Shape Analysis via Monotonic Abstraction

Parosh Aziz Abdulla[1], Ahmed Bouajjani[2], Jonathan Cederberg[1], Frédéric Haziza[1], Ran Ji[1] and Ahmed Rezine[1,2].

[1] Uppsala University, Sweden and [2] LIAFA, University of Paris 7, France.

Abstract. We propose a new formalism for reasoning about dynamic memory heaps, using monotonic abstraction and symbolic backward reachability analysis. We represent the heaps as graphs, and introduce an ordering on these graphs. This enables us to represent the violation of a given safety property as the reachability of a finitely representable set of bad graphs. We also describe how to symbolically compute the reachable states in the transition system induced by a program.

1 Introduction

Software verification needs the use of efficient algorithmic techniques for the analysis of infinite-state models. The sources of infiniteness are multiple and can be related to complex control such as (potentially recursive) procedure calls and dynamic creation of processes, or to the manipulation of (unbounded-size) dynamic data-structures and variables ranging over infinite data domains. A lot of work has been devoted in the last decade to the design of automatic verification techniques for infinite-state models, and several general approaches and formal frameworks have emerged allowing either to establish decidability results and derive verification algorithms (e.g., [2, 20]), or to define generic exact/abstract analysis procedures (e.g., [29, 22, 11, 7]).

One of the widely adopted frameworks in this context of infinite-state verification is based on the concept of *monotonic systems w.r.t. a well-quasi ordering* [2, 20]. This framework provides a scheme for proving the termination of the (backward) reachability analysis, and it has been used for the design of verification algorithms for various models including Petri nets, lossy channel systems, timed Petri nets, broadcast protocols, etc. (see, e.g., [5, 18, 19, 6]). The idea is, given a class of models, to define a preorder $\preceq$ on the configuration space such that (1) $\preceq$ is a simulation relation on the considered models, and (2) $\preceq$ is a well-quasi ordering (WQO for short). If such a preorder can be defined, then it can be proved that the reachability problem of an upward-closed set of configurations (w.r.t. $\preceq$) is decidable. Indeed, (1) monotonicity implies that for any upward-closed set, the set of its predecessors is an upward-closed set, and (2) the fact that $\preceq$ is a WQO implies that every upward-closed set can be characterized by its *finite* set of minimal elements. Therefore, starting from an upward-closed set of configurations U, the iterative computation of the backward reachable configurations from U necessarily terminates since only a finite number of steps are needed to capture all minimal elements of the set of predecessors of U. Obviously, this requires that upward-closed sets can be effectively represented and manipulated (i.e., there are procedures for, e.g., computing immediate predecessors and unions, and for checking entailment). This general scheme can be applied for the verification of safety properties since this problem can be reduced to checking the reachability of a set of bad configurations which is typically an upward-closed set w.r.t. the considered preorder. (For instance, mutual exclusion is violated as soon as there are (at least) two processes in the critical section.)

Unfortunately, many systems do not fit into this framework, in the sense that there is no nontrivial (useful) WQO for which these systems are monotonic. Nevertheless, a natural approach to overcome this problem is, given a preorder $\preceq$, to define an abstract semantics of the considered systems which forces their monotonicity. Basically, the idea is to consider that a transition is possible from a configuration c_1 to c_2 if it is possible from any smaller configuration $c_1' \preceq c_1$ to c_2. This simple idea has been used recently in works concerning the verification of parametrized networks of (finite/infinite-state) processes, and surprisingly, it leads to quite efficient abstract analysis techniques which allow to handle *fully automatically* several non-trivial examples of such systems [3, 4]. This encourages us to investigate its application to other classes of complex systems.

In [1], we developed a framework based on the approach introduced above for the verification of sequential iterative programs manipulating dynamic memory heaps. The issue of verifying automatically such programs has received a lot of attention in the last few years, and many approaches and techniques have been developed including static-analysis and abstraction-based frameworks (see, e.g., [28]), logic-based frameworks(see, e.g., [27, 25]), automata-based frameworks (see, e.g., [21, 14]), etc. In [1], we introduced a framework based on symbolic (backward) reachability analysis using upward-closed sets of heap graphs (w.r.t. some appropriate preorder). As a first step toward this framework, we presented there the results of our investigations concerning the case of programs manipulating heap structures with *one* next-selector, i.e., heaps of programs manipulating lists with possibility of sharing and circularity.

