Toward Symbolic Verification of Programs Handling Pointers [1]

Sébastien Bardin Alain Finkel David Nowak

LSV, ENS Cachan & CNRS UMR 8643

Abstract

We aim at checking safety properties on systems with pointers which are naturally infinite state systems. In this paper, we introduce Symbolic Memory States, a new symbolic representation well suited to the verification of systems with pointers. We show SMS enjoys all the good properties needed to check safety properties, such as closure under union, canonicity of the representation and decidable inclusion. We also introduce pointer automata, a model for programs using dynamic allocation of memory. We define the properties we want to check in this model and we give undecidability results. The verification part is still work in progress.

Key words: memory, model checking, symbolic representation

1 Introduction

During the past 20 years, model checking [3] has developed into a key approach to verifying finite-state systems. It has a number of advantages over traditional approaches that are based on simulation, testing, and deduction. In particular, model checking is automatic, and provides counter-examples in case of error. Model checking has been applied successfully to industrial systems.

However little research has been conducted on systems using dynamic allocation of memory. This is all the more surprising since pointers are a both very useful and very error-prone feature available for a long time in programming languages.

Common safety properties to check on software using pointers are the absence of *segmentation fault* and *memory leak*. A segmentation fault occurs when one tries to access a memory cell which has been deallocated. Then the program crashes. A memory leak occurs when a memory cell can no longer be reached because there is no pointer to it. It can cause software failure if this happens an unbounded number of times. The main difficulty in

[1] Partially supported by contract P11L03/F01304/0 between EDF and LSV.

analyzing software with pointers is aliasing: two syntactically different pointer expressions can refer to the same memory cell.

We aim at checking safety properties on systems with pointers, using the model-checking framework. Since the number of allocated memory cells is generally unbounded, usual enumerative methods to compute the reachability set cannot be ensured to terminate: they only provide a semi-algorithm. In the past few years, a well studied technique to deal with infinite systems in model-checking has emerged: (1) symbolic representations of infinite sets of reachable states by finite structures, and (2) acceleration or widening to compute infinite sets of successors in a single step.

In this paper, we address the problem of finding a symbolic representation adapted to programs with pointers. We introduce Symbolic Memory States (for short, SMSs), a symbolic representation well suited to the verification of systems with pointers. SMSs enjoy all the good properties needed to check safety properties: closure under union, canonicity and decidable inclusion. We also introduce pointer automata, a model for programs using dynamic allocation of memory. We define the properties we want to check in this model and we give undecidability results. The verification part is still work in progress. We briefly sketch acceleration techniques we are currently developing for this model.

Related work.

Few formal methods provide assistance in the area of assurance of software systems using pointers. In the field of static analysis, pointer alias analysis [7] consists in checking if two pointers may refer to the same memory cell. But this method lacks precision and returns lots of spurious warnings. Shape analysis [9], based on abstract interpretation, computes automatically an abstraction of the reachable states of a small program. But the user has to provide the abstraction, and the returned abstraction is not automatically analyzed. The analysis by the user turns out to be quite tricky [9]. In PALE [10], the user annotates the program with invariants, which are then automatically checked. But finding the invariants is difficult. SMS's are based on ideas similar to those in [9]: the aggregation of intermediate nodes of a list. However, in an SMS the exact number of intermediate nodes is kept as a formula on some counters associated with edges, while in TVLA [9] this information is reduced to a value in 3-valued logic, for "there is a successor for sure", "there is no successor for sure", and "there may be some successors". This way termination is guaranteed in TVLA, but the loss of information leads to abstractions difficult to analyze. On the contrary an SMS is an exact representation, but the computation of the reachability set is not guaranteed. This is why acceleration techniques are needed in order to help the convergence of the semi-algorithm. This approach has already been successfully conducted on systems with counters [2,4,6], or systems with clocks and lossy channels [1].

Outline.

In Section 2, we define SMSs, and prove that they enjoy all the good properties needed to check safety properties. We then introduce in Section 3 pointer automata, which are automata extended with pointer variables and common operations on these variables. In Section 4, we formalize segmentation fault and memory leak as reachability properties on the transition system associated with a pointer automaton and its initial SMS, and we show that these properties are undecidable in general. Finally, in Section 5, we sketch our current research on pointer automata verification.

