Generalised multi-pattern-based verification of programs with linear linked structures

Milan Češka, Pavel Erlebach and Tomáš Vojnar
FIT, Brno University of Technology, Božetčehova 2, 61266, Brno, Czech Republic.
E-mail: ceska@fit.vutbr.cz; erlebach@fit.vutbr.cz; vojnar@fit.vutbr.cz

Abstract. The paper deals with the problem of automatic verification of programs working with extended linear linked dynamic data structures, in particular, pattern-based verification is considered. In this approach, one can abstract memory configurations by abstracting away the exact number of adjacent occurrences of certain memory patterns. With respect to the previous work on the subject the method presented in the paper has been extended to be able to handle multiple patterns, which allows for verification of programs working with more types of structures and/or with structures with irregular shapes. The experimental results obtained from a prototype implementation of the method show that the method is very competitive and offers a big potential for future extensions.

Keywords: Formal verification; Program analysis; Dynamic linked data structures

1. Introduction

Even the best programmer makes mistakes. Finding these mistakes can take a very long time and moreover, it does not have to be successful, especially in pointer handling programs working with possibly unbounded dynamic linked data structures. Verification of these programs is therefore a very active field of research. Many interesting results have been introduced in this area, but the situation is still unsatisfactory, mainly due to limited usability, inefficiency, the need of manual intervention and so on.

In this work, we contribute to the research on verification of programs with dynamic linked data structures by significantly improving a method first proposed in [YaB02, Yav04] which we denote as pattern-based verification. The method is based on identifying some repeated patterns in the so-called shape graphs that represent memory configurations and on abstracting the memory configurations by not remembering the exact number of occurrences of such patterns. In [YaB02, Yav04], the user of the method was supposed to provide the appropriate memory pattern to be used for abstraction. In [CEV05], we proposed a technique for a fully automatic detection of the memory patterns, which made the entire method fully automatic. However, only one pattern was considered. In this work, we extend pattern-based verification to be able to handle multiple patterns.

Let us just add that [YaB02, Yav04] attaches a counter to every summary node of a shape graph. The counter counts the number of instances of a memory pattern hidden behind the summary node. Methods of automated

Correspondence and offprint requests to: P. Erlebach, E-mail: erlebach@fit.vutbr.cz

1 Of course, as the problem we are dealing with is undecidable, the method we propose is semi-algorithmic, i.e., it may not terminate or it may answer “do not know” in some cases. Finding a class of structures and programs for which our method is guaranteed to stop with a definite answer remains a possible future work. However, it does not seem to be easy to identify a practically meaningful and decidable setting as the systems we consider become undecidable very quickly.
abstraction over integers may then be used to keep certain information about the value of these counters. We do not consider this extension here, but it can easily be introduced.

Our method can currently handle extended linear linked data structures by which we mean dynamic linked data structures with a linear skeleton (possibly with bidirectional links or closed into a cycle) on top of which two additional kinds of links can be added: (1) links to some globally shared dynamically allocated memory objects and (2) links to some local bounded linked sub-structures. This means that we cover a wide range of very practical structures such as lists, doubly-linked lists, circular (doubly-linked) lists that may be extended with often used constructions like additional pointers to the first or last element of the list or with elements having their “useful data” stored in separately allocated memory locations.

The basic intuition behind our method of automatic detection of memory patterns is simple. A memory pattern of an extended linear structure must have some kind of a linear skeleton (or backbone). This skeleton is surrounded by the so-called entry node and exit node. When we have several adjacent instances of the pattern, the exit node of the first instance becomes the entry node of another instance, etc. Therefore these nodes must have an equal selector environment (i.e., an equal set of both ingoing and outgoing selectors next pointer links) and equal selector paths to globally shared nodes. This fact may be exploited in detecting the pattern.

We currently handle the considered class of programs by generating for each control point of a program a set of abstract shape graphs covering all memory configurations that one may obtain at this point and detect whether there are no pointer manipulation errors (like references via undefined pointers or memory leaks). The generated abstract shape graphs can then be used for checking further safety properties of the programs like the fact that the result of reversing a linked list is again a linked list or even that it is exactly the reversal of the input list. If an error is found our technique provides us with a concrete error trace.

We have implemented the method in a prototype Prolog-based tool and tried it out on a number of programs of the described kind. The results are very encouraging both from the point of view of the simplicity of the method (and a large potential for further generalisations and improvements) as well as from the point of view of its very competitive efficiency.

Related work. Apart from [YaB02, Yav04] which inspired our work and which we significantly extend in this work, there has been published a number of other works on the topic of formal verification (or shape analysis or alias analysis) of programs with dynamic linked data structures.

The approach of Pale [MoS01] is based on using WS1S/WS2S (weak second-order monadic logic of one/two successors) and Mona [KIM01] to encode and handle pointer manipulations. The approach can handle some structures that we currently cannot handle (like trees and their extensions) though we are already working on an extension of our method covering such structures too. Unlike our approach, the approach of Pale is, however, not fully automatic for non-loop-free code—one has to manually specify loop invariants which may not be easy.

TVLA [SRW02] uses three valued logic to encode shape graphs. TVLA can also handle some more general structures than our approach in its current state. The TVLA approach is more automatic than the one of Pale, but still the user may be required to provide core predicates defining the class of structures used and instrumentation predicates (or simulation invariants in the later approach of [IRR04]) to make the abstraction sufficiently precise. Only very recently, there appeared [LRS05] a way to automatically generate some instrumentation predicates.

A fully automatic approach to verification of dynamic linked structures with one selector is presented in [BHMV05]. In this work, based on abstract regular model checking, memory configurations of programs with linear data structures are encoded using words over a suitable finite alphabet. Sets of configurations are then described as regular languages by finite automata, finite transducers are used to encode pointer-manipulating program statements, and by their repeated application the reachability set is computed. A special abstract-check-refine framework is used to make this method terminate as often as possible and to make it efficient. Compared to this approach, we can handle some more general structures like doubly-linked lists, lists with additional pointers, etc.

Another interesting, fully automatic approach is [LYY05] based on using a combination of shape graphs and a certain kind of attributed grammars. This approach can handle some structures (based on trees) that we currently cannot handle. On the other hand, we can also handle some structures that the approach of [LYY05] cannot handle (doubly-linked lists with additional pointers, etc.).

In [Ven99] following the earlier work of [Jon81, Deu94], the special problem of may-alias analysis is considered and an automata- and constraint-based representation of memory is used. May-alias analysis is also considered in [Ven04] where memory objects are represented by special time-stamps related to the state of the program when they were created. In [BIL03], an alias logic with a Hoare-like proof system based on representing memory configurations as a collection of automata is introduced.