More precisely, we considered that heaps are represented as labeled graphs, where labels correspond to positions of program variables. We proposed a preorder $\preceq$ between heap graphs which corresponds basically to the following: Given two graphs g_1 and g_2, we have $g_1 \preceq g_2$ if g_1 can be obtained from g_2 by a sequence of transformations consisting of either deleting an edge, a variable, or an isolated vertex, or of contracting segments (i.e., sequence of vertices) without sharing in the graph.

Actually, our graph representations in [1] correspond in general to sets of heaps instead of a single one. They can be seen as minimal patterns (w.r.t. $\preceq$), and they represent all the heaps that subsume (w.r.t. $\preceq$) these patterns. Therefore, our graph representations define upward-closed sets of heap graphs.

We also provided procedures for computing sets of predecessors w.r.t. the abstract semantics we consider (introduced above), and for checking entailment. These procedures allow to define a *simple algorithm* which computes an over-approximation of the set of backward reachable configurations starting from an upward-closed set of heap graphs (effectively given as a finite set of minimal elements). We showed that this algorithm *always terminates* by proving that the preorder we defined on heap graphs is a WQO.

Our analysis allows to check properties such as absence of null dereferencing as well as absence of garbage creation. Moreover, it allows to check shape (well-formedness) properties of the heaps (for instance the fact that the output is always a list without sharing). We showed indeed that these kinds of verification problems can be reduced to the problem of reaching sets of bad configurations corresponding to the existence in the heap graph of some *minimal bad patterns*. We also provided experimental results showing the effectiveness of our approach. In this report, we propose an alternative order on the graphs which arise, and show how to compute the predecessor using this new ordering. This new ordering allows for checking other properties than the ordering proposed in [1]. For example, the detection of dangling pointers is possible.

Related work. As mentioned before, several approaches to the automatic analysis of programs with dynamic linked data structures have been proposed (see, e.g., [28, 17, 21, 14]). Shape analysis as introduced in [28] is based on the computation of abstract shape graphs using the so-called instrumentation predicates. An automata-based approach using abstract regular model checking (ARMC) [15] has been proposed in [13, 14]. In [17, 10], an automatized analysis approach based on separation logic combined with abstraction techniques (close to widening techniques) has been proposed. With respect to these approaches, the one we present in this paper is conceptually and technically different and simpler. In particular, the ARMC-based approach needs the manipulation of quite complex encodings of the heap graphs into words or trees (in order to represent sets of heap encodings using finite-state automata), and use a sophisticated machinery for manipulating these encodings based on representing program statements as (word/tree) transducers. In contrast, the approach presented here uses a natural representation of heaps as graphs and employs direct procedures for computing operations on these graphs. This direct approach has already shown its advantages w.r.t. the approach using transducers in the context of regular model checking for parametrized networks of processes [3]. Also, the approach we present uses a built-in abstraction principle which is different from the ones used in the existing approaches, and which makes the analysis fully automatic.

The existing approaches mentioned above (shape analysis, abstract regular model checking, separation logic) can handle some classes of general heap structures (including doubly linked lists, lists of lists, trees, etc.). Although the techniques presented in this paper concern the case of heap structures with 1-next selector, our approach (based on upward-closed abstractions w.r.t. preorders on graphs) can in principle be extended to more general classes of heaps.

Concerning the particular class of programs manipulating heaps with 1-next selector, there are many other verification approaches which have been developed recently (see, e.g., [24, 13, 23, 12, 16, 8]). Almost all these works use the fact that in this case (1) the heap graphs are collections of reversed trees potentially having their roots connected to a loop, and moreover (2) the number of leaves and shared vertices in these graphs is bounded linearly in terms of the number of program variables. For instance, in [24], these properties are used to define an abstraction which consists of contracting all segments without sharing. In our case, we use these properties in order to prove that the preorder we propose on graph representations is a WQO. However, our abstraction is different from the one proposed for instance in [24] since we can have graphs which are not minimal w.r.t. to contraction (e.g., we can express the fact that the length of a segment is at least some given natural number), and we can also have graphs corresponding to a partial description of the heap where only a *part* of the reachable heap from *some* of the program variables is constrained.