2 Symbolic Memory State

The memory heap is the place in the memory where allocated memory cells are stored. Roughly speaking, this is a collection of memory cells containing data and addresses in the memory heap. We distinguish three kinds of addresses: either it is a valid address of an existing memory cell in the memory heap; or it is an invalid address *and we cannot check it*; or it is a special value (`NULL` in the programming language C) meaning the address is invalid *and we can check it*. In C, a program manipulates invalid addresses only when a pointer is declared but not initialized, or when a memory cell is allocated and then deallocated.

2.1 Restrictions of the model

For our analysis, we do not consider the full power of pointer programming. We make three restrictions.

- First we do not consider data, so we can eliminate data in our representation of the memory heap.
- Second we also do not consider pointer arithmetic, i.e. we are not interested in exact value of addresses of memory cells, but only on the fact that one memory cell has a pointer to another one.
- Finally we impose that a memory cell has at most one pointer to another memory cell. So we restrict ourselves to programs manipulating linear linked lists and cyclic lists.

2.2 Memory graph

We can represent the memory heap as an oriented graph, whose nodes are the valid memory cells allocated so far, plus two special nodes corresponding to the `NULL` pointer, and all the invalid addresses in the memory. We also have a third special node which is the error node. It will denote invalid pointer expressions, typically the successor of a memory cell which contains an invalid address or the `NULL` value. An edge from a node n_1 to a node n_2 denotes that the memory cell n_1 contains the address of the memory cell n_2. A node is also labeled by a set of variables (possibly empty) which represents the pointer variables which contain its address.

Let X be a finite set of variables.

Definition 2.1 A memory graph is either the special graph *SegF* where $N_{SegF} = \emptyset$, or a tuple $G = (N_G, z_G, p_G, r_G, E_G, var_G)$ where:

- N_G is the finite set of nodes,
- z_G, p_G and r_G are respectively the *null* node, the *sink* node and the *error* node,
- E_G is the finite set of edges,
- var_G is a function from N_G to 2^X,

and such that:

- for all nodes $n \in N_G$, there exists a unique node $n' \in N_G$ such that $(n, n') \in E_G$, in other words E_G is a functional relation;
- (z_G, r_G), (p_G, r_G) and (r_G, r_G) are in E_G;
- for all $x \in X$, there exists a unique node $n \in N_G$ such that $x \in var_G(n)$.

Let *VarCptSet* be a countably infinite set of counter variables. A *labeled* memory graph is a pair (G, c) where G is a memory graph and c is an injective function from E_G to *VarCptSet*. Note that in the case of a labeled memory graph $(SegF, c)$, c is necessarily the empty function. We implicitly identify labeled memory graphs which only differ in their choice of counter variables.

Two labeled memory graphs (G, c) and (G', c') are *isomorphic* iff there exists a bijective function $\varphi : N_G \to N_{G'}$ such that for all $n_1, n_2 \in N_G$, $(n_1, n_2) \in E_G$ iff $(\varphi(n_1), \varphi(n_2)) \in E_{G'}$, for all $n \in N_G$, $var(n) = var(\varphi(n))$.

2.3 Symbolic Memory State

We define Symbolic Memory States (for short, SMSs) which are symbolic representations of possibly infinite sets of memory graphs. They are memory graphs whose edges are labeled by counters constrained by a formula. Intuitively, an edge between two nodes will represent a list whose length is the value of the labeling counter. We also define a concretization function which associates to an SMS the set of memory graphs it represents.

Definition 2.2 An atomic SMS is defined by:

- If (G, c) is a labeled memory graph and ϕ is a Presburger formula whose free variables are in $Im(c)$ (where $Im(c)$ is the image of c), then (G, c, ϕ) is an atomic SMS;
- An SMS is a finite set of atomic SMSs.

Suppose (G, c, ϕ) is an atomic SMS. An *assignment satisfying* ϕ is a function $v : Im(c) \to \mathbb{N}$ such that $\phi[k_i{:=}v(k_i)]$ is true. The assignment of (G, c, ϕ) with respect to v is the memory graph G' built from G where each arrow

(s_k, t_k) labeled by a counter variable k is replaced by a chain of $v(k)-1$ intermediate nodes, connecting the source s_k with the first and the last with the target t_k. Intuitively, k encodes the length of the path between s_k and t_k.

