On partial state matching

Pavel Jančík¹ and Jan Kofroň¹

¹ Charles University, Faculty of Mathematics and Physics, Malostranske namesti 25, 118 00, Praha 1, Czech Republic.
URL: http://d3s.mff.cuni.cz

Abstract. During explicit software model checking, the tools spend a lot of time in state matching. This is implied not only by processing a huge number of states, but also by the fact that state representation is usually not small either. In this article, we present two dead variable analyses; applying them during the code-model-checking process results in size reduction of both state representation and explored state space itself. We implemented the analyses inside Java PathFinder and evaluate their impact in terms of memory and time reduction using several non-trivial benchmarks.

Keywords: Explicit model checking, dead variable analysis, optimization, performance.

1. Introduction

In explicit software model checking, the model checkers spend a lot of time in the state matching process. During the state space traversal, state matching identifies equivalent states to avoid multiple exploration of the same parts of the state space. This usually implies computing a state representation for each reached state that is easy to compare and store, and trying to find the state being currently explored in the set of states visited earlier. Many optimizations, such as Partial Order Reduction [Dor93] and thread symmetry [VHB+03], have been introduced to reduce the number of states that need to be explored. At the same time, since the state representation in case of software model checking is usually not of negligible size, its reduction can significantly speed up the state matching process as well. The related optimization techniques in this case focus on fast compression of state representation, such as those in [Huf52, rle], hashing, and identification of unimportant parts of states [SM07, LJ06, BFG99], e.g., the environment variables being the same over the entire program run, to be excluded.

While most of the optimizations are easy and fast to compute and apply if information about entire state space is available (e.g., for partial order reduction, it is clear which traces end up in the same state and what are the potential successors of the states along them), they become challenging if the state space is generated on-the-fly; this is the typical case of today's tools. Then, conservative simplifications have to be made to preserve correctness of the model checking results.

This work was partially supported by the Grant Agency of the Czech Republic Project 14-11384S.
Correspondence and offprint requests to: J. Kofroň, Email: jan.kofron@d3s.mff.cuni.cz
1.1. Problem statement

The existing techniques for state space reduction in the on-the-fly state-space-generation settings focus themselves just either to identify equivalent sets of event sequences resulting in the same program state or in identification of unused parts (w.r.t. future behaviour) [Dor93] or identification of dead variables, but restricting themselves just to local or non-heap variables [BFG99]. However, there is much space for reduction of the state space representation also when data stored in the heap are considered (both global variables and object fields). Involving their consideration in dead variable reduction (DVR) can further improve the state matching performance resulting thus in more scalable and applicable software model checking.

1.2. Goals

In this article, we describe two techniques aiming at identifying dead parts of the states, i.e., data that does not influence the future behaviour of the program, considering also data stored in the heap. We leave the analysis of dead local variables to other techniques, which can be combined with our approach. In particular, we introduce two dead variable analyses (DVA) of data stored in the heap differing in computational demands and precision, both suitable for being applied in an on-the-fly software model checker. Further, we also show the benefits of applying each of them on several benchmarks demonstrating thus the contribution to performance of on-the-fly explicit model checking. This article is an extension of our previous work [JK16]; here, we describe the analyses in more detail as well as provide detailed proofs and more experimental results supporting the significance of our contribution. In addition to that, we also discuss the aspects of DVA on the bytecode level.

2. Background

The purpose of state matching is to detect behaviourally equivalent states. For complex representation of program states, it can happen that future behaviours of two or more different states are equivalent. Informally, a state matching algorithm should match states if the future behaviour of the program starting in either of them is the same w.r.t. the property being verified. For reachability properties this means that from both of them, a state violating the property can either be reached or not. Since this view would be difficult to apply in the on-the-fly settings, we adopt a weaker one: two states are equivalent if the same set of equivalent states can be reached from either one.

We say that a variable is dead if it is not read until the program terminates or until its value is (re-)written before being read. Program states which differ in dead variables only are behaviourally equivalent and thus should be matched.

The contribution of DVR is two-fold. First, it reduces the state space, since states differing only in values of the identified dead variables are matched. This means that only a single representative of each set of matched states is explored. Second, state matching (i.e., canonicalization, hashing) can process only the live parts of state representation (ignoring dead parts), thus making the whole state matching process faster. This is also of a particular importance, since explicit-state model checkers spend a large amount of their runtime (approx. 30%) by state matching [NR09]. The former effect applies both for DVAs over local variables as well as for those focusing on the whole program state. On the other hand, the second effect can be observed only if a large enough portion of a program state is identified as dead, which can be expected only for the heap. Instead of ignoring dead variables, some DVR implementations set their value to a predefined constant (e.g., 0 or null). In such cases, the former effect is eliminated, since the size of the program state is not reduced at all. Note that the more precise DVA is, the more these effects manifest themselves.

Let us illustrate the aforementioned effects on the Java program in Fig. 1 and the red–black trees in Fig. 2 stored in the tree variable. The corresponding class is listed in Fig. 3. The tree is shared among threads and we assume that the program can generate either of them. No assertion is violated irrespective of whether either the left or the right tree has been generated. While the colours in red–black trees are used only in modifying operations (insertions and deletions), the contains operation does not access them and thus the variables (fields) representing the colours of nodes are dead. The same holds for the right descendant of the root node holding value 5.