In [12, 9], translations from programs with lists to counter automata have been defined based on the representation of heap graph as its contracted version supplied with the information about the length of each contracted sharing-free segments. These translations allow to use various existing techniques for the analysis of counter systems in order to check safety properties involving constraints relating the lengths of different lists, or to check termination. Such analysis involving quantitative reasoning cannot be done with the techniques presented in this paper. As said above, these techniques can handle some reasoning about the sizes of the lists, but only concerning constraints on minimal lengths. However, extensions of our techniques to more general constraints (e.g., gap-order constraints [26]) are possible.

Outline. In the next section, we introduce the class of programs we consider together with their graph representations. In Section 3, we describe a set of graph operations which we use in the subsequent sections. Section 4 introduces the new ordering on configurations. In Section 5, we introduce a relation which we use as the basic step in the reachability algorithm. Finally, in Section 6, we discuss conclusions and further work.

2 Preliminaries

We consider programs that operate on data structures with one next-pointer such as traditional singly-linked lists and circular lists (possibly sharing their parts). We represent the store as a graph, where the vertices represent the list cells, and the successor of a vertex represents the cell pointed to by the current one. The graphs are of a special form in the sense that each vertex has at most one successor. A program also uses a finite set of pointers which we call *variables.* A cell is labeled by the (possibly empty) set of variables pointing to it.

For a partial function f, we write $f(a) \neq \bot$ to denote that $f(a)$ is in defined, and $f(a) = \bot$ to denote that $f(a)$ is undefined. For a (partial) function f, we use $f[a \leftarrow b]$ to denote the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ if $x \neq a$. We will abuse this notation and take $f[a \leftarrow \bot]$ to mean the the function f' such that $f'(x) = f(x)$ if $x \neq a$ and $f'(a) = \bot$. For a (possibly partial) function $f : A \to B$ and a subset $A' \subseteq A$, we denote by $f|_{A'}$ the restriction of f to the set A'. Formally this means that $f|_{A'}$ is the function $f' : A' \to B$ such that for all $a \in A'$, $f(a) \neq \bot \implies f'(a) = f(a)$ and $f(a) = \bot \implies f'(a) = \bot$

Formally, we assume a finite set X of variables. $\Box$ and $\#$ are two constants, which represent uninitialized and null respectively. A *graph* g is a triple $(V, succ, \lambda)$ where $V \supseteq \{\Box, \#\}$ is a finite set of *vertices*, *succ* is a function $succ : (V \setminus \{\Box, \#\}) \rightarrow V$ and λ is a partial function $\lambda : X \rightharpoonup V$. For simplicity, we use $W(V)$ to denote $V \setminus \{\Box, \#\}$. Intuitively, $W(V)$ corresponds to the list cells in the heap. The function *succ* defines the successors of the list cells in the heap. For a vertex $v \in W(V)$, $succ(v) = \#$ represents that v has the null pointer as its next-pointer, which in general is a well-defined ending of a list segment. If $succ(v) = \Box$, the list cell represented by v has a dangling next-pointer. The partial function λ defines the vertex to which a given variable points. If $\lambda(x) \neq \bot$, the $\lambda(x)$ is to be interpreted in the same way as *succ*, i.e. x is a null pointer, dangling pointer or pointing to a cell.

A *program* P is a pair (Q, T) where Q is a finite set of *control states* and T is a finite set of *transitions.* A transition is a triple (q_1, a, q_2) where $q_1, q_2 \in Q$ are control states and a is an *action.* An *action* is of one of the following forms $x = y$, $x \neq y$, $x := y$, $x.next = y$, or $x := y.next$ The transition corresponds to the program changing control state from q_1 to q_2 while performing the operation described in a on the data structure. We choose to work with the above minimal set of operations. Other operations, e.g., $x = y.next$, $x \neq y.next$, etc, can be expressed using the given set.

