External Memory Search for Verification of Multi-threaded C++ Programs

Stefan Edelkamp, Shahid Jabbar, Dino Midzic, Daniel Rikowski, Damian Sulewski

With the advent of multi-core processors, the need for development of multi-threaded softwares has become indispensable. Verification of multi-threaded programs, particularly those that involve sharing of memory resources, poses a greater challenge than their sequential counterparts. A certain class of software model checking problems can be transformed to AI search problems in graphs. Search algorithms, such as DFS, BFS, A*, etc. can then be applied to find an erroneous program location in a given program. Unfortunately, verification of softwares is very memory intensive. In this paper, we equip the search algorithms used in model checking a C++ program, with a controlled access to secondary memory such as hard disk. We exploit the concept of a *signature* of a state that allows the full state vector to stay on secondary memory. The extended search algorithms are now able to solve larger problems that were unsolvable due to the memory bottleneck.

1 Introduction

The dependence of humans on software is increasing at an astonishing rate. With the advent of multi-core processors, the need for development of multi-threaded softwares has become indispensable. Similarly, the necessity for verified softwares in critical domains has brought new challenges to the model checking community. Verification of multi-threaded softwares poses a greater challenge than a sequential program; inconsistencies in the shared data can result in serious errors at run-time.

For verification of safety-critical softwares, a common practice is that designers would normally write the system specification in a formal language like *Z* and (manually) prove formal properties over that specification. When the development process gradually shifts toward the implementation phase, the specification must be completely rewritten in the actual programming language (usually C++). With this approach, software designers are faced with two major problems: additional overhead for re-formulating the same program logic in a different language and the introduction of new errors in the re-writing process that may falsify properties that hold in the formal specification.

Model checking is a formal verification method for state-based systems. It has been successfully applied in various fields including process engineering, hardware design and protocol verification. Recent applications of model checking technology deal with the verification of software implementations (rather than checking a formal specification). The advantage of this approach is manifold when compared to the modus operandi in the established software development cycle.

There are two main paradigms of program verification. The first one revolves around a static analysis of the program code and to reason on possible outcomes. Such methods have their roots in *abstract interpretation* and *data-flow analysis*. The second paradigm transforms the verification problem into a search problem in graphs. Starting from the initial vertex encoding the variables' values, execution of each program statement generates a new vertex in this graph. A repetitive execution of statements results in a search graph encoding the program flow. A program *fails* to comply a required property, if a vertex encoding the erroneous values is reached in that graph. The underlying search algorithm can be borrowed from Breadth-First Search (BFS), Depth-First Search (DFS) or Best-First Search. To construct a vertex of the search graph that encodes a program's state, software model checkers rely on the extension or implementation of architectures capable of interpreting machine code. These architectures include virtual machines [25] and debuggers [19].

A drawback to the state-based methods is the so-called *state space explosion* problem. Each vertex of the search graph must memorize the contents of all the allocated memory regions. As a consequence, the generated states may quickly exceed the available RAM. A second effect is the generation of states with very large sizes. As a solution to the former problem, many techniques have been proposed [3]. They include *partial-order reduction* that prunes the state space by exploiting the commutativity of the edges/transitions, *symmetry detection* exploits problem regularities on the state vector, while *abstraction methods* analyze approximated state spaces. The remedies for the latter problem include *binary state encoding* that allows to represent larger state sets through boolean functions and *bit-state hashing* that compresses the state vector down to a few bits, while failing to disambiguate some of the synonyms.

Recent advances in dealing with the state space explosion include the use of more sophisticated AI search algorithms that exploit *heuristics* to guide the search process toward the erroneous state. Such directed/guided model checking approaches have shown significant advances in the verification of communication protocols [7], mu-calculus [22], Java [9] and hardware [2].

But even with refined exploration strategies, model checking is bounded by the main memory resources. Several memory-limited algorithms have been developed, e.g. [8, 11,

Auszug aus: Künstliche Intelligenz, Heft 2/2008, ISSN 0933-1875, BöttcherIT Verlag, Bremen, **www.kuenstliche-intelligenz.de/order**