Special section on SPIN

SPIN model checking: an introduction

Gerard Holzmann¹, Eli Najm², Ahmed Serhrouchni²

¹ Bell Laboratories, MH 2C-521, 700 Mountain Avenue, Murray Hill, New Jersey 07974, USA;
E-mail: gerard@research.bell-labs.com
² Ecole Nationale Supérieure des Télécommunications, 46, rue Barrault, 75013 - Paris, France

1 Introduction

A long-standing and elusive problem in software engineering is to devise reliable means that would allow us to check the correctness of distributed systems code mechanically. Writing reliable distributed code is notoriously difficult; locating the inevitable bugs in such code is therefore important. As is well-known and often repeated, traditional testing methods are of little use in this context, because the most pernicious bugs typically depend on subtle race conditions that produce peculiar and unexpected interleavings of events.

Well-known are the deadlock and starvation problems that plagued the designers of the first distributed systems code in the 1960s and 1970s (see for instance [16] p.155). In simple cases, a small set of strictly enforced rules can preserve the sanity in a system. One such rule is the requirement that frequently used resources in an operating system can only be allocated in a fixed order, to prevent circular waiting. But the simple rules only cover the known problems. Each new system builds a new context, with its own peculiarities and hazards. This is illustrated by the well-publicized description of the hangup problem in the control software of the Mars Pathfinder a few years ago [11]. In retrospect, the hangup scenario could be understood in very simple terms, yet it was missed even in the long and unusually thorough (but traditional) software testing process that had been used.

2 Software model checking

The first attempts to develop automated techniques to analyze simple versions of distributed systems, starting in the late 1970s, focused almost exclusively on data communication protocols. The reason is that protocol descriptions were from the beginning typically expressed in graph-like or statemachne-like notations, rather than in general programming languages, which favored experiments with the construction of simple automated tools. As far as we can tell, the first person to build a general purpose verification system for protocols was Jan Hajek, then working at the University of Eindhoven in The Netherlands [4]. Hajek published little about the algorithms that he used to build his Approver tool, though, so we known little more than that it used graph algorithms and some heuristics to identify undesirable reach-able states (nodes in the system graph) and undesir-able executions sequences (finite and infinite paths in the graph). Andy Tanenbaum confirms [15] that Hajek used his system to check the correctness of the protocols that appeared in the first edition of the now classic Computer Networks book [14].

In the next few years many others followed, most of whom developed specialized formalisms for expressing protocol behaviors, but generally performing a comparable type of analysis. In our own group at Bell Labs we built a first version of a reachability analyzer in 1980 and successfully applied it to a local model of a telephone switch to find design errors [5]. That system, called PAN, is the earliest ancestor of today’s SPIN model checker, but other than that it used an on-the-fly verification technique rooted in a depth-first exploration of a system’s graph, of course, it no longer has much in common with today’s SPIN.

Over the years, the tools, like PAN, that were built for reachability analysis of communication protocols evolved into full-fledged automata-based model checking systems addressing the more general area of distributed systems software. The origin and evolution of tools like SPIN is markedly different from that of the classic logic model checkers pioneered by Ed Clarke and his students at CMU in the USA, e.g., [1] and Joseph Sifakis and students at Grenoble University in France, e.g., [10]. Initially, sys-
tems such as PAN were restricted to the verification of safety properties and a very small class of liveness properties (such as proving the absence of certain types of non-progress cycles). We made no attempt to extend the logic beyond this small class of correctness properties for two basic reasons:

1. The smaller class appeared sufficient at the time for the types of applications that we applied the reachability analyzers to (i.e., relatively simple protocol models and models of mutual exclusion algorithms).
2. The computational expense of even these simple types of checks could easily swamp the resources of the best computers in the early 1980s. (Several orders of magnitude slower and smaller than the average desktop PC today.)

Any attempt to increase the expressive power of the logic immediately translated into huge computational expenses that made it all but impossible to apply a tool to anything other than the most trivial exercises, as the first builders of full-fledged logic model checkers also soon discovered. By virtue of our pragmatic restrictions, we were able to analyze respectably sized problems with the PAN system, reaching several millions of reachable states on machines that would be considered unusable even as toys today. Not many other tools could match this behavior.

From then on the advances in performance, especially hardware, changed all the rules in the late 1980s. In 1989 we extended the expressive power for the SPIN system to that of omega-regular properties (a set that includes Linear Temporal Logic as a special case) to take full advantage of the more powerful processors. Since then, further algorithmic improvements have been made (e.g., search reduction techniques and state compression methods) and the hardware performance improvements have continued. The result is that today most of us have access to a range of very powerful model checking engines for software verification. The ones we are interested in are based, like the first reachability analyzers from the early 1980s, on automata and graph theory, not on logic systems. SPIN is generally considered to be the leading example of a highly efficient model checker in this group.

This evolution of the automata-based systems may be contrasted with that of classic logic-based model checkers with, as its leading example, the well-known SMV model checker developed at CMU [8]. Most people are not aware that systems like SMV and SPIN are based on different types of theories and have evolved quite independently over the last two decades. The most notable difference between these tools is the type of application for which they were built and optimized: SMV is primarily built for hardware verification [8]; SPIN is built for software verification [6]. Internally the tools are almost incomparable. SMV is based on logic, uses BDD encodings for the transition relations, and it applies a basically breadth-first graph exploration technique to solve the model checking problem for the branching time logic CTL; SPIN is based on automata theory, it uses an explicit depth-first search for omega regular properties, with as its most prominent subset the linear time logic LTL [9].

Due to the differences in their domains of application, logics, and modeling languages, it is hard to make accurate comparisons between these tools. Those who have tried have typically concluded that for comparable applications, the tools are very competitive, neither tool systematically outperforming the other, e.g., [2]. Despite this there are long-standing myths that hold that either LTL or CTL model checking would necessarily be more efficient, or that CTL or LTL would be a more adequate logic to express correctness properties. In practice, both tools are virtually undefeatable within their own domain of application. For SPIN this is the verification of the interactions of asynchronously executing concurrent processes in a distributed system.

3 SPIN based model checking in a nutshell

SPIN implements what has become known as the Vardi-Wolper framework for automata-theoretic verification [12]. Curiously, it was not a deliberate plan to build SPIN to conform to this methodology, but more a discovery that it could easily do so after the tool was almost completed. SPIN and most of its predecessors are implementations of standard automata-based reachability analyzers, with primary emphasis on performance. When the implementation of SPIN was nearing completion in early 1989, it had support for two new types of correctness claims, in addition to the by then standard support for proving absence of deadlock and absence of non-progress cycles. We had added two new types of claims. One was used to express positive correctness requirements (behaviors required to be present) and was named an always claim. The other was used for expressing negative correctness requirements (behaviors required to be absent) and was named a never claim. A former colleague at Bell Labs, Costas Courcoubetis, quickly noted that the functionality of the never claim could be explained in terms of Vardi-Wolper’s automata-theoretic method and sufficed for expressing LTL, and even the richer set of all omega regular properties in which LTL is contained. After a short discussion we decided to omit the experimental always claims, and to adopt the Vardi-Wolper theory as a basis for the model checker. By doing so, SPIN succeeded in combining an efficient implementation of a classic reachability analyzer with a firm and well-understood theory for LTL model checking. The combination is likely responsible for the remarkable appeal of SPIN in both academia and industry.

SPIN accepts specifications written in a meta-language named PROMELA, (short for Process Meta Language). The semantics of the language are carefully chosen to make it impossible to define anything other than models for which the reachable system states can, in principle,