A *configuration* c is a pair (q, g) where $q \in Q$ is a control state and g is a graph. We define a transition relation on configurations as follows. Let $t = (q_1, a, q_2)$ be a transition and let $c = (q, g)$ and $c' = (q', g')$ be configurations. We write $c \xrightarrow{t} c'$ to denote that $q = q_1$, $q' = q_2$, and $g \xrightarrow{a} g'$, where $g \xrightarrow{a} g'$ holds if one of the following conditions is satisfied:

- a is of the form $x = y$, $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(y) \neq \Box$, $\lambda(x) = \lambda(y)$, and $g' = g$.
- a is of the form $x \neq y$, $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(y) \neq \Box$, $\lambda(x) \neq \lambda(y)$, and $g' = g$.
- a is of the form $x := y$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \Box$, $V' = V$, $succ' = succ$, and $\lambda' = \lambda[x \leftarrow \lambda(y)]$.
- a is of the form $x := y.next$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \#$, $\lambda(y) \neq \Box$, $V' = V$, $succ' = succ$, and $\lambda' = \lambda[x \leftarrow succ(\lambda(y))]$.
- a is of the form $x.next := y$, $\lambda(x) \neq \bot$, $\lambda(y) \neq \bot$, $\lambda(x) \neq \#$, $\lambda(x) \neq \Box$, $\lambda(y) \neq \Box$, $V' = V$, $\lambda' = \lambda$, and $succ' = succ[\lambda(x) \leftarrow \lambda(y)]$.

We define $\longrightarrow$ as $\bigcup_{t \in T} \xrightarrow{t}$ and use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. For sets C_1 and C_2 of configurations, we use $C_1 \longrightarrow C_2$ to denote that $c_1 \longrightarrow c_2$ for some $c_1 \in C_1$ and $c_2 \in C_2$. By $c_1 \longrightarrow C_2$ we mean $\{c_1\} \longrightarrow C_2$. We define $c_1 \xrightarrow{*} C_2$, $C_1 \xrightarrow{*} C_2$, etc in a similar manner to above.

3 Operations on Graphs

In this section, we define a number of operations on graphs which we use in the subsequent sections. For the rest of the section, we assume a graph $g = (V, succ, \lambda)$.

For $v \in V$ and $w \in W(V)$, we use $(g.\mathit{succ})[w \leftarrow v]$ to denote the graph $g' = (V', \mathit{succ}', \lambda')$ where $V' = V$, $\lambda' = \lambda$, and $\mathit{succ}' = \mathit{succ}[w \leftarrow v]$. Intuitively, we only modify g so that v becomes the successor of w. We define $(g.\lambda)[x \leftarrow v]$ analogously. That is, we make x point to v.

For a vertex $v \in W(V)$, we say that v is *simple* if $|\mathit{succ}^{-1}(v)| = 1$ and $\lambda^{-1}(v) = \emptyset$. In other words, v has exactly one predecessor and no label. We say that v is a *leaf* in g if $\mathit{succ}^{-1}(v) = \emptyset$, and $\lambda^{-1}(v) = \emptyset$. In other words, v has no predecessors and it is not labeled by any variable. We say that v is *isolated* in g if v is a *leaf*, and $\mathit{succ}(v) = \Box$. In other words, v is a leaf with successor uninitialized.

Operations on vertices. For a vertex $v \notin V$, we use $g \oplus v$ to denote the graph $g' = (V', \mathit{succ}', \lambda')$ such that $V' = V \cup \{v\}$, $\mathit{succ}' = \mathit{succ}[v \leftarrow \Box]$, and $\lambda' = \lambda$, *i.e.* we add a new cell to g. Observe that the added vertex is then isolated.

For a vertex $v \in W(V)$, we use $g \ominus v$ to denote the graph $g' = (V', \mathit{succ}', \lambda')$ such that $V' = V \setminus \{v\}$, $\mathit{succ}' = \mathit{succ}|_{W(V)\setminus\{v\}}$, and $\lambda' = \lambda$. Note that this operation is not well defined if there is a vertex v' such that $v \neq v'$ and $\mathit{succ}(v') = v$.

Operations on variables. We define $g \oplus x$ to be the set of graphs we get from g by letting x point anywhere inside g, except on the constant representing uninitialized. Formally, we define $g \oplus x$ to be the smallest set containing each graph g' such that one of the following conditions is satisfied:

1. There is a $v \in V$ such that $v \neq \Box$ and $g' = (g.\lambda)[x \leftarrow v]$, *i.e.* we make x point to some vertex in g.
2. There is a $v \notin V$, a $v' \in V$ and graphs g_1, g_2 such that $g_1 = (g \oplus v)$ $g_2 = (g_1.\lambda)[x \leftarrow v]$, $g' = (g_2.\mathit{succ})[v \leftarrow v']$, *i.e.* we add a leaf to g and make x point to it.
3. There is a $v \notin V$ and graphs g_1, g_2 such that $g_1 = (g \oplus v)$ $g_2 = (g_1.\lambda)[x \leftarrow v]$, $g' = (g_2.\mathit{succ})[v \leftarrow v]$, *i.e.* we add an isolated loop to g and make x point to it.
4. There are $v_1 \in W(V)$, $v_2 \notin V$, and graphs g_1, g_2, g_3, such that $g_1 = g \oplus v_2$, $g_2 = (g_1.\mathit{succ})[v_2 \leftarrow \mathit{succ}(v_1)]$, $g_3 = (g_2.\mathit{succ})[v_1 \leftarrow v_2]$, and $g' = (g_3.\lambda)[x \leftarrow v_2]$. *i.e.* we add a new vertex v_2 in between v_1 and its successor and make x point to v_2.
5. There are $v_1 \in W(V)$, $v_2, v_3 \notin V$, and graphs g_1, g_2, g_3, g_4 and g_5, such that $g_1 = g \oplus v_2$, $g_2 = (g_1.\mathit{succ})[v_2 \leftarrow \mathit{succ}(v_1)]$, $g_3 = (g_2.\mathit{succ})[v_1 \leftarrow v_2]$, $g_4 = g \oplus v_3$, $g_5 = (g_4.\mathit{succ})[v_2 \leftarrow v_3]$, and $g' = (g_4.\lambda)[x \leftarrow v_3]$. *i.e.* we add a new vertex v_2 in between v_1 and its successor, add a new leaf v_3 to it, and make x point to v_3.

For variables x and y with $\lambda(x) \neq \perp$ and $\lambda(x) \neq \Box$, we define $g \oplus_{=x} y$ to be the graph $g' = (g.\lambda)[y \leftarrow \lambda(x)]$, *i.e.* we make y point to the same vertex as x. Furthermore, we define $g \oplus_{\neq x} y$ to be the smallest set containing each graph g' such that $g' \in (g \oplus y)$ and $\lambda'(y) \neq \lambda'(x)$, *i.e.* we make y point anywhere inside g except to the vertex pointed to by x.

For variables x and y with $\lambda(x) \neq \perp$, $\lambda(x) \neq \Box$ and $\lambda(x) \neq \#$, we define $g \oplus_{x\rightarrow} y$ to be the set of graphs we get from g by letting y point to the successor of the cell pointed to by x. Formally, we define $g \oplus_{x\rightarrow} y$ to be the smallest set containing each graph g' such that one of the following conditions is satisfied:

1. $g' = (g.\lambda)\left[y \leftarrow \mathit{succ}(\lambda(x))\right]$, i.e.we make y point to the successor of the vertex pointed to by x
2. there is a vertex $v \notin V$, and graphs $g_i = (V_i, \mathit{succ}_i, \lambda_i)$ for $i = 1, 2, 3$, such that $g_1 = g \oplus v$, $g_2 = (g_1.\mathit{succ})\left[v \leftarrow \mathit{succ}_1(\lambda(x)\right]$, $g_3 = (g_2.\mathit{succ})\left[\lambda(x) \leftarrow v\right]$, and $g' = (g_3.\lambda)\left[y \leftarrow v\right]$, i.e.we add a new vertex v in between the vertex pointed to by x and its successor and make y point to v.

For variables x and y with $\lambda(x) \neq \perp$, we define $g \oplus_{x\leftarrow} y$ to be the set of graphs we get from g by letting y point to any cell whose successor is pointed to by x. Formally, we define $g \oplus_{x\leftarrow} y$ to be the smallest set containing each graph g' such that one of the following conditions is satisfied:

1. there is a vertex $v \in W(V)$ such that $\mathit{succ}(v) = \lambda(x)$, and $g' = (g.\lambda)\left[y \leftarrow v\right]$, i.e., we make y point to a direct predecessor of $\lambda(x)$
2. there is a vertex $v \notin V$ and graphs g_1, g_2, such that $g_1 = g \oplus v$, $g_2 = (g_1.\mathit{succ})\left[v \leftarrow \lambda(x)\right]$, $g' = (g_2.\lambda_2)\left[y \leftarrow v\right]$. i.e.we add a new leaf v to g, make its successor the vertex pointed to by x, and make y point to it.
3. there are vertices $v_1 \in W(V)$ and $v_2 \notin V$, and graphs g_1, g_2, g_3, such that $\mathit{succ}(v_1) = \lambda(y)$ $g_1 = g \oplus v_2$, $g_2 = (g_1.\mathit{succ})\left[v_2 \leftarrow \lambda(x)\right]$, $g_3 = (g_2.\mathit{succ})\left[v_1 \leftarrow v_2\right]$, and $g' = (g_3.\lambda)\left[y \leftarrow v_2\right]$. i.e.we add a new vertex v_2 in between the vertex pointed by x and its predecessors and make y point to v_2.

