guaranteed to terminate and can be used for detecting deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results. In practice, VeriSoft can be used for systematically and efficiently testing the correctness of any concurrent system, whether its state space is acyclic or not.

VeriSoft searches the state spaces of concurrent systems for errors of the types listed in Section 3. When an error is detected, a scenario leading to the error state is exhibited to the user. An interactive graphical simulator/debugger is also available for replaying scenarios and following their executions at the instruction or procedure/fonction level. Values ot variables of each process can be examined interactively. In manual-simulation mode, the user can also explore any path in the state space of the system with the same set of debugging tools.

VeriSoft has been tested on various examples of concurrent reactive C programs to demonstrate the practicability of our approach. As an example, VeriSoft successfully discovered a previously unknown error in a concurrent 2500-line C program controlling robots operating in an unpredictable environment. These encouraging experimental results bode well for the applicability of VeriSoft to the analysis of actual software products. Several such applications are currently being investigated in cooperation with switching-software development and testing organizations in Lucent Technologies. Additional information on VeriSoft is (and will be) available at `http://www.bell-labs.com/~god`.

References

[BG96] B. Boigelot and P. Godefroid. Model checking in practice: An analysis of the ACCESS.bus protocol using SPIN. In *Proceedings of Formal Methods Europe'96*, volume 1051 of *Lecture Notes in Computer Science*, pages 465–478, Oxford, March 1996. Springer-Verlag.

[CGH+93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and Their Apllications*. North-Holland, 1993.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 1(15):36–72, 1993.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, Cambridge, MA, October 1992. IEEE Computer Society.

[FGM+92] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *Proc. of the 14th International Conference on Software Engineering ICSE'14*, Melbourne, Australia, May 1992. ACM.

[FHS95] A. R. Flora-Holmquist and M. Staskauskas. Formal validation of virtual finite state machines. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, pages 122–129, Boca Raton, April 1995.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.

[God97] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.

[HK90] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 1990.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[Rud92] H. Rudin. Protocol development success stories: Part I. In *Proc. 12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, June 1992. North-Holland.