State-Space Caching Revisited

PATRICE GODEFROID
Université de Liège, Institut Montefiore B28, 4000 Liège Sart-Tilman, Belgium

GERARD J. HOLZMANN
AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

DIDIER PIROTTIN
Université de Liège, Institut Montefiore B28, 4000 Liège Sart-Tilman, Belgium

Received November 22, 1993; Revised June 3, 1994

Abstract. State-space caching is a verification technique for finite-state concurrent systems. It performs an exhaustive exploration of the state space of the system being checked while storing only all states of just one execution sequence plus as many other previously visited states as available memory allows. So far, this technique has been of little practical significance: it allows one to reduce memory usage by only two to three times, before an unacceptable blow-up of the run-time overhead sets in. The explosion of the run-time requirements is due to redundant multiple explorations of unstored parts of the state space. Indeed, almost all states in the state space of concurrent systems are typically reached several times during the search.

In this paper, we present a method to tackle the main cause of this prohibitive state matching: the exploration of all possible interleavings of concurrent executions of the system, which all lead to the same state. Then, we show that, in many cases, with this method, most reachable states are visited only once during state-space exploration. This enables one not to store most of the states that have already been visited without incurring too much redundant explorations of parts of the state space, and makes therefore state-space caching a much more attractive verification method. As an example, we were able to completely explore a state space of 250,000 states while storing simultaneously no more than 500 states and with only a three-fold increase of the run-time requirements.

Keywords: Verification, concurrency, state explosion, model-checking, caching

1. Introduction

The effectiveness of state-space exploration techniques for debugging and proving correct concurrent reactive systems is increasingly becoming established as tools are being developed. The number of “success stories” about applying these techniques to industrial-size systems keeps growing (e.g., see [18]). The reason why these techniques are so successful is mainly due to their simplicity: they are easy to understand, easy to implement and, last but not least, easy to use: they are fully automatic. Moreover, the range of properties that they can verify has been substantially broadened in the last decade thanks to the development of model-checking methods for various temporal logics. The only real limit of state-space exploration verification techniques is the often excessive size of the state space.

More precisely, memory is the main limiting factor of most conventional reachability analysis algorithms. Given a finite-state system, these verification algorithms perform an
exhaustive exploration of the state space of the system being checked in order to prove or disprove properties of the system. This exploration amounts to simulating all possible behaviors the system can have from its initial state and storing all reachable states. To avoid significant run-time penalties for disk-access, reachable states can only be stored in a randomly accessed memory, i.e., in the main memory available in the computer where the algorithm is executed. Therefore the applicability of these verification algorithms is limited by the amount of main memory available. Typically, it only takes a few minutes of run-time to fill up the whole main memory of a classical computer.

During the search, once states have been visited they are stored. Storing states avoids redundant explorations of parts of the state space. If a stored state is encountered again later in the search, it is not necessary to revisit all its successors. It is worth noticing that states that are reached only once during the search do not need to be stored. Storing them or not would not change anything about the time requirements of the method. Of course, it would be preferable not to store them in order to decrease the memory requirements, but with a conventional algorithm it is virtually impossible to predict if a given state will be visited once or more than once.

Typically, almost all states in the state space of concurrent systems are reached several times during the search. There are two causes for this:

1. From the initial state, the explorations of all interleavings of a single finite concurrent execution of the system always lead to the same state. This state will thus be visited several times because of all these interleavings.
2. From the initial state, explorations of different finite concurrent executions may lead to the same state.

In this paper, we introduce a technique to avoid the effects of the first cause given above. Then we study the impact of this technique on real-protocol state spaces. In many cases, when using this method, most of the states are reached only once during the search.

Sadly, it is not possible to determine which states are visited only once before the search is completed. However, the risk of double work when not storing an already visited state becomes very small since the probability that this state will be visited again later during the search becomes very small. This enables one not to store most of the states that have already been visited without incurring too much redundant explorations of parts of the state space. The memory requirements can thus strongly decrease without seriously increasing the time requirements. This makes possible the complete exploration of very large state spaces (several tens of million states) in a reasonable time (a few hours). In most cases, time, not memory, is the main limiting factor of this verification technique.

In the next Section, we recall the principles of state-space caching and present some results obtained with this method for the verification of four real protocols. Then we show how this verification method can be substantially improved by the use of “sleep sets”. Sleep sets were introduced in [5, 7]. In Section 3, we recall the basic idea behind sleep sets. Then, we present a new simple version of the sleep set scheme. We study properties of sleep sets and prove two theorems. Section 4 presents and compares the results obtained with the state-space caching method with and without the use of sleep sets. In Section 5, some suggestions to further improve the effectiveness of the method are investigated.