For a variable x, we define $g \ominus x$ to be the graph g' where $g' = (g.\lambda)\left[x \leftarrow \perp\right]$.
Operations on edges. For a graph g with $\lambda(x) \neq \perp$, we define $g \boxminus (x \rightarrow)$ to be the smallest set containing each graph g' such that one of the following conditions is satisfied:

1. there is a $v \in V$ such that $g' = (g.\mathit{succ})\left[\lambda(x) \leftarrow v\right]$, i.e.we make some vertex in g the successor of g
2. there are vertices $v_1 \in W(V)$ and $v_2 \notin V$ and graphs g_1, g_2, g_3 such that $g_1 = g \oplus v_2$, $g_2 = (g_1.\mathit{succ})\left[v_2 \leftarrow \mathit{succ}(v_1)\right]$, $g_3 = (g_2.\mathit{succ})\left[v_1 \leftarrow v_2\right]$, $g' = (g_3.\mathit{succ})\left[\lambda(x) \leftarrow v_2\right]$, i.e.we put an new vertex v_2 in between v_1 and its successor, and make the new vertex the successor of $\lambda(x)$
3. there is a vertex $v \notin V$ and graphs g_1, g_2 such that $g_1 = g \oplus v$, $g_2 = (g_1.\mathit{succ})\left[v \leftarrow v\right]$, $g' = (g_2.\mathit{succ})\left[\lambda(x) \leftarrow v\right]$, i.e.we add a new single vertex loop to g, and make the new vertex the successor of $\lambda(x)$.

4 Ordering

In this section, we introduce an ordering on configurations. Based on the ordering, we will define the coverability problem which we use to check safety properties, and define the abstract transition relation. The latter is an over-approximation of the concrete transition relation.
Ordering. Let $g = (V, \mathit{succ}, \lambda)$ and $g' = (V', \mathit{succ}', \lambda')$. We write $g \triangleleft g'$ to denote that one of the following properties is satisfied:

(i) *Variable Deletion*: $g = g' \ominus x$ for some variable x,
(ii) *Leaf Deletion*: there is a cell $w \in W(V')$ such that w is a leaf, and $g = g' \ominus w$,
(iii) *Isolated Cycle Deletion*: there is a cell $w \in W(V')$ such that w is simple in g', $succ(w) = w$, and $g = g' \ominus w$, or
(iv) *Contraction*: there are cells $w_1, w_2 \in W(V')$, a vertex $v \in V'$ and a graph g_1 such that w_2 is simple, $w_1 \neq w_2$, $w_2 \neq v$, $succ'(w_1) = w_2$, $succ'(w_2) = v$, $g_1 = (g_1.succ)[w_1 \leftarrow v_3]$ and $g = g_1 \ominus w_2$.

We write $g \preceq g'$ to denote that there are $g_0 \lhd g_1 \lhd g_2 \lhd \cdots \lhd g_n$ with $n \geq 0$, $g_0 = g$, and $g_n = g'$. That is, we can obtain g from g' by performing a finite sequence of variable deletion, leaf deletion, isolated cycle deletion and contraction operations. For configurations $c = (q, g)$ and $c' = (q', g')$, we write $c \preceq c'$ to denote that $q' = q$ and $g \preceq g'$.

For a configuration c, we use $c\!\uparrow$ to denote the *upward closure* of c, *i.e.* $c\!\uparrow = \{c' \mid c \preceq c'\}$. We use $c\!\downarrow$ to denote the *downward closure* of c, *i.e.* $c\!\downarrow = \{c' \mid c' \preceq c\}$. For a set C of configurations, we define $C\!\uparrow$ as $\bigcup_{c \in C} c\!\uparrow$. We define $C\!\downarrow$ analogously.

Safety Properties. In order to analyze safety properties, we study the *coverability problem* defined below.

Coverability
Instance

– Sets C_{Init} and C_F of configurations.

Question Is it the case $C_{Init} \xrightarrow{*} C_F\!\uparrow$?

Intuitively, $C_F\!\uparrow$ represents a set of "bad" states which we do not want to reach during the execution of the program. This set is represented by a set C_F of minimal elements. Therefore, checking safety with respect to these properties amounts to solving the coverability problem.

Abstract Transition Relation. We write $c_1 \xrightarrow{t}_A c_2$ to denote that there is a c_3 such that $c_3 \preceq c_1$ and $c_3 \xrightarrow{t} c_2$. In other words, a step of the abstract transition relation consists of first moving to a smaller configuration (wrt $\preceq$) and then performing a step of the concrete transition relation. Notice that the abstraction corresponds to an over-approximation and therefore any safety property which holds in the abstract system will also hold in the concrete one.

5 Computing Predecessors

The main idea behind our algorithm to solve the coverability problem, is to perform backward reachability analysis. The basic step of the algorithm uses a relation $\rightsquigarrow$ defined on the set of configurations. Intuitively, $c \rightsquigarrow c'$ means that, from c', we can perform a transition and reach a configuration in the upward closure of c. First, we give the formal definition of $\rightsquigarrow$, and then describe some of its properties, and in particular how it relates to the transition relation $\longrightarrow$.

For a graph $g = (V, succ, \lambda)$, a graph g', and an action a, we write $g \overset{a}{\rightsquigarrow} g'$ to denote that one of the following conditions is satisfied:

1. a is of the form $x = y$ and one of the following conditions is satisfied:
   (a) $\lambda(x) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(y) \neq \bot$, $\lambda(x) = \lambda(y)$ and $g' = g$.
   (b) $\lambda(x) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(y) = \bot$, and $g' = g \oplus_{=x} y$.
   (c) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$, and $\lambda(y) \neq \Box$, $g' = g \oplus_{=y} x$.
   (d) $\lambda(x) = \bot$, $\lambda(y) = \bot$, and $g' = g_1 \oplus_{=x} y$ for some $g_1 \in g \oplus x$.
   In order to be able to perform the action, the variables x and y should point to the same vertex. If one (or both) of them are missing, then we add them to the graph (with the restriction that they point to the same vertex).
2. a is of the form $x \neq y$ and one of the following conditions is satisfied:
   (a) $\lambda(x) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \Box$, $\lambda(x) \neq \lambda(y)$ and $g' = g$.
   (b) $\lambda(x) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(y) = \bot$, and $g' \in g \oplus_{\neq x} y$.
   (c) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \Box$, and $g' \in g \oplus_{\neq y} x$.
   (d) $\lambda(x) = \bot$, $\lambda(y) = \bot$, and $g' \in g_1 \oplus_{\neq x} y$ for some $g_1 \in g \oplus x$.
   We proceed as in case 1, but now under the restriction that x and y point to different vertices (rather than to the same vertex).
3. a is of the form $x := y$ and one of the following conditions is satisfied:
   (a) $\lambda(x) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(y) \neq \bot$, $\lambda(x) = \lambda(y)$ and $g' = g \ominus x$.
   (b) $\lambda(x) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(y) = \bot$, and $g' = g_1 \ominus x$ where $g_1 = g \oplus_{=x} y$.
   (c) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \Box$, and $g' = g$.
   (d) $\lambda(x) = \bot$, $\lambda(y) = \bot$, and $g' \in g \oplus y$.
   The difference compared to case 1 is that the variable x may have had any value before performing the assignment. Therefore, we remove x from the graph.
4. a is of the form $x := y.next$ and one of the following conditions is satisfied:
   (a) $\lambda(x) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \Box$, $\lambda(y) \neq \#$, $succ(\lambda(y)) = \lambda(x)$ and $g' = g \ominus x$.
   (b) $\lambda(x) \neq \bot$, $\lambda(y) = \bot$, and $g' = g_1 \ominus x$ where $g_1 \in g \oplus_{x\leftarrow} y$.
   (c) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \Box$, $\lambda(y) \neq \#$, and $g' = g$.
   (d) $\lambda(x) = \bot$, $\lambda(y) = \bot$, and there are graphs g_1, g_2 such that $g_1 \in g \oplus x$, $g_2 \in g_1 \oplus_{y\leftarrow} x$ and $g' = g_2 \ominus x$.
   Similarly to case 3 we remove x from the graph. The successor of y should point to the same vertex as x.
5. a is of the form $x.next := y$ and one of the following conditions is satisfied:
   (a) $\lambda(x) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(x) \neq \#$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \Box$, $succ(\lambda(x)) = \lambda(y)$ and $g' \in g \boxminus (x \rightarrow)$.
   (b) $\lambda(x) \neq \bot$, $\lambda(x) \neq \Box$, $\lambda(x) \neq \#$, $succ(\lambda(x)) \neq \Box$ and $\lambda(y) = \bot$ and $g' \in g_1 \boxminus (x \rightarrow)$, where $g_1 \in g \oplus_{x\rightarrow} y$.
   (c) $\lambda(x) = \bot$, $\lambda(y) \neq \bot$, $\lambda(y) \neq \Box$, and $g' \in g_1 \boxminus (x \rightarrow)$, where $g_1 \in g \oplus_{y\leftarrow} x$.
   (d) $\lambda(x) = \bot$, $\lambda(y) = \bot$ and there are graphs g_1, g_2 such that $g_1 \in g \oplus y$, $g_2 \in g_1 \oplus_{y\leftarrow} x$ and $g' \in g_2 \boxminus (x \rightarrow)$.
   After performing the action, the successor of the vertex labeled by x should be the same vertex as the one labeled by y. Before performing the action, the successor could have been anywhere inside the graph, and the corresponding edge is therefore removed.

Remark In the above definition, we assume that x and y are different variables. It is straightforward to handle the case where they are the same variable.

6 Conclusions

In [1], we give a proof that the ordering on graphs we defined therein is a WQO. We also supplied a proof for the monotonicity of the predecessor relation. The same proof ideas can be used for proving that the ordering presented in this report is a WQO and to prove monotonicity of the predecessor relation.

The main feat of the new approach is the possibility to express new properties such as dangling pointers, and it seems very promising. It also shows the generality of the approach, in the sense that using different orderings one can reason about different properties. This is the key to treating more general structures.

References

1. P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic abstractions for programs with dynamic memory heaps. In *CAV'08*. LNCS, 2008.
2. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160(1-2):109–127, 2000.
3. P. A. Abdulla, G. Delzanno, N. B. Henda, and A. Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *TACAS'07*, pages 721–736. LNCS 4424, 2007.
4. P. A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV'07*, pages 145–157. LNCS 4590, 2007.
5. P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Inf. Comput.*, 127(2):91–101, 1996.
6. P. A. Abdulla and B. Jonsson. Model checking of systems with many identical timed processes. *Theor. Comput. Sci.*, 290(1):241–264, 2003.
7. P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A Survey of Regular Model Checking. In *CONCUR'04*, pages 35–48. LNCS 3170, 2004.
8. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis of single-parent heaps. In *VMCAI'07*, pages 91–105. LNCS 4349, 2007.
9. S. Bardin, A. Finkel, É. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. In *Proceedings of the 5th Intern. Workshop on Automated Verification of Infinite-State Systems (AVIS'06)*, 2006.
10. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV'07*, pages 178–192. LNCS 4590, 2007.
11. A. Bouajjani. Languages, rewriting systems, and verification of infinite-state systems. In *ICALP'01*, pages 24–39. LNCS 2076, 2001.
12. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
13. A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.
14. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.

15. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.
16. D. Distefano, J. Berdine, B. Cook, and P. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
17. D. Distefano, P. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.
18. E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *LICS*, pages 70–80, 1998.
19. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proceedings of LICS '99*, pages 352–359. IEEE Computer Society, 1999.
20. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *TCS*, 256(1-2):63–92, 2001.
21. J. Jensen, M. Jørgensen, N. Klarlund, and M. Schwartzbach. Automatic Verification of Pointer Programs Using Monadic Second-order Logic. In *Proc. of PLDI'97*, 1997.
22. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theor. Comput. Sci.*, 256(1-2):93–112, 2001.
23. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006.
24. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.
25. P. W. O'Hearn. Separation logic and program analysis. In *SAS'06*, page 181. LNCS 4134, 2006.
26. P. Z. Revesz. A closed-form evaluation for datalog queries with integer (gap)-order constraints. *Theor. Comput. Sci.*, 116(1&2):117–149, 1993.
27. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.
28. S. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
29. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV'98*, pages 88–97. LNCS 1427, 1998.