Towards model-checking programs with lists[⋆]

Alain Finkel[1], Étienne Lozes[1], and Arnaud Sangnier[1,2]

[1] LSV, ENS Cachan & CNRS UMR 8643
61, av. Pdt Wilson 94235 Cachan Cedex, France
{finkel|lozes|sangnier}@lsv.ens-cachan.fr
[2] EDF R&D

Abstract. We aim at checking safety and temporal properties over models representing the behavior of programs manipulating dynamic singly-linked lists. The properties we consider not only allow to perform a classical shape analysis, but we also want to check quantitative aspect on the manipulated memory heap. We first explain how a translation of programs into counter systems can be used to check safety problems and temporal properties. We then study the decidability of these two problems considering some restricted classes of programs, namely flat programs without destructive update. We obtain the following results : (1) the model-checking problem is decidable if the considered program works over acyclic lists (2) the safety problem is decidable for programs without alias test. We finally explain the limit of our decidability results, showing that relaxing one of the hypothesis leads to undecidability results.

1 Introduction

Context. The model checking techniques of infinite-state systems are now an active research area. These techniques have been successfully applied to a lot of infinite-state transition systems like counter systems, lossy channel systems, pushdown automata, timed automata, hybrid automata, and rewriting systems. For some of these models, there exist today both an impressive set of theoretical results and efficient automatic tools for verifying safety properties.
Among the different models, counter systems enjoy a central position for both theoretical results and maturity of tools like `FAST` [3], `ASPIC` [19], `LASH` [20] and `TREX` [1]. Moreover, counter systems, defined by Presburger relations, allow to model and to verify quantitative properties over various applications such as the TTP protocol [2] and broadcast protocols [16]. Different acceleration methods for computing the reachability relation of such counter systems have been developed . In particular, an acceleration method (recalled in [17]) has been completely implemented in the verification tool `FAST` [3].
In this work, we study a peculiar class of infinite state systems, called pointer systems, which allows to model the behavior of programs working over singly-linked lists. For pointer systems, qualitative properties (absence of memory violation like invalid pointer dereferencing, memory leak) are difficult to analyze, and may drastically rely on

⋆ This work has been partially supported by contract 4300038040 between EDF R&D/LSV and by the AVERILES project.

quantitative properties (lists of equal length, finitely increasing memory allocation,...). Some recent works illustrate this relation between quantitative and qualitative aspects in pointer systems for the purpose of verifying termination [7] or safety [11].

Motivations. Our attempt in this paper is to demonstrate that this relation between quantitative and qualitative properties is central in many aspects of pointer systems. Consider for example the following program that allocates a copy of a list, and then disposes both lists. In this program, the variables x,y,t and u are pointers to list elements.

```
void copy-and-delete(List x){
 List y,t,u;
 y = x; t=NULL; u=t;
 while (y!=NULL){
   y=y->next;
   t=malloc();
   t->next=u;
   u=t;}
 y=x;
while (y!=NULL){
  y=y->next;
  u=u->next;
  free(x); x=y;
  free(t);t=u;
}
```

Such a program cannot be analyzed correctly by tools based on purely qualitative shape analysis, whereas combining shape analysis and quantitative analysis it can be automatically established that this program is safe reminding that both lists are of equal length after the first loop and that the initial list is acyclic. It is true that this example program seems quite artificial, but it has to be seen as an abstraction of a more complex program in which all the operations which had no effect on the shape of the data structure have been removed. In fact, in this work we consider programs which can only manipulate singly-linked lists and which have to be considered as an abstraction of real programs.

We believe that situations such as the one we present with the above example are not so rare in programs manipulating pointers, and a quantitative shape analysis could be worth to be considered in several practical cases. More generally, this is certainly a promising perspective to consider systems that combine pointers and counters operations and design an analysis that relates both aspects of the computation. Counters could model parameters of list-scanning algorithms (for instance, a procedure that returns the nth element of a list), but also concurrency aspects like semaphores, thread identifier, etc. Quantitative shape analysis could be well suited for model-checking temporal properties relying on the algorithms already proposed for counter systems, such as in [13]. Previous attempts of model-checking temporal properties on pointer systems have been mostly based on either over-approximating, or non terminating algorithms for which completeness is usually poorly studied. Exact and complete algorithms for rich classes of programs, though less efficient than others in general, can play a crucial role in practice.