The *concretization function* γ is defined inductively by:

- $\gamma(G, c, \phi)$ is the set of all possible assignments of (G, c, ϕ), if G is different of *SegF*,
- $\gamma(\mathit{SegF}, \emptyset, \phi) = \begin{cases} \{\mathit{SegF}\} \text{ if } \phi \text{ is satisfiable} \\ \emptyset \text{ otherwise} \end{cases}$
- $\gamma(\{M_1, \ldots, M_n\}) = \bigcup_{i=1}^{n} \gamma(M_i)$

The operator $\sqcup$ such that $\gamma(M_1 \sqcup M_2) = \gamma(M_1) \cup \gamma(M_2)$ is defined the evident way.

2.4 Canonical form

Different SMSs can have the same concretization. We show that we can compute a canonical form. In the case of an SMS (G, c, ϕ) where ϕ is satisfiable, the natural choice for a canonical form is the one where the memory graph is as simple as possible. But if ϕ is not satisfiable, there is no natural choice for a canonical form: we arbitrarily choose $(\mathit{SegF}, \emptyset, \phi)$ as a canonical form.

We write $\mathit{deg}(n)$ for the number of input edges of a node n. A node n is called a *core node* if $\mathit{deg}(n) \neq 1$ or $\mathit{var}(n) \neq \emptyset$.

An atomic SMS (G, c, ϕ) is in canonical form iff: the set N_G of nodes of G only contains core nodes; for all counter variable $k \in \mathit{Im}(c)$, the formula $\phi \Rightarrow k \neq 0$ is valid; If ϕ is not satisfiable, then G is equal to *SegF*.

Definition 2.3 An SMS $\{M_1, \ldots, M_n\}$ is in canonical form iff each M_i is in canonical form, and for any two distinct i and j between 1 and n such that M_i is of the form (G_i, c_i, ϕ_i) and M_j is of the form (G_j, c_j, ϕ_j), (G_i, c_i) and (G_j, c_j) are not isomorphic.

An atomic SMS of the form $(\mathit{SegF}, \emptyset, \phi)$ is thus necessarily in canonical form.

Theorem 2.4 *For all SMS M, we can compute a unique SMS M' in canonical form such that $\gamma(M) = \gamma(M')$.*

Sketch of the proof.

For an atomic SMS M, we first build an equivalent SMS $M' = (G, c, \phi)$ with only core nodes. This is done easily by aggregating intermediate nodes. Then for each edge e_i of G, labeled by the counter c_i, we consider two cases, either $c_i \geq 0$ (the edge is kept and the formula becomes $\phi \wedge c_i > 0$) or $c_i = 0$ and the edge is removed. We construct the two SMSs $M'_{i,1} = (G, c, \phi \wedge c_i > 0)$ and $M'_{i,2}$ obtained from M' removing edge c_i, merging its source node and destination node, and aggregating intermediate nodes. We iterate the operation on $M'_{i,1}$ and $M'_{i,2}$ for another edge j, and then iterate on the resulting SMSs for another edge k and so on for all edges . Let $M_{can} = \{M'_1, M'_2, \ldots, M'_{2^n-1}, M'_{2^n}\}$ the SMSs obtained at the last iteration. If in M_{can} there exists atomic SMS (G, c, ϕ_1) and (G, c, ϕ_2), they are removed from M_{can} and replaced by $(G, c, \phi_1 \vee \phi_2)$. The final SMS M_{can} obtained is canonical and $\gamma(M_{can}) = \gamma(M)$. □

About complexity. The complexity is exponential in both time and space. Crucial points are the number of intermediate SMS created and checking graph isomorphism. However the scope of this paper was not to propose efficient algorithm, and the above procedure does not attempt to be efficient. We outline some ideas that we are exploring to obtain faster procedures. If we abstract dead nodes (nodes which are not reachable from labeled nodes), we get two interesting properties. First graph isomorphism can be check easily (in linear time), second the number of possible memory graphs becomes finite and depends of the number of variables (which is a constant). These results seem promising but are still in progress.

Figure 1(a) shows an example of a non canonical SMS. Its counter k can be equal to 0. Figure 1(b) shows the canonical form for this SMS. The formula ϕ_i are written closed to the memory graphs. If there is no formula, then $\phi = True$. We can see that the two labeled memory graphs of (b) are not isomorphic and their counters are different from 0.

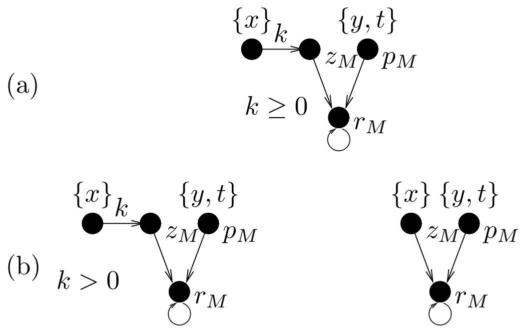

Fig. 1. an SMS (a) and its canonical form (b)

Corollary 2.5 *SMSs are closed under intersection.*

2.5 Predicates

Order.

We want to use SMS for building reachability sets using fixpoint computation. Such a procedure requires to test the inclusion of sets computed so far. So we need to define a relation on SMSs which is consistent with inclusion of concretizations. We first define the relation $\sqsubseteq$ inductively on SMSs in canonical form by (M_1 and M_2 are SMSs in canonical form):

- $(G, c, \phi) \sqsubseteq (G', c', \phi')$ iff: G and G' are different of *SegF*; (G, c) and (G', c') are isomorphic; and the formula $\phi \Rightarrow \phi'$ is valid.
- $(SegF, \emptyset, \phi) \sqsubseteq (SegF, \emptyset, \phi')$ iff ϕ is not satisfiable or ϕ' is satisfiable.
- $M_1 \sqcup M_2 \sqsubseteq M$ iff $M_1 \sqsubseteq M$ and $M_2 \sqsubseteq M$.
- $M \sqsubseteq M_1 \sqcup M_2$ iff $M \sqsubseteq M_1$ or $M \sqsubseteq M_2$.

We then extend this relation to any SMSs M_1 and M_2 by $M_1 \sqsubseteq M_2$ iff $M_1' \sqsubseteq M_2'$ where M_1' and M_2' are respectively the canonical forms of M_1 and M_2.

Theorem 2.6 *The relation $\sqsubseteq$ is a decidable order on SMSs. Moreover, let M_1 and M_2 be two SMSs, then $M_1 \sqsubseteq M_2 \Leftrightarrow \gamma(M_1) \subseteq \gamma(M_2)$.*

Corollary 2.7 *Emptiness is decidable for SMSs.*

Memory leak.

We want to check two main properties on systems with pointers: the absence of both segmentation faults and memory leaks. We define both of these notions via SMS's. While it is immediate for segmentation fault (there is a special memory graph for it), we need a few definitions for the memory leak.

A node in a graph is *dead* iff there is no path to this node from a node n such that $var(n) \neq \emptyset$. Intuitively, a dead node will represent a memory cell that is allocated but can neither be accessed nor deallocated by the program.

Definition 2.8 An SMS is a *memory leak* iff the graph of its canonical form contains at least one dead node.

3 Pointer automaton

We define pointer automata which will be the language in which we write the programs to be verified. A pointer automaton is a finite automaton with a fixed finite set X of pointer variables. And each transition is labeled by a guarded action whose syntax is defined by:

$l ::=$	$x \in X$	left pointer expression	$e ::=$	**Null**	pointer expression
$\mid$	$l.s$		$\mid$	l	
$p ::=$	*True*	predicate	$a ::=$	**nop**	action
$\mid$	*IsNull*(e)		$\mid$	l:=**malloc**	
			$\mid$	**free**(e)	
$g ::=$	p	guard	$\mid$	l:=e	
$\mid$	$\neg p$				

We write G (respectively, Σ) for the countably infinite set of guards (respectively, actions).

Definition 3.1 A pointer automaton is a triple $A = (Q, \Delta, q_0)$ where Q is the finite set of control states, $\Delta \subseteq Q \times G \times \Sigma \times Q$ is the finite transition relation, and q_0 is the initial state.

Example 3.2 Fig. 2 presents a function `reverse` written in C, and its corresponding pointer automaton. Fig. 3 gives a graphical representation of this automaton. Here we are just interested in the behaviour *inside the program*, so we do not consider deallocation of local variables when exiting the program. This example is taken from [8].

```
/* list.h */
typedef struct node
{
  struct node* n;
  int data;
}* List;
```

```
/* 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, $IsNull(x)$, **nop**, 7)

Fig. 2. A `C` program which reverses a list and the corresponding pointer automaton

Operational semantics. The operational semantics of pointer automata in term of memory graphs is defined in the obvious way, in terms of adding/removing nodes, edges and labeling variables. We just give two examples:

- `x:=malloc` creates a new node labeled by x (x does not labeled any more the node it was labeling before the instruction) and connected to the special node p (the deallocated memory).
- `free(`l`)` tests if the node n represented by the pointer expression l is either the error node r or the sink node p. In this case, the memory graph returned is $SegF$ (a segmentation fault occurs[2]). Otherwise n and all its connected edges are removed. Nodes connected to n are now connected to p. Variables which were labeling n are now labeling p.

We have also defined an operational semantics in term of SMSs and transition system, coherent with the operational semantics on memory graphs with respect to the concretization γ. However, the semantics is quite involved and is omitted in this paper.

[2] For the second case, there is no specification in Unix. We took the Linux behaviour.

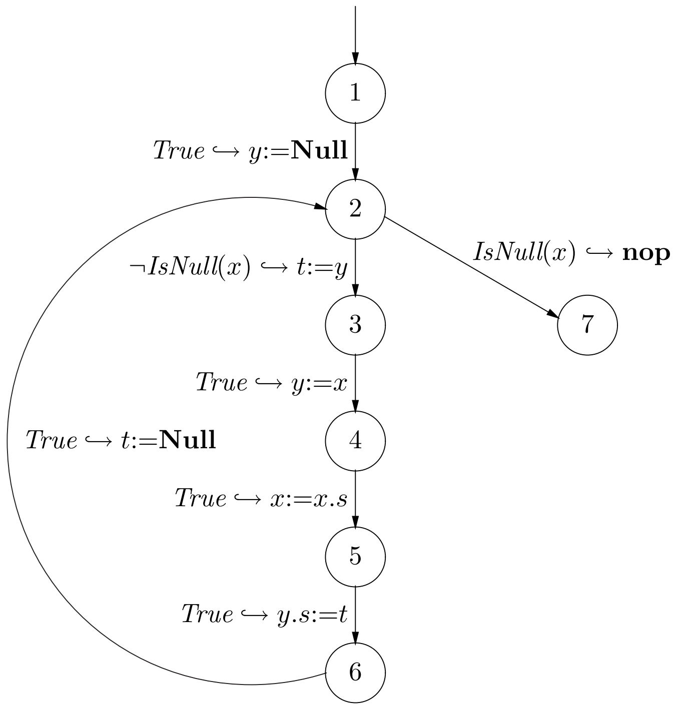

Fig. 3. A graphical representation of the pointer automaton of `reverse`

4 Undecidability results

We formally define the properties to be checked on a pointer automaton and its initial SMS, and show that they are undecidable.

NoSegF For any $q \in Q$, a state of the form $(q, (SegF, \emptyset, \phi))$ where ϕ is satisfiable is not reachable.

NoMemLeak For any $q \in Q$, a state (q, M) where M is a memory leak is not reachable.

Theorem 4.1 *The properties* **NoSegF** *and* **NoMemLeak** *are undecidable for pointer automata with at least 3 pointer variables.*

Sketch of the proof. We encode any 2-counter machine by a pointer automaton with three variables. Intuitively each counter c_i is represented by a list whose length is the value of c_i. The third pointer is used to add elements to the lists. Operations on counters (test to 0, increment, decrement) are simulated by operations on lists (test to null, add an element, remove an element). Here whether 0 is reachable from a given initial set of states is equivalent to the properties **NoSegF** or **NoMemLeak**. This problem is well-known to be undecidable for 2-counter machines. $\square$

5 Perspective

At this stage, we can check invariants (represented by SMSs) on pointer automata by computing one iteration. This can be helpful on some small examples, but for realistic examples, appropriate invariants are difficult to determine. We could try to isolate decidable subclasses. For instance, it is likely that we gain decidability if we restrain ourselves to pointer automata with 1 or 2 variables. We could also try to reduce the language of actions such as forbidding the action **malloc**. But all these subclasses are not realistic enough.

That is why we are now focusing on acceleration techniques which provide semi-algorithms and have proved to be efficient on other models, such as counters, clocks or lossy channels [1,2,4,5,6]. The acceleration of a loop comes down to computing in one step the effect of iterating an arbitrary number of time a loop of the control graph.

At first sight, acceleration is similar to widening in abstract interpretation. The difference is that widening aims at termination and thus sacrifices exactness, while acceleration does not guarantee termination but is exact, in the sense that it computes only reachable states. When the semi-algorithm terminates it returns the exact reachability set of the model.

Let us try to compute the set of reachable states of the program `reverse` (example 3.2) when the initial state is $(1, M_1)$ where M_1 is the SMS given in Figure 1. We can first try to compute the set of reachable states by iteration. Let us focus on the control state 7. The reachable states at the first, second and third iteration are respectively depicted in canonical form on Figure 4(1), (2) and (3). At each iteration the SMS strictly increases. We thus cannot compute the set of reachable states simply by iteration. However we can easily see that there is a pattern which is repeated at each iteration. In fact, in this example, the infinite sequence of iterations converges. This limit is depicted in Figure 4(∞).

We expect some accelerations techniques may amount to computing the limit of such infinite sequence. Up to now, there is no general framework for acceleration, and existing acceleration techniques strongly depend on symbolic representations. We will try to adapt existing techniques and graph rewriting methods, and then apply them on concrete examples.

References

[1] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: a Tool for Reachability analysis of Complex Systems. In *Proc. of CAV'2001*, LNCS 2102, pages 368–372. Springer, 2001.

[2] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Proc. of CAV'2003*, LNCS 2725, pages 118–121. Springer, 2003.

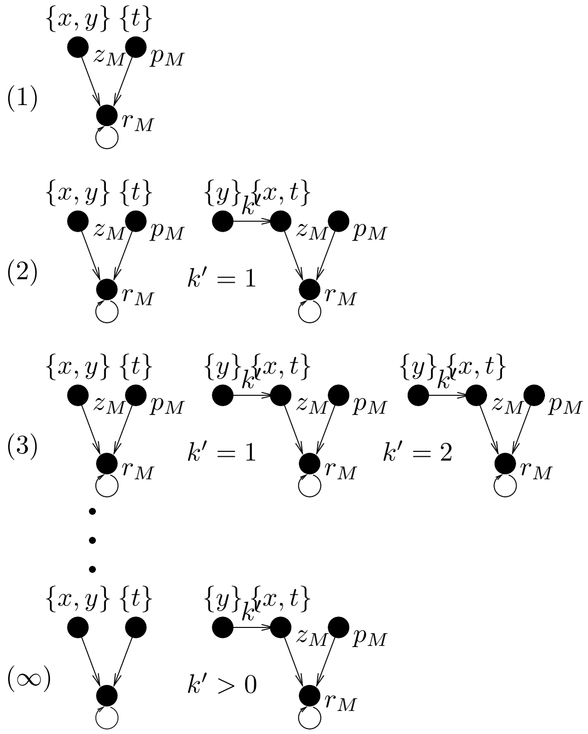

Fig. 4. The first (1), second (2), third (3) iteration, and the limit (∞) of an infinite sequence of SMSs

[3] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[4] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1998.

[5] A. Bouajjani. Languages, rewriting systems, and verification of infinite-state systems. In *Proc. of ICALP'01*, LNCS 2076, pages 24–39. Springer, 2001.

[6] A. Finkel and J. Leroux. How To Compose Presburger-Accelerations: Applications to Broadcast Protocols. In *Proc. of FSTTCS'02*, LNCS 2556, pages 145–156. Springer, 2002.

[7] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. of PASTE'01*, pages 54–61. ACM Press, 2001. Invited talk.

[8] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analysis. In *Proc. of SAS'00*, pages 280–201.

[9] 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.

[10] A. Møller and M. Schwartzbach. The pointer assertion logic engine. In *Proc. of PLDI-01*, volume 36.5 of *ACM SIGPLAN Notices*, pages 221–231. ACM Press, 2001.