From pointer systems to counter systems using shape analysis

Sébastien Bardin [2] Alain Finkel [2] Étienne Lozes [2]

LSV, ENS Cachan & CNRS UMR 8643
61 av. Pdt Wilson, 94235 Cachan Cedex France

Arnaud Sangnier [2]

EDF R&D, LSV, ENS Cachan & CNRS UMR 8643
61 av. Pdt Wilson, 94235 Cachan Cedex France

Abstract

We aim at checking safety properties on systems manipulating dynamic linked lists. First we prove that every pointer system is bisimilar to an effectively constructible counter system. We then deduce a two-step analysis procedure. We first build an over-approximation of the reachability set of the pointer system. If this over-approximation is too coarse to conclude, we then extract from it a bisimilar counter system which is analyzed via efficient symbolic techniques developed for general counter systems.

Key words: dynamic allocation, automatic verification, counter system, pointer system, shape analysis

1 Introduction

Context. The model checking techniques for infinite-state systems are now an active research area. These techniques allow to verify different kinds of models like pushdown systems, channel systems, counter systems, pointer systems and many other models like rewriting systems. For some of these models, there exist today tools for verifying such systems: Moped (for pushdown systems), TREX [1] (for channel systems), FAST[6], LASH[16], TREX[1], BRAIN[17] and ALV (for counter systems), TVLA[13] and PALE[14] (for pointer systems).

[1] This work has been partially supported by contract 4300038040 between EDF/LSV and by the AVERILES project.

[2] `[bardin|finkel|lozes|sangnier]@lsv.ens-cachan.fr`

The model of counter systems is very expressive, it is well-known and it has been used (alone or with some extensions) for modelling a lot of case studies (see for instance the recent verification of the embedded protocol TTP by using the tool FAST[6]).

The problem. Explicit memory management is a common source of errors in programs. Mechanisms of dynamic allocation are naturally available in imperative languages such as C, and functional languages often include such facilitites for efficiency purpose. Explicit memory management is typically exploited in industrial, real-time or embedded systems. This efficiency has naturally a price: in such a programming style, the safety of the operations of allocation, deallocation, and dereferencing relies on a sound conception of the program. As so, many unsafe behaviors can happen, such as *memory violation*, *memory leak*, unwilled aliasing, etc.

There is currently a great interest for defining automatic methods for the verification of these systems with pointers. However, there exists different models of programs with memory which are not compared and, in general, they are not supported by model checkers: as a matter of fact, TVLA is an analyser using abstract interpretation and PALE is a prover which needs the help of the user.

We would like to be able to automatically verify the qualitative and quantitative properties. *Shape analysis*, first introduced by Sagiv and Lev-Ami [12,13,15], opened a way to a qualitative analysis of the memory manipulated by the program. Typical questions answered by this method are list-shape preservation, aliasing detection, etc.

The quantitative analysis [5,11] based on Presburger arithmetics, recently extended by more powerful decidable arithmetics [9] tries to answer about the length equality of two lists, the preservation of the memory size, etc.

In this paper we address the problem of verification of both types of properties for a class of (non-interprocedural) programs manipulating list structures, possibly with circularity and sharing, called here *pointer systems* [5]. We abstract from standard data structures: pointers are the only data type we consider, and we manipulate abstract heap addresses without pointer arithmetics.

Our contribution is to propose an automatic translation of any pointer system S_P into a bisimilar counter system S_C. This translation is based on a notion of memory shape. This translation is more intuitiv than the other translations in logic formula. Another advantage of our translation is that the input formalism can be easily extended to pointer systems with counters. We define a general framework for both quantitative and qualitative analysis. We develop a two-step analysis procedure of pointer systems based on the counter system generated. (1) A static analysis of the counter system: this analysis gives an over-approximation of the memory shapes the system may reach, allowing to derive safety properties efficiently in some cases. (2) If the first analysis is not tight enough, then a reachability analysis is performed at the counter system level to refine the previous approximation. This analysis relies on efficient acceleration techniques developed for counters [3,4], and can discover automatically complex arithmetic relations in the memory heap.

Related Works. Some tools and techniques checking reachability properties on pointer programs have been already proposed. We list here the main works we are aware of:

- PALE[14] translates both the program and the memory heap in a decidable monadic logic. The tool can be used for lists as well as trees, but it requires annotating the program with loop invariants.
- TVLA [12,13,15] is based on a finite abstraction w.r.t. predicates typically stating that a memory cell points to another one. The approach works for lists and trees, but the user has to provide predicates controlling the preciseness of the abstraction.
- In [8] the authors propose technics based on abstract regular model checking: the structure of the memory heap is represented by regular expressions, and an *abstract,check and refine* procedure is designed to check properties. The authors present encouraging benchmarks.
- Smallfoot[7] is an automatic verification tool that checks separation logic specifications of sequential and concurrent programs that manipulate recursive dynamically-allocated (linked) data structures.
- In [9,10] the authors introduce a logic geared towards quantitative shape analysis.
- In [5] we introduced *pointer systems* to model the class of programs we are interested in. We propose a semi-algorithmic method based on symbolic representations of infinite sets of configurations of the pointer system, called *symbolic memory states (SMS)*. However such an iterative fixpoint computation must be equiped with some adequat widening (or acceleration) techniques in order to help convergence. This adequat widening was missing.
- In [11] (not published), Finkel and Nowak proposed a translation of pointer systems into counter systems. However this translation leads to counter systems labeled by relations instead of functions. As a matter of fact, it was not feasible to re-use verification techniques developed for traditional counter systems.

2 Preliminaries

We present in this section our model of programs (pointer system), our model of the memory heap (memory graph) (see [5]) and counter systems. In the following we consider given two finite, disjoint sets, the set $\mathbb{V}$ of pointer variables and the set $\mathbb{K}$ of counter variables. To avoid ambiguity, we range over pointer variables with $x, y, \ldots$ and over counter variables with $k_1, k_2, \ldots$

2.1 Pointer and counter systems

A systems model is a tuple $\mathcal{M} = (\mathcal{D}, \mathcal{G}, \mathcal{A})$ where $\mathcal{D}$ is an infinite set of data, $\mathcal{G} \subseteq 2^{\mathcal{D}}$ is set of guards, and $\mathcal{A} \subseteq \mathcal{D}^{\mathcal{D}}$ set of actions. We usually note $d \models g$ for $d \in g$ and $\mathsf{post}(a, d)$ for $a(d)$. In the frame of a systems model, we may consider a particular system:

Definition 2.1 (System). A system S in $\mathcal{M} = (\mathcal{D}, \mathcal{G}, \mathcal{A})$ is a pair $S = (Q, \delta)$, where Q is a finite set whose elements are called control states, and δ is a finite subset of $Q \times \mathcal{G} \times \mathcal{A} \times Q$, called the transition relation.

As usual, we often note $q \xrightarrow{g?a} q'$ a given transition (q, g, a, q'). The semantics of a system S is given by means of its associated transition system $TS(S) = (Q \times \mathcal{D}, \rightarrow)$ where $\rightarrow \subseteq (Q \times \mathcal{D})^2$ is defined as:

$$(q, d) \ \rightarrow \ (q', d') \ \text{ iff } \ \exists\, g, a \text{ such that } (q, g, a, q') \in \delta,\ d \models g \text{ and } \mathsf{post}(a, d) = d'$$

We note $\mathsf{Reach}_S(q, d)$ the set $\{(q', d') | (q, d) \rightarrow^* (q', d')\}$ where $\rightarrow^*$ denotes the reflexive and transitive closure of $\rightarrow$.

A standard model of systems is the *counter systems model* $\mathcal{M}_c = (\mathbb{N}^{\mathbb{K}}, \Phi, \mathcal{F})$ where $\mathbb{N}^{\mathbb{K}}$ is the set of counter valuations, Φ is the set of Presburger formulas over $\mathbb{K}$, and $\mathcal{F}$ is the set of linear functions in $\mathbb{N}^{\mathbb{K}}$. We range over with val for valuations, ϕ for a Presburger formula, and f for a linear function.

Pointer systems are systems accessing a heap (in [5] they were called *pointer automata*). We denote the pointer model $\mathcal{M}_p = (\mathcal{D}_p, \mathcal{G}_p, \mathcal{A}_p)$ where $\mathcal{D}_p$ is the set of memory graphs we will define in the next section, and $\mathcal{G}_p$ and $\mathcal{A}_p$ are the set of guards g and actions a, respectively defined by the following grammars:

$$g ::= \mathit{True} \mid \mathit{IsNull}(x) \mid \neg \mathit{IsNull}(x)$$

$$a ::= x{:=}E \mid x.s{:=}E \mid x{:=}\mathbf{new} \mid \mathbf{free}(x) \mid \mathbf{skip}$$

$$\text{where } E \text{ is either } \mathbf{null}, \text{ or } x \text{ or } x.s.$$

The formal definition of the semantics of guards and actions is sketched in the next section. Intuitively, **null** is the null pointer, x denotes the memory cell n_x pointed to by x and $x.s$ denotes the memory cell pointed to by n_x. The actions **free** and **new** respectively deallocate and allocate memory cells.

Example. Figure 1 presents a `reverse` function written in C and the corresponding pointer system in both textual and graphical representation. This example is taken from [13].

2.2 Memory model and semantics

With assumptions made on the class of programs we want to analyze, the memory heap can be modeled as a finite oriented graph whose nodes are

```
/* reverse.c */
#include ‘‘list.h’’
List reverse(List x) {
  List y,t;
  y = NULL;
  while (x!=NULL) {
    t=y;
    y=x;
    x=x->n;
    y->n=t;
    t=NULL;
  }
  return y;
}
```

(1, *True*, y:=**null**, 2),
(2, $\neg$*IsNull*(x), t:=y, 3),
(3, *True*, y:=x, 4),
(4, *True*, x:=$x.s$, 5),
(5, *True*, $y.s$:=t, 6),
(6, *True*, t:=**null**, 2),
(2, *IsNull*(x), **skip**, 7)

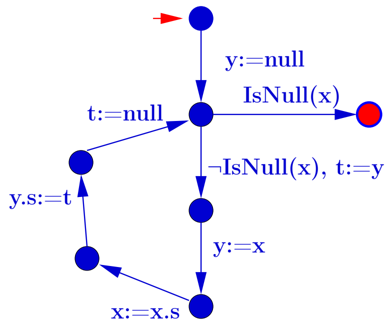


Figure 1. A C program reversing a list and an equivalent pointer system.

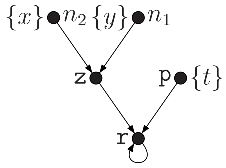
allocated memory cells. An edge from node n_1 to node n_2 indicates that the memory cell n_1 contains the address of memory cell n_2 (n_1 points to n_2). Each node is also labeled by the finite set of pointer variables pointing to the cell. Finally the graph contains three special nodes: `z` models the **null** pointer, `p` models all the memory cells with illegal address and `r` represents an illegal computation (typically, the successor of **null** is `r`). The definition of *memory graph* directly follows.

Definition 2.2 (Memory graph).[5] A *memory graph* MG is a tuple (N, $next$, var) such that

- N is a finite set of nodes, containing three distinguished elements `p`, `r` and `z`.
- $next$ is a total function from N to N, called the successor function, such that $next^{-1}(\mathtt{r}) = \{\mathtt{z}, \mathtt{p}, \mathtt{r}\}$.
- $var : N \longrightarrow 2^{\mathbb{V}}$ is a function such that $\{var(n)\}_{n \in N}$ forms a partition of $\mathbb{V}$.

Each node has exactly one successor since a memory cell points either to a valid memory cell, or to **null** or to an invalid memory cell. Edges are defined by the pairs $(n, next(n))$ for $n \in N$. Figure 2 shows an example of memory graph (empty sets of variables are not written). We will denote $\mathcal{MG}$ the sets of all memory graphs.

We say that a memory graph has a *memory violation* if $var(\mathtt{r}) \neq \emptyset$. It has a *memory leakage* if N contains at least one node that cannot be accessed from any node labeled with at least one variable. A memory graph is said to be *unsafe* if it has a memory violation or a memory leakage, otherwise it is said to be *safe*. As anounced in the previous section, it is possible to give a semantic to $MG \models g$ and $\mathsf{post}(a, MG)$. However, we skip here the formal definition of these notions. Intuitively, post is defined in terms of adding and deleting edges, nodes and labels of memory graphs. For example the instruction `x:=`**new** creates a new node n', moves the x label to it, and set $next(n')$to `p`.



$N = \{n_1, n_2, \mathtt{z}, \mathtt{p}, \mathtt{r}\}$

$next(n_1) = next(n_2) = \mathtt{z}$

$var(n_1) = \{y\}$, $var(n_2) = \{x\}$,

$var(\mathtt{p}) = \{t\}$

Figure 2. Example of memory graph.

Remark 2.3 This behaviour is very closed to the one of C. It is easy to adapt it to various languages and specifications. For example, in Java, the node created by **new** would be linked to `z` rather than `p`.

We precise that we define our semantics in order to detect on-the-fly memory violations and memory leaks: when an unsafe memory graph occurs, the computation stops, because no guard is satisfied by an unsafe memory graph (even $True$).

3 Computing with memory shapes

In this section, we present an abstract view of memory states we call memory shapes. The first section gives a formal definition of this notion, whereas the second section explains how we may define a symbolic computation on these objects.

3.1 Memory shapes

We now introduce memory shapes. Before giving our formal definition, we collect some useful notions on memory graphs. We say that a node n of a memory graph MG is a *core node* if either the input degree of n (that is the number of incoming edges) is different of 1, or n is labeled by at least one pointer variable, or n is one of the three special nodes. A memory graph is said to be *minimal* [5] if it contains only core nodes.

Definition 3.1 (Memory shape). A memory shape is:

- either SegF or MemLeak;
- or a tuple $(N, next, var, K, c)$ where $(N, next, var)$ is a safe, minimal memory graph, $K \subseteq \mathbb{K}$ is a finite set of counter variables and $c : N \setminus \{\mathtt{p}, \mathtt{z}, \mathtt{r}\} \to K$ is a bijection.

We will denote $\mathcal{MS}$ the set of all memory shapes. We call *valued memory shape* a pair (MS, val) where MS is a memory shape and $val : K \to \mathbb{N}^*$ maps each counter in MS to a strictly positive integer. Intuitively, a valued memory shape represents, in a more compact way, a memory graph without memory leak: each $c(n)$ labeling the edge (n, n') represents the succession of $val(c(n))$ edges in the original memory graph. Conversely, any safe memory graph can be represented by an adequate valued memory shape. This gives us a function $\langle . \rangle : \mathcal{MS} \times \mathbb{N}^{\mathbb{K}} \to \mathcal{MG}$ that associates to a valued memory shape (MS, val) the corresponding memory graph $\langle MS, val\rangle$. This function is surjective on safe memory graphs, and in practice we turn it into a bijection adding a counter labeling discipline in memory shapes (but this point is not relevant for this presentation). We recall below two important properties of memory shapes [5,9,2].

Theorem 3.2 *The two following properties hold for memory shapes with a set $\mathbb{V}$ of pointers variables:*

(i) $|\mathcal{MS}| \leq (2 \cdot |\mathbb{V}|)^{3 \cdot |\mathbb{V}|}$.

(ii) the number of counters in a memory shape is bounded by $2 \cdot |\mathbb{V}|$*:*

Remark 3.3 For the sake of simplicity, we do not present here a discipline on counter labeling. This point is somehow irrelevant in this presentation, except to gain a bijection as we will mention later on. But it can be noted that the isomorphism of memory shapes up to counter labeling is decidable in linear time, which we hardly use in practice.

3.2 Symbolic computation

In this section, we introduce the symbolic computation on memory shapes. It relies on two functions TEST and POST that lift at the abstract level the concrete computation. We will now give the intuition on how we can take advantage of our notion of memory shape to compute these functions.

The TEST function has to decide wether a given memory shape satisfies a guard, that is all underlying memory graphs satisfy it. Definition 3.1 ensures that only the z node of the memory shape can be mapped to the z node of the memory graph. So checking a guard $IsNull(x)$ simply boils down to check if x labels the z node in the memory shape.

The POST function has to produce, from a given memory shape and a pointer action, the set $\{(\phi_i, f_i, MS_i)\}$ of all possible issues: here the ϕ_i's define mutual excluding conditions on counters that ensure that a unique memory shape will be reached in each case. For every guard, the corresponding memory shape MS_i is computed, as the linear function f_i updating counters accordingly.

We skip here the formal definition of this function, and rely on a particular example to clarify our point. The interested reader will find in appendix a more detailed presentation.

Example 3.4 Consider the memory shape MS_1 represented in figure 3 and the pointer action $x.s := y$. The POST function will return the set $\big\{\big(k_3 = 1, f : (k_1, k_2, k_3) \rightarrow (k1 + k2, 0, k_3), MS_2\big)\ ,\ \big(k_3 > 1, id_{\mathbb{K}}, \mathsf{MemLeak}\big)\big\}$ in which MS_2 denotes the memory shape represented in figure 3. Intuitively, if the counter associated to the pointer variable x is strictly greater than 1, the action $x.s := y$ will lead to a memory leak. Otherwise, the shape MS_2 is reached, the edges labeled with k_1 and k_2 are collapsed, and one generates the counter action $k_1 := k_1 + k_2$.

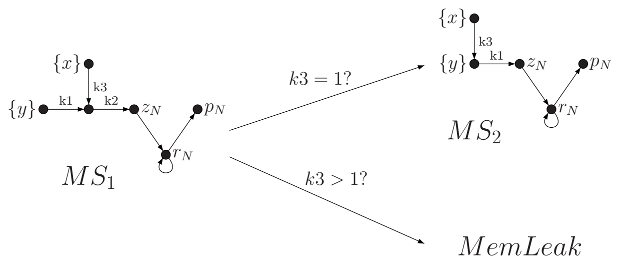


Figure 3. Effect of the action x.s:=y on a memory shape MS_1

More formally, the symbolic test function $\mathsf{TEST} : \mathcal{MS} \times \mathcal{G} \rightarrow \{0,1\}$ and the symbolic computation function $\mathsf{POST} : \mathcal{A} \times \mathcal{MS} \rightarrow 2^{\Phi \times \mathcal{F} \times \mathcal{MS}}$, are defined such that they enjoy the following properties:

Proposition 3.5 (i) *the memory shape holds enough information for deciding the guard, that is* $\mathsf{TEST}(MS, g) = 0$ *iff for all counter valuation val,* $\langle MS, val\rangle \not\models g$*, and* $\mathsf{TEST}(MS, g) = 1$ *iff for all counter valuation val,* $\langle MS, val\rangle \models g$*,*

(ii) *the symbolic computation mimicks at the abstract level the concrete computation: if* $\mathsf{POST}(a, MS) = \{(\phi_i, f_i, MS_i)\}_{i \in I}$*, then the* g_i*'s form a partition of* $\mathbb{N}^{\mathbb{K}}$*, and for all valuation, if* $val \models \phi_i$*,* $\mathsf{post}(a, \langle MS, val\rangle) = \langle MS_i, f_i(val)\rangle$*.*

4 Translation of pointer systems into counter systems

In this section, we present a translation from pointer systems to counter systems using the notions of memory shape and symbolic computation we have just defined. We first state the definition of the counter system and its soundness with respect to the pointer system it is built from. In a second time, we present an algorithm to construct effectively this counter system.

4.1 Principle

Equipped with the functions TEST and POST previously introduced, the translation of a pointer system (Q_p, δ_p) into a counter system (Q_c, δ_c) is defined by $Q_c = Q_p \times \mathcal{MS}$ and δ_c is defined as follows:

$$\bigcup_{q \xrightarrow{g_p ? a_p} q' \in \delta_p} \quad \bigcup_{MS \,:\, \mathsf{TEST}(MS, g) = 1} \left\{ \begin{array}{l} \langle q, MS\rangle \xrightarrow{g_i ? a_i} \langle q', MS_i\rangle \\ \text{with } \mathsf{POST}(a, MS) = \{(g_i, a_i, MS_i)\}_{i \in I} \end{array} \right\}$$

Note that δ_c is a finite part of $Q_c \times \Phi \times \mathcal{F} \times Q_c$, hence (Q_c, δ) actually defines a counter system.

Theorem 4.1 (Soundness of translation) *Given a pointer system (Q_p, δ_p), and (Q_c, δ_c) the counter system defined above, there is a bisimulation between both underlying transitions systems.*

According to Proposition 3.5, the relation

$$\mathcal{R} = \left\{ \left((q, \langle MS, val\rangle)\ , ((q, MS), val)\right) \middle| \ q \in Q_p, MS \in \mathcal{MS}, val \in \mathbb{N}^{\mathbb{K}} \right\}$$

is a bisimulation between both transition systems.

4.2 Translation algorithm

The algorithm works with a set $Current$ of the control states (q, MS) known to be reachable and for which outgoing transitions have not been computed yet, and a set $Treated$ of reachable control states that have already been treated ensuring single-pass translation.

Algorithm 1 From pointer system to counter system

Input : (Q_p, δ_p) a pointer system;
Output : (Q_c, δ_c) a counter system;

```
for all ∀(q_0, MS_0) ∈ Q_p × 𝓜𝓢 do
  Current ← {(q_0, MS_0)};
  Q_c ← {(q_0, MS_0)};
  Treated ← ∅;
  while Current ≠ ∅ do
    pick (q, MS) in Current \ Treated;
    for all q --g?a--> q' ∈ δ_p; do
      if MS ⊨ g_p then
        for all (φ, f, MS') ∈ POST(a, MS) do
          δ_c ← δ_c ∪ {(q, MS) --φ?f--> (q', MS')};
          Q_c ← Q_c ∪ {(q', MS')};
          Current ← Current ∪ {(q', MS')};
        end for
      end if
    end for
    Treated ← Treated ∪ {(q, MS)};
    Current ← Current \ {(q, MS)}
  end while
end for
```

Theorem 4.2 ***(Soundness of the algorithm)*** *The algorithm 1 terminates and it computes the counter system defined in section 4.1.*

Remark 4.3 In practice, we use this algorithm for a fixed initial control state q_0 and a fixed initial memory shape MS_0. This gives a much smaller counter system (Q_0, δ_0) such that $Q_0 \subseteq Q_c$ and $\delta_0 \subseteq \delta_c$, (Q_0, δ_0) is the strongly connected component of the control graph of (Q_c, δ_c) containing (q_0, MS_0). We explain below why this is sufficient for our analysis.

The complexity results of Theorem 3.2 argues for termination and efficiency of the algorithm. Moreover, Theorem 4.1 ensures the soundness of the algorithm.

Remark 4.4 A first partial translation from pointer systems to counter systems has been made in [11].

5 Analysis of pointer systems

5.1 Analysis of the counter system

As a consequence of the bisimulation result, the set of memory shapes appearing in the control states of (Q_0, δ_0) is an over-approximation of the memory shapes actually reachable in the original pointer system. More formally, we define the function $\mathsf{Abs} : Q_p \times \mathcal{MG} \to Q_p \times \mathcal{MS}$ mapping $(q, \langle MS, val\rangle)$ onto (q, MS). Then our over-approximation result can be stated as follows:

Theorem 5.1 ***(Over-approximation)*** *For all $q_0 \in Q_p$, for all $MS_0 \in \mathcal{MS}$, and for all $val \in \mathbb{N}^{\mathbb{K}}$, $\mathsf{Abs}(\mathsf{Reach}_{S_p}((q, \langle MS_0, val\rangle))) \subseteq Q_0$.*

This result allows us to perform a verification of the pointer system in two passes. The first step consists in the static analysis of the counter system. We directly check on Q_0 if we produced states (q, SegF) or $(q, \mathsf{MemLeak})$. If not, we may directly conclude that the pointer system is safe. Otherwise, we have to check if the unsafe states are actually reachable in the transition system. For this, we rely on the tool FAST[3] and the techniques of acceleration that are implemented in it.

5.2 The reverse function example

We illustrate our analysis on the program reversing a list introduced in figure 1 applied on a non empty single-linked list. So we set MS_0 to be the corresponding memory shape and generate the counter system (Q_0, δ_0). We then observe that no control state contains the memory shape SegF, but one state contains MemLeak. Hence after this first step we can conclude that there will not be a memory violation. In order to know if a memory leak might happen, we analyze the counter system using FAST. This second step tells us that the control state containing the memory shape MemLeak will not be reached in the pointer system (for any valuation).

6 Perspectives

We defined a translation from pointer systems to counter systems and used it to give both a qualitative and quantitative analysis of the pointer system. While doing this, we believe we defined a general framework for both types of analysis, which was not so clearly stated in other works. In order to tackle more complex examples, we are currently working on a better integration of the quantitative analysis at the translation stage.

Acknowledgements We thank David Nowak and Philippe Schnoebelen for enlighting discussions.

References

[1] A. Annichini, A. Bouajjani and M. Sighireanu TReX: A Tool for Reachability Analysis of Complex Systems In *lncs2102*, SV, pp 368-372, 2001.

[2] S. Bardin Vers un Model Checking Avec Accélération Plate des Systèmes Hétérogènes. PhD Thesis. October 2005.

[3] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *CAV'03*, LNCS 2725. Springer, 2003.

[4] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *ATVA'05*, LNCS 3707. Springer, 2005.

[5] S. Bardin, A. Finkel, and D. Nowak. Toward symbolic verification of programs handling pointers. In *AVIS'04*, ENTCS. Elsevier Science Publishers.

[6] S. Bardin, A. Finkel, and J. Leroux. FASTer acceleration of counter automata. In *TACAS'04*, LNCS 2988, Springer, 2004.

[7] J. Berdine, C. Calcagno, P.W. O'Hearn Symbolic Execution with Separation Logic. In *APLAS'05*, LNCS 3780, pp. 52-68, 2005.

[8] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *TACAS'05*, LNCS 3440. Springer, 2005.

[9] M. Bozga and R. Iosif. On decidability within the arithmetic of addition and divisibility. In *FOSSACS'2005*, LNCS 3441. Springer, 2005.

[10] M. Bozga and R. Iosif. Quantitative Verification of Programs with Lists In *VISSAS'05*, IOS Press, NATO Science Series, 2005.

[11] A. Finkel and D. Nowak From Pointer Automata to counter Automata. Draft. April 2005.

[12] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. *ACM SIGSOFT Software Engineering Notes*, 25(5):26–38, 2000.

[13] T. Lev-Ami and M. Sagiv. Tvla: A system for implementing static analysis. In *SAS'00*, LNCS 1824. Springer, 2000.

[14] A. Møller and M. Schwartzbach. The pointer assertion logic engine. In *ACM PLDI'01)*, volume 36.5 of *ACM SIGPLAN Notices*. ACMPress 2001.

[15] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[16] Lash homepage : *http://www.montefiore.ulg.ac.be/ boigelot/research/lash/*

[17] Brain homepage : *http://www.cs.man.ac.uk/ voronkov/BRAIN/index.html*

[18] Alv homepage: *http://www.cs.ucsb.edu/ bultan/composite/*

A Translation Technical Points: successors of memory shape

We describe here the function denoted POST which produces from a given memory shape and a pointer action a set of triples $\{(\phi_i, f_i, MS_i)\}$ describing all the possible issues; the ϕ_i's define mutual excluding conditions on counters that ensure that a unique memory shape will be reached in each case. For every guard, the corresponding memory shape MS_i is computed, as the linear function f_i updating counters accordingly. The eight algorithms that we give correspond to the eight kind of actions which can appear in a pointer automaton. For each function, we suppose that the given memory shape is denoted $MS = \langle N, next, var, K, c\rangle$.

Algorithm 2 Algorithm of $\mathsf{POST}_{x:=null}$

Let $n \in N$ such that $x \in var(n)$;
if $n = \mathtt{z}$ **then**
  Return $\{(True, \mathbf{id}, MS)\}$
else
  if $n = \mathtt{p}$ or $var(n) \neq \{x\}$ or $deg(n) > 1$ **then**
    Return $\{(True, \mathbf{id}, MS')\}$ where $MS' = \langle N, next, var', K, c\rangle$ with $var'(\mathtt{z}) = var(\mathtt{z}) \cup \{x\}$, $var'(n) = var(n) \setminus \{x\}$ and $\forall m \in N \setminus \{\mathtt{z}, n\}$, $var'(m) = var(m)$;
  else
    if $deg(n) = 0$ **then**
      Return $\{(True, \mathbf{id}, MemLeak)\}$;
    else
      Let n' be the node such that $next(n') = n$, k and k' the counter variables such that $c(n) = k$ and $c(n') = k'$;
      Return $\{(True, [k' := k{+}k'; k = 0], MS')\}$ where $MS' = \langle N\setminus\{n\}, next', var', K\setminus\{k\}, c'\rangle$ with :
      - $var'(\mathtt{z}) = var(\mathtt{z}) \cup \{x\}$, $\forall m \in N \setminus \{n, \mathtt{z}\}$, $var'(m) = var(m)$,
      - $next'(n') = next(n)$ and $\forall m \in N \setminus \{n, n'\}$, $next'(m) = next(m)$,
      - $\forall m \in N \setminus \{n, \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c'(m) = c(m)$;
    end if
  end if
end if

Algorithm 3 Algorithm of $\mathsf{POST}_{x:=y}$

Let $n, m \in N$ such that $x \in var(n)$ and $y \in var(m)$;
if $n = m$ **then**
  Return $\{(True, \mathbf{id}, MS)\}$;
else
  if $n = \mathtt{z}$ or $n = \mathtt{p}$ or $var(n) \neq \{x\}$ or $deg(n) > 1$ **then**
    Return $\{(True, \mathbf{id}, MS')\}$ where $MS' = \langle N, next, var', K, c\rangle$ with $var'(m) = var(m) \cup \{x\}$, $var'(n) = var(n) \setminus \{x\}$ and $\forall n' \in N \setminus \{m, n\}$, $var'(n') = var(n')$;
  else
    if $deg(n) = 0$ **then**
      Return $\{(True, \mathbf{id}, MemLeak)\}$;
    else
      Let n' be the node such that $next(n') = n$, k and k' the counter variables such that $c(n) = k$ and $c(n') = k'$;
      Return $\{(True, [k' := k' + k; k := 0], MS')\}$ where $MS' = \langle N \setminus \{n\}, s', var', K \setminus \{k\}, c'\rangle$ with :
      • $var'(m) = var(m) \cup \{x\}$, $\forall p \in N \setminus \{n, m\}$, $var'(p) = var(p)$,
      • $next'(n') = next(n)$ and $\forall p \in N \setminus \{n, n'\}$, $next'(p) = next(p)$,
      • $\forall p \in N \setminus \{n, \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c(p) = c(p)$;
    end if
  end if
end if

Algorithm 4 Algorithm of $\mathsf{POST}_{x.s:=null}$

Let $n, n' \in N$ such that $x \in var(n)$ and $n' = next(n)$;
if $n = \mathtt{p}$ or $n = \mathtt{z}$ **then**
  Return $\{(True, \mathbf{id}, SegF)\}$;
else
  if $deg(n') = 2$ and $var(n') = \emptyset$ and $n' \notin \{\mathtt{z}, \mathtt{p}, \mathtt{r}\}$ **then**
    Let n'' be the node such that $next(n'') = n'$ and k, k', k'' be the counter variables such that $c(n) = k$, $c(n') = k'$ and $c(n'') = k''$;
    Return $\{(k = 1, [k'' = k' + k''; k' = 0], MS'); (k > 1, \mathbf{id}, MemLeak)\}$ where $MS' = \langle N \setminus \{n'\}, next', var', K \setminus \{k'\}, c'\rangle$ with :
    • $\forall p \in N \setminus \{n'\}$, $var'(p) = var(p)$,
    • $next'(n'') = next(n')$, $next'(n) = \mathtt{z}$, $\forall p \in N \setminus \{n'', n, n'\}$, $next'(p) = next(p)$,
    • $\forall p \in N \setminus \{n', \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c'(p) = c(p)$;
  else
    Let k be the counter variable such that $c(n) = k$;
    Return $\{(k = 1, \mathbf{id}, MS'); (k > 1, \mathbf{id}, MemLeak)\}$ where $MS' = \langle N, next', var, K, c\rangle$ with :
    • $next'(n) = \mathtt{z}$ and $\forall p \in N \setminus \{n\}$, $next'(p) = next(p)$;
  end if
end if

Algorithm 5 Algorithm of $\mathsf{POST}_{x.s:=y}$

Let $n, m, n' \in N$ such that $x \in var(n)$, $y \in var(m)$ and $n' = next(n)$;
if $n = \mathtt{p}$ or $n = \mathtt{z}$ **then**
  Return $\{(True, \mathbf{id}, SegF)\}$;
else
  if $deg(n') = 2$ and $var(n') = \emptyset$ and $n' \notin \{\mathtt{z}, \mathtt{p}, \mathtt{r}\}$ **then**
    Let n'' be the node such that $next(n'') = n'$ and k, k', k'' be the counter variables such that $c(n) = k$, $c(n') = k'$ and $c(n'') = k''$;
    Return $\{(k = 1, [k'' = k' + k''; k' = 0], MS'); (k > 1, \mathbf{id}, MemLeak)\}$ where $MS' = \langle N \setminus \{n'\}, next', var', K \setminus \{k'\}, c'\rangle$ with :
    • $\forall p \in N \setminus \{n'\}$, $var'(p) = var(p)$,
    • $next'(n'') = next(n')$, $next'(n) = m$, $\forall p \in N \setminus \{n'', n, n'\}$, $next'(p) = next(p)$,
    • $\forall p \in N \setminus \{n', \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c'(p) = c(p)$;
  else
    Let k be the counter variable such that $c(n) = k$;
    Return $\{(k = 1, \mathbf{id}, MS'); (k > 1, \mathbf{id}, MemLeak)\}$ where $MS' = \langle N, next', var, K, c\rangle$ with :
    • $next'(n) = m$ and $\forall p \in N \setminus \{n\}$, $next'(p) = next(p)$;
  end if
end if

Algorithm 6 Algorithm of $\mathsf{POST}_{x.s:=y.s}$

Let $n, m, n', m' \in N$ such that $x \in var(n)$, $y \in var(m)$, $n' = next(n)$ and $m' = next(m)$;
if $n = \mathtt{p}$ or $n = \mathtt{z}$ or $m = \mathtt{p}$ or $m = \mathtt{z}$ **then**
  Return $\{(True, \mathbf{id}, SegF)\}$;
else
  if $n' = m'$ **then**
    Return $\{(True, \mathbf{id}, MS)\}$
  else
    if $deg(n') = 2$ and $var(n') = \emptyset$ and $n' \notin \{\mathtt{z}, \mathtt{p}, \mathtt{r}\}$ **then**
      Let n'' be the node such that $next(n'') = n'$ and k, k', k'', l be the counter variables such that $c(n) = k$, $c(n') = k'$,$c(n'') = k''$ and $c(m) = l$;
      Return $\{(k = 1 \wedge l = 1, [k'' = k' + k''; k' = 0], MS'); (k > 1, \mathbf{id}, MemLeak); (k = 1 \wedge l > 1, [k'' = k' + k''; k' = 1; l := l - 1], MS'')\}$ where $MS' = \langle N \setminus \{n'\}, next', var', K \setminus \{k'\}, c'\rangle$ with :
      • $\forall p \in N \setminus \{n'\}$, $var'(p) = var(p)$,
      • $next'(n'') = next(n')$, $next'(n) = m'$, $\forall p \in N \setminus \{n'', n, n'\}$, $next'(p) = next(p)$,
      • $\forall p \in N \setminus \{n', \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c'(p) = c(p)$;
      and $MS'' = \langle N, next'', var, K, c\rangle$ with :
      • $next''(m) = n'$, $next''(n') = m'$ and $\forall p \in N \setminus \{m, n'\}$, $next''(p) = next(p)$;
    else
      Let k, l be the counter variables such that $c(n) = k$ and $c(m) = l$;
      Return $\{(k = 1 \wedge l = 1, \mathbf{id}, MS'); (k > 1, \mathbf{id}, MemLeak); (k = 1 \wedge l > 1, [new\_k := 1; l := l - 1], MS'')\}$ where $MS' = \langle N, next', var, K, c\rangle$ with :
      • $next'(n) = m'$ and $\forall p \in N \setminus \{n\}$, $next'(p) = next(p)$;
      and $MS'' = \langle N \cup \{new\_n\}, next'', var'', K \cup \{new\_k\}, c''\rangle$ with :
      • $new\_n \notin N$,
      • $new\_k \notin K$,
      • $var''(new\_n) = \emptyset$ and $\forall p \in N$,$var''(p) = var(p)$,
      • $next''(n) = new\_n$, $next''(m) = new\_n$, $next''(new\_n) = next(m)$ and $\forall p \in N \setminus \{n, m\}$, $next''(p) = next(p)$,
      • $c''(new\_n) = l$, $c''(m) = new\_k$ and $\forall p \in N \setminus \{m, \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c''(p) = c(p)$.
    end if
  end if
end if

Algorithm 7 Algorithm of $\mathsf{POST}_{x:=y.s}$

Let $n, m, m' \in N$ such that $x \in var(n)$, $y \in var(m)$ and $m' = next(m)$;
if $m = \mathtt{p}$ or $m = \mathtt{z}$ **then**
  Return $\{(True, \mathbf{id}, SegF)\}$;
else
  Let l be the counter variable such that $c(m) = l$;
  if $n = \mathtt{z}$ or $n = \mathtt{p}$ or $var(n) > 1$ or $deg(n) > 1$ **then**
    Return $\{(l = 1, \mathbf{id}, MS'); (l > 1, [new\_k := 1; l := l-1], MS'')\}$ where $MS' = \langle N, next, var', K, c\rangle$ with :
    • $var'(m') := var(m') \cup \{x\}$, $var'(n) = var(n) \setminus \{x\}$ and $\forall p \in N \setminus \{n, m'\}$, $var'(p) = var(p)$;
    and $MS'' = \langle N \cup \{new\_n\}, next'', var'', K \cup \{new\_k\}, c''\rangle$ with :
    • $new\_n \notin N$,
    • $new\_k \notin K$,
    • $next''(new\_n) = m'$, $next''(m) = new\_n$, $\forall p \in N \setminus \{m\}$, $next''(p) = next(p)$,
    • $var''(new\_n) = \{x\}$, $var''(n) = var(n) \setminus \{x\}$ and $\forall p \in N \setminus \{n\}$, $var''(p) = var(p)$,
    • $c''(new\_n) = l$, $c''(m) = new\_k$ and $\forall p \in N \setminus \{m, \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c''(p) = count(p)$;
  else
    if $deg(n) = 0$ **then**
      Return $\{(True, \mathbf{id}, MemLeak)\}$;
    else
      Let n' be the node such that $next(n') = n$ and let k' be the counter variable such that $c(n') = k'$;
      if $x = y$ **then**
        if $n = m'$ **then**
          Return $\{(True, \mathbf{id}, MS)\}$
        else
          Return $\{(l = 1, [k' := k'+1; l := 0], MS'); (l > 1, [k' = k'+1; l := l-1], MS)\}$ where $MS' = \langle N \setminus \{n\}, next', var', K \setminus \{l\}, c'\rangle$ with
          • $next'(n') = m'$ and $\forall p \in N \setminus \{n, n'\}$, $next'(p) = next(p)$,
          • $var'(m') = var(m') \cup \{x\}$ and $\forall p \in N \setminus \{n, m'\}$, $var'(p) = var(p)$,
          • $\forall p \in N \setminus \{n, \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c'(p) = c(p)$
        end if
      else
        Let k be the counter variable such that $c(n) = k$;
        if $n = m'$ **then**
          Return $\{(True, [l := 1; k := k + l - 1], MS)\}$;
        else
          Return $\{(l = 1, [k' := k' + k; k := 0], MS'); (l > 1, [k' := k' + k; k := 1; l := l-1], MS'')\}$ where $MS' = \langle N \setminus \{n\}, next', var', K \setminus \{k\}, c'\rangle$ with ;
          • $next'(n') = next(n)$ and $\forall p \in N \setminus \{n, n'\}$, $next'(p) = next(p)$,
          • $var'(m') = var(m') \cup \{x\}$ and $\forall p \in N \setminus \{m', n\}$, $var'(p) = var(p)$,
          • $\forall p \in N \setminus \{n\}$, $c'(p) = c(p)$;
          and $MS'' = \langle N, next'', var, K, c''\rangle$ with :
          • $next''(n') = next(n)$, $next''(m) = n$, $next(n) = m'$ and $\forall p \in N \setminus \{n', m', n\}$, $next''(p) = next(p)$,
          • $c''(m) = k$, $c''(n) = l$ and $\forall p \in N \setminus \{n, m, \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c''(p) = count(p)$
        end if
      end if
    end if
  end if
end if

Algorithm 8 Algorithm of $\mathsf{POST}_{New(x)}$

Let $n \in N$ such that $x \in var(n)$;
if $n = \mathtt{z}$ or $n = \mathtt{p}$ or $var(n) \neq \{x\}$ or $deg(n) > 1$ **then**
  Return $\{(True, [new\_k := 1], MS')\}$ where $MS' = \langle N \cup \{new\_n\}, next', var', K \cup \{new\_k\}, c'\rangle$ with :
  • $new\_n \notin N$,
  • $new\_k \notin C$,
  • $next'(new\_n) = \mathtt{p}$ and $\forall p \in N$, $next'(p) = next(p)$,
  • $var'(new\_n) = \{x\}$, $var'(n) = var(n) \setminus \{x\}$ and $\forall p \in N \setminus \{n\}$, $var'(p) = var(p)$,
  • $c'(new\_n) = new\_k$ and $\forall p \in N \setminus \{\mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c'(p) = c(p)$;
else
  if $deg(n) = 0$ **then**
    Return $\{(True, \mathbf{id}, MemLeak)\}$;
  else
    Let n' be the node such that $next(n') = n$, k and k' the counter variables such that $c(n) = k$ and $c(n') = k'$;
    Return $\{(True, [k' := k' + k; k := 1], MS')\}$ where $MS' = \langle N, next', var, K, c\rangle$ with :
    • $next'(n') = next(n)$, $next'(n) = \mathtt{z}$ and $\forall p \in N \setminus \{n, n'\}$, $next'(p) = next(p)$,
  end if
end if

Algorithm 9 Algorithm of $\mathsf{POST}_{Free(x)}$

Let $n \in N$ such that $x \in var(n)$;
if $n = \mathtt{z}$ **then**
  Return $\{(True, \mathbf{id}, MS)\}$;
else
  if $n = \mathtt{p}$ **then**
    Return $\{(True, \mathbf{id}, SegF)\}$;
  else
    Let k be the counter variable sucht that $c(n) = k$;
    Return $\{(k = 1, [k := 0], MS'), (k > 1, \mathbf{id}, MemLeak)\}$ where $MS' = \langle N \setminus \{n\}, next', var', K \setminus \{k\}, c'\rangle$ with :
    • $\forall p \in N \setminus \{n\}$ such that $next(p) = n$, $next'(p) = \mathtt{p}$ and $\forall p \in N \setminus \{n\}$ such that $next(p) \neq n$, $next'(p) = next(p)$,
    • $var'(\mathtt{p}) = var(\mathtt{p}) \cup var(n)$ and $\forall p \in N \setminus \{\mathtt{p}, n\}$, $var'(p) = var(p)$,
    • $\forall p \in N \setminus \{n, \mathtt{z}, \mathtt{p}, \mathtt{r}\}$, $c'(p) = c(p)$;
  end if
end if