In broad lines, we would like to reuse the principles of standard shape analysis to define a quantitative shape analysis, and specifically rely on the tool FAST for the quantitative aspects. FAST implements a loops acceleration that succeeds in computing the reachability set of any flat counter system; the same acceleration can be exploited to model-check CTL* properties on these systems [13]. By flat, we mean that the control flow graph does not contain nested loops. Even if the class of such systems seems to be quite restricted with respect to usual programs, several programs transformations can be tried to flatten the system and cover a much richer class of programs. A flattening procedure is implemented in FAST, and many classes of programs on which this procedure succeeds have been identified [21]. Furthermore many case studies have been verified with this approach.

Our research program, in this paper, was to try to follow the same tracks as for counter systems in the framework of pointer systems. Thus, we and others considered the class of flat pointer systems and tried to define an acceleration for them. Unfortunately, it has been shown that for flat pointer systems, and even for flat pointer systems without destructive update, the problem of reachability of a control state is undecidable. This result is somehow unsettling and we wanted to better understand for which classes of pointer systems some CTL* properties (including reachability and safety) are decidable or not.

Contribution. Our main contribution is to investigate the possibility of a quantitative model-checker for singly-linked lists manipulating programs. We define a CTL* logic which is able to express quantitative properties of the memory managed by pointer systems. For this temporal logic, we show that the model-checking problem reduces to the one for counter systems developed in [13], provided an adequate translation from pointers to counters is given. This result is important for us, since it serves as a foundation for a two-steps analysis of pointer systems that consists in translating them into counter systems and then, with the help of FAST, to verify safety properties. It remains to provide a good translation of pointer systems into counter systems. We obtain three categories of results concerning the translations:

First, we show that, from the experimental point of view, the translation defined in [5] allows us to verify all well-known singly-linked lists case studies, and also some new ones that are not immediately verifiable by the other tools and methodologies. Most of the case studies yield flat counter systems for which we know in advance that FAST will terminate.

Second, we propose a new translation of pointer systems into counter systems and we characterize several classes of pointer systems without destructive update for which our analysis terminates. The main feature of this new translation is that it preserves flatness of the systems (unlike the first translation). Using this new translation, we prove the decidability of the CTL* model-checking for flat pointer systems without destructive update and with an acyclic initial configuration and the decidability of safety problems for flat pointer systems without destructive update and without alias test. In [11], the authors prove that some safety problems were decidable when considering flat programs without destructive update and with an initial configuration containing at most one cyclic list. We hence extend here these results to the model-checking when the initial configuration is acyclic and we propose a new class of flat programs without destructive update for which the safety problem is decidable.

Third, we explore the limits of our analysis, and of the decidability of CTL* model-checking for classes of flat pointer systems. We show that we cannot extend our analysis of flat pointer systems without destructive update and alias test if cyclic initial configurations are allowed. Conversely, we show that the safety problem becomes undecidable for flat pointer systems that keep their memory configurations acyclic, but can perform destructive update. This last point answers an open problem asked in [11].

Related work. Many tools and techniques to check safety properties on programs manipulating pointers have been developed recently. PALE [26] verifies safety properties on programs annotated with loop invariants. TVLA [22] is another tool based on abstract interpretation, where the user has the possibility to refine the abstraction providing adequate predicates. In [27], the framework of predicate abstraction is used to manipulate boolean formulae representing the heap. In [15], the authors present a shape analysis method based on separation logics formulae to analyze programs manipulating singly-linked lists. Their method always terminates but might yield false alarms due to the over-approximation brought by the abstraction. Other methods have been proposed which use already existing model-checking techniques. For instance, in [9], the authors verify safety properties on programs manipulating singly-linked lists using abstract regular model-checking. They have extended their work to programs with more complex data structures [10]. In [8], the authors also propose a translation towards counter systems very similar to the one of [5]. In [24], the authors propose to combine shape analysis and arithmetic analysis using the same kind of techniques. None of these considers temporal model-checking or state a completeness result for these analysis.
Some efforts have already been made to introduce temporal logic for pointer verifications as the Evolution Temporal Logic [29], which used techniques similar to the one presented in TVLA, or the Navigation Temporal Logic presented in [14]. Recently, in [12], the authors have introduced a temporal logic based on separation logic. But, to our knowledge, these different logics do not allow to express quantitative properties of the memory heaps.

2 Preliminaries

In this section, we collect some useful notions about counter and pointer systems. We assume a set C of counter variables, and a set V of pointer variables.

2.1 Counter systems

We recall that *Presburger arithmetic* is the first order theory of the structure $\langle \mathbb{N}, +, = \rangle$. Given a Presburger formula ϕ with free variables belonging to C and $\mathbf{a} \in \mathbb{N}^C$, we write $\mathbf{a} \models \phi$ if ϕ is true for the valuation $\mathbf{a}$. We will denote by $[\![\phi]\!]$ the set described by the formula ϕ. A *Presburger-linear function* f is a partial function which can be represented by a tuple $(A, \mathbf{b}, \phi)$ where A is a square matrix in $\mathbb{N}^{C \times C}$, $\mathbf{b} \in \mathbb{Z}^C$ and ϕ is a Presburger formula such that $f(\mathbf{a}) = A.\mathbf{a} + \mathbf{b}$ for every $\mathbf{a} \models \phi$. We denote by Σ_C the set of such functions.

Definition 1 (Counter system). *A* counter system *is a graph whose edges are labeled with Presburger linear functions, that is a tuple $CS = \langle Q, E \rangle$ where Q is a finite set of control states and $E \subseteq Q \times \Sigma_C \times Q$ is a finite set of transitions.*

To a counter system $CS = \langle Q, E \rangle$, we associate the transition system $TS(CS) = \langle Q \times \mathbb{N}^C, \rightarrow \rangle$ defined by $(q, \mathbf{a}) \rightarrow (q', \mathbf{a}')$ if there is a transition (q, f, q') in E with $f = (A, \mathbf{b}, \phi)$ such that $\mathbf{a} \models \phi$ and $\mathbf{a}' = f(\mathbf{a})$. We see here that when a transition of the counter system is labelled with a Presburger-linear function $(A, \mathbf{b}, \phi)$ the Presburger formula ϕ plays the role of a guard on the transition and the action of the transition is represented by the translation which associates to each $\mathbf{a} \in \mathbb{N}^C$ the vector $A.\mathbf{a} + \mathbf{b}$.

A *simple cycle* in a graph $G = \langle Q, E \rangle$ is a closed path (where the initial and final vertices coincide) with no repeated edge. G is said to be *flat* if every $q \in Q$ belongs to at most one cycle. Let $CS = \langle Q, E \rangle$ be a counter system. We define the monoid of CS, denoted by $< CS >$, as the multiplicative monoid generated by the matrices present in the labels of CS. More formally, $< CS >= \bigcup_{i \geq 0} \{A_1.A_2.\ldots.A_i \mid \forall j \in \{1, \ldots, i\}$ there exists $(q, (A_j, \mathbf{b}, \phi), q') \in E\}$. A counter system CS is said to have *the finite monoid property* if the multiplicative monoid $< CS >$ is finite. Note that since a counter system has a finite number of control states and of transitions, one can decide whether a counter system is flat or not. This also holds for the finite monoid property, in fact using a result of [25], one can prove that the problem of knowing whether the monoid of a counter system is finite is decidable .

For the following theorem, we use the fact that the control states of a counter system $CS = \langle Q, E \rangle$ can be encoded into positive integers (ie $Q \subseteq \mathbb{N}$) and then the set of configurations is represented by $\mathbb{N}^{|C|+1}$.

Theorem 1. *[17] Let CS be a flat counter system $\langle Q, E \rangle$ with the finite monoid property and $TS(CS) = \langle \mathbb{N}^{|C|+1}, \rightarrow \rangle$ its associated transition system. Then the relation $\rightarrow^*$ is effectively Presburger definable.*

Note that as mentioned in [17], this last result extends and completes previous results on acceleration techniques for counter systems presented in [28].

Let us recall the syntax of the temporal logic for counter systems, called FOCTL*(Pr) in [13] :

$$\Phi \ ::= q \mid \psi \mid \exists y.\Phi \mid \neg \Phi \mid \Phi \wedge \Phi \mid \mathsf{X}\Phi \mid \Phi \mathsf{U} \Phi \mid \mathsf{A}\Phi$$

where q is a control state, y is a variable of a countable set VAR and ψ is a Presburger formula over $C \cup$ VAR.
We now give the semantics of this temporal logic. We consider a counter system $CS = \langle Q, E \rangle$ and its associated transition system $TS(CS) = \langle Q \times \mathbb{N}^C, \rightarrow \rangle$. Let π be a configuration path in $TS(CS)$. For an integer i, we denote by $\pi(i) \in Q \times \mathbb{N}^C$ the i-th configuration of π, $\pi_{\leq i}$ the initial part of π up to position i and $|\pi|$ the length of π. Let ρ be a variable valuation, that is a partial map from VAR to $\mathbb{N}$. For $i \in \mathbb{N}$ and a formula Φ of FOCTL*(Pr), the satisfaction relation $\models$ is inductively defined at position i of a configuration path π as follows:

- $\pi, i \models_\rho q$ iff $\pi(i) = (q, \mathbf{a})$ for some $\mathbf{a} \in \mathbb{N}^C$;
- $\pi, i \models_\rho \psi$ iff $\pi(i) = (q, \mathbf{a})$ with $q \in Q$ and $(\mathbf{a}, \rho) \models \psi$ in Presburger arithmetic;

- $\pi, i \models_\rho \exists y.\Phi$ iff there is $m \in \mathbb{N}$ such that $\pi, i \models_{\rho[y \mapsto m]} \Phi$ where $\rho[y \mapsto m]$ denotes the variable valuation equal to ρ except that the variable y is mapped to the integer value m;
- $\pi, i \models_\rho \neg\Phi$ iff $\pi, i \not\models \Phi$;
- $\pi, i \models_\rho \Phi \wedge \Phi'$ iff $\pi, i \models \Phi$ and $\pi, i \models_\rho \Phi'$;
- $\pi, i \models_\rho \mathsf{X}\Phi$ iff $i < |\pi|$ and $\pi, i+1 \models \Phi$;
- $\pi, i \models_\rho \Phi\mathsf{U}\Phi'$ iff $\exists j$ such that $i \leq j < |\pi|$ and $\pi, j \models \Phi'$, and for all k such that $i \leq k < j$, $\pi, k \models \Phi$;
- $\pi, i \models_\rho \mathsf{A}\Phi$ iff for all configuration path π' such that $\pi_{\leq i} = \pi'_{\leq i}$, we have $\pi', i \models_\rho \Phi$.

We denote by $CS \models \Phi$ the fact that all the configurations paths π in $TS(CS)$ verify $\pi, 0 \models \Phi$. The next result shows that the theorem 1 can be extended to temporal properties, in fact :

Theorem 2. *[13] For a flat counter system CS with the finite monoid property, and a FOCTL***(Pr) formula Φ, it is decidable whether $CS \models \Phi$.*

2.2 Pointer systems

We now define the model of pointer systems that will be the core of our study. We use pointer systems to represent the behavior of programs manipulating singly-linked lists. The main idea of this model consists in representing the memory heap as a graph in which each node has at most one successor. In the sequel, we use the symbol $\perp$ to express that a function is undefined.

Definition 2 (Memory graph). *A* memory graph *is a labeled graph that can be represented by a tuple $MG = (N, succ, loc)$ such that :*

- *N is a finite set of nodes such that $\{\mathtt{null}, \perp\} \cap N = \emptyset$;*
- *$succ$ is a function from N to $N \cup \{\mathtt{null}, \perp\}$ called the successor function;*
- *loc is a function from V to $N \cup \{\mathtt{null}, \perp\}$ which associates a node with each pointer variable;*
- *for all nodes $n \in N$, there is $v \in V$ and $i \in \mathbb{N}$ such that $n = succ^i(loc(v))$.*

Note that the last condition intuitively expresses that the memory graph represents a heap without memory leak, i.e. all nodes are reachable in the graph from a node pointed to by a variable. We impose that memory graphs do not contain memory leaks because anyway the nodes not reachable from a variable in a memory graph that could remain after executing an instruction would not have any incidence on the behavior of the program. In the sequel, we will see that if an action performed a memory leak, we consider it as a fault. Remark that we could also have a semantic with a garbage collector and in this case we would delete in the graphs the nodes that are not reachable by a pointer variable. We denote by $\mathcal{MG}_V$ ($\mathcal{MG}$ for short) the set of all memory graphs over V. We will say that two memory graphs are equal if there exists an isomorphism between their underlying graphs which respects the positions of the variables. A *cyclic list* in a memory graph is a simple cycle in the underlying graph and a memory graph is said to be *acyclic* if it does not contain any cyclic list.

We define *guarded pointer actions* as pairs denoted (g, a) where guards and actions are defined by the following grammar :

$$\begin{aligned} g &::= True \mid Isnull(x) \mid x = y \mid \neg g \mid g \wedge g \\ a &::= x{:=}e \mid x.succ{:=}e \mid x{:=}\textbf{malloc} \mid \textbf{free}(x) \mid \textbf{skip} \\ e &::= \text{NULL} \mid x \mid x.succ \end{aligned}$$

where x, y are pointer variables belonging to V and $x.succ$ represents the successor node of the cell pointed to by x. We denote by $\mathcal{G}$ the set of pointer guards, $\mathcal{A}$ the set of pointer actions and $\Sigma_P = \mathcal{G} \times \mathcal{A}$. For a memory graph $MG \in \mathcal{MG}$ and a pointer guard $g \in \mathcal{G}$, we denote by $MG \models g$ the fact that MG satisfies g. For a pointer action $a \in \mathcal{A}$, we define the partial function $[\![a]\!]_P : \mathcal{MG} \rightarrow \mathcal{MG}$ which associates to a memory graph MG the memory graph $[\![a]\!]_P(MG)$ obtained after executing the action a over MG. The function $[\![a]\!]_P$ is defined partially because there are some situations in which the action a realizes what we call a fault on a memory graph MG and in this case $[\![a]\!]_P(MG)$ is not defined. In our approach, we consider as a fault both memory violation and memory leak. Intuitively, a memory violation occurs when an action tries to move a pointer variable to the successor of the `null` node or to the successor of an undefined node; whereas a memory leak occurs when moving a variable leads to a graph where there exists a node which is not reachable from a node labeled with a variable. We also believe it would not change our results to consider garbage-collected programs (ie programs for which a memory leak is not considered as an error).

Definition 3 (Pointer system). *A* pointer system *is a graph whose edges are labeled with pairs of pointer guards and actions, that is a tuple $PS = \langle Q, E \rangle$ where Q is a finite set of control states and $E \subseteq Q \times \Sigma_P \times Q$.*

We then associate a transition system $TS(PS) = \langle Q \times \mathcal{MG}, \rightarrow \rangle$ with a pointer system $PS = \langle Q, E \rangle$. Its transition relation is defined by : $(q, MG) \rightarrow (q', MG')$ if there is a transition $q \xrightarrow{(g,a)} q'$ in E such that $MG \models g$ and $MG' = [\![a]\!]_P(MG)$. A configuration of a pointer system $PS = \langle Q, E \rangle$ is a pair (q, MG); for a set of configurations $\mathcal{C}$, we write $\texttt{Reach}(PS, \mathcal{C})$ to denote the set of configurations reachable in $TS(PS)$ from some configuration in $\mathcal{C}$.

2.3 Representing infinite sets of memory graphs

We introduce now a symbolic representation of memory graphs. Intuitively, a memory shape is a memory graph where the intermediate nodes of an unshared list segments are skipped and replaced by a counter recording the length of this list segment.

Definition 4 (Memory shape). *[5,6] A* memory shape *is a tuple $MS = (N, succ, loc, l)$ such that :*

- $(N, succ, loc)$ *is a memory graph verifying:*
  - *for all nodes $n \in N$, either $loc^{-1}(n) \neq \emptyset$, or $|succ^{-1}(\{n\})| \geq 2$;*

– *$l : N \rightarrow C$ is an injective function which associates with each node a counter variable.*

We denote by $\mathcal{MS}$ the set of memory shapes. We will write C_{MS} for the set of counters appearing in a memory shape MS (i.e. the image of the function l). To a pair $(MS, \mathbf{a}) \in \mathcal{MS} \times \mathbb{N}^C$ (such that the values of the counters in C_{MS} are strictly positive), we associate the memory graph $MS(\mathbf{a})$ obtained from the memory graph underlying MS by inserting intermediate nodes on list segments in order to have a list length equal to the value of the counter. As said in [5,6], for a fixed set V there is a finite number of memory shapes and for each memory graph MG there exists a memory shape MS and a valuation $\mathbf{a}$ such that $MG = MS(\mathbf{a})$. An example is given in figure 1.

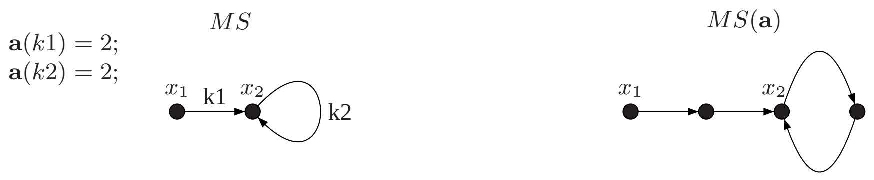


Fig. 1. A memory graph associated with a memory shape and a valuation

To represent infinite sets of memory graphs, we define what we call the *symbolic memory shapes*. A symbolic memory shape is a pair (MS, ψ) where MS is a memory shape and ψ a Presburger formula over C_{MS}. The interpretation of a symbolic memory shape is given by $[\![(MS, \psi)]\!] = \{MS(\mathbf{a}) \mid \mathbf{a} \models \psi\}$.

Definition 5 (Symbolic memory state). *A* symbolic memory state SMS *is a finite set* $\{(MS_1, \psi_1), ..., (MS_r, \psi_r)\}$ *of symbolic memory shapes* (MS_i, ψ_i)*.*

For a symbolic memory state $SMS = \{(MS_1, \psi_1), ..., (MS_r, \psi_r)\}$ the concrete interpretation is given by $[\![SMS]\!] = \bigcup_{i \in \{1,...,r\}} [\![(MS_i, \psi_i)]\!]$. We denote by $\mathcal{SMS}$ the set of symbolic memory states. In [6], the authors have shown that symbolic memory states enjoy good properties, in particular that it is possible to define complement and intersection operators for this representation.

3 Model checking issues

In this section, we define the safety and temporal properties we consider in this work. We first formally define the model-checking problems. Then, we recall a method, presented in [5], to analyze pointer systems translating them into a bisimilar counter system.

3.1 Model-checking programs with pointers

We define now the notions of safety and model-checking when the considered model is a pointer system.

A symbolic configuration of a pointer system $PS = \langle Q, E \rangle$ is a finite set of pairs (q, SMS), for $q \in Q$ and SMS a symbolic memory state.

Definition 6 (Safety in pointer systems). *The safety problem for a pointer system is defined by :*

– ***Input :*** *A pointer system $PS = \langle Q, E \rangle$, an initial symbolic configuration* $\mathtt{INIT}$ *and a "bad" symbolic configuration* BAD*;*
– ***Output :*** $\mathtt{Reach}(PS, [\![\mathtt{INIT}]\!]) \cap [\![\mathsf{BAD}]\!] \overset{?}{=} \emptyset$.

Note that the problem of deciding whether a given pointer system may reach a given control state, may performs a memory violation, or a memory leak, reduce to this generic safety problem.
We now consider a temporal logic for pointer systems based on the quantitative shape logic of the previous section :

$$\Phi \ ::= \ q \mid SMS \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{X}\Phi \mid \Phi\mathsf{U}\Phi \mid \mathsf{A}\Phi$$

where q is a control state and SMS is a symbolic memory state. We denote by CTL^*_{mem} this logic. We give the semantics of this logic, defined by a relation $\pi, i \models \Phi$ between traces π of a pointer system and a formula Φ of CTL^*_{mem}. We consider a pointer system $PS = \langle Q, E \rangle$ and its associated transition system $TS(PS) = \langle Q \times \mathcal{MG}, \rightarrow \rangle$. Let π be a configuration path in $TS(PS)$. For an integer i, we denote by $\pi(i) \in Q \times \mathcal{MG}$ the i-th configuration of π, $\pi_{\leq i}$ the initial part of π up to position i and $|\pi|$ the length of π. For $i \in \mathbb{N}$ and a formula Φ of CTL^*_{mem}, the satisfaction relation $\models$ is inductively defined at position i of a configuration path π as follows:

– $\pi, i \models q$ iff $\pi(i) = (q, MG)$ for some $MG \in \mathcal{MG}$;
– $\pi, i \models SMS$ iff $\pi(i) = (q, MG)$ with $q \in Q$ and $MG \in [\![SMS]\!]$;
– $\pi, i \models \neg\Phi$ iff $\pi, i \not\models \Phi$;
– $\pi, i \models \Phi \wedge \Phi'$ iff $\pi, i \models \Phi$ and $\pi, i \models \Phi'$;
– $\pi, i \models \mathsf{X}\Phi$ iff $i < |\pi|$ and $\pi, i+1 \models \Phi$;
– $\pi, i \models \Phi\mathsf{U}\Phi'$ iff $\exists j$ such that $i \leq j < |\pi|$ and $\pi, j \models \Phi'$, and for all k such that $i \leq k < j, \pi, k \models \Phi$;
– $\pi, i \models \mathsf{A}\Phi$ iff for all configuration path π' such that $\pi_{\leq i} = \pi'_{\leq i}$, we have $\pi', i \models \Phi$.

We are then interested in solving the model-checking problem of formulae of CTL^*_{mem} for pointer systems. We define here this problem.

Definition 7 (Model-checking). *The model-checking problem for pointer systems is defined by :*

– ***Input :*** *A pointer system $PS = \langle Q, E \rangle$, an initial symbolic configuration* $\mathtt{INIT}$ *and a formula Φ of CTL^*_{mem};*
– ***Output :*** *Do we have $\pi, 0 \models \Phi$ for all traces π of $TS(PS)$ such that $\pi(0) \in [\![\mathtt{INIT}]\!]$?*

3.2 From pointer systems to counter systems

Let $PS = \langle Q_P, E_P \rangle$ be a pointer system. In [5], we give an effective algorithm to build a counter system $CS(PS)$ which is bisimilar to PS. Before to present this translation, let us recall the definition of a bisimulation :

Definition 8 (Bisimulation). *Given two transition systems $TS_1 = (S_1, \rightarrow_1)$ and $TS_2 = (S_2, \rightarrow_2)$, a relation $R \subseteq S_1 \times S_2$ is a bisimulation if and only if, for all $(s_1, s_2) \in R$:*

1. *If $\exists s_1' \in S_1$ such that $s_1 \rightarrow_1 s_1'$ then $\exists s_2' \in S_2$ such that $s_2 \rightarrow_2 s_2'$ and $(s_1', s_2') \in R$;*
2. *If $\exists s_2' \in S_2$ such that $s_2 \rightarrow_2 s_2'$ then $\exists s_1' \in S_1$ such that $s_1 \rightarrow_1 s_1'$ and $(s_1', s_2') \in R$.*

The translation presented in [5] used the memory shapes and is based on the following principle : given a memory shape $MS \in \mathcal{MS}$ and a pointer action $a \in \mathcal{A}$, it is possible to define a set $POST(a, MS)$ of pairs $((A, \mathbf{b}, \phi), MS')$ where $(A, \mathbf{b}, \phi)$ represents a Presburger-linear function and MS' a memory shape such that, for all $((A, \mathbf{b}, \phi), MS') \in POST(a, MS)$, and for all $\mathbf{a}, \mathbf{a}' \in \mathbb{N}^C$, we have :

$$MS'(\mathbf{a}') = [\![a]\!]_p(MS(\mathbf{a})) \text{ if and only if } \mathbf{a} \models \phi \text{ and } \mathbf{a}' = A.\mathbf{a} + \mathbf{b}$$

The counter system $CS(PS)$ is then equal to $\langle Q_C, E_C \rangle$ where :

– $Q_C = Q_P \times \mathcal{MS}$,
– E_C is the transition relation defined as follows $((q, MS), (A, \mathbf{b}, \phi), (q', MS')) \in E_C$ if and only if there exists a transition $(q, (g, a), q') \in E_P$ such that $MS \models g$ and $((A, \mathbf{b}, \phi), MS') \in POST(a, MS)$.

In this definition, we say that a memory shape $MS = (N, succ, loc, l)$ satisfies a guard g if its underlying memory graph $(N, succ, loc)$ satisfies g. Note that since the sets $\mathcal{MS}$ and Q_P are finite, the counter system $CS(PS)$ can effectively be built. Furthermore, using the property of the function $POST$, we deduce that the relation :

$$B = \{((q, MG), ((q', MS), \mathbf{a})) \in (Q_P \times \mathcal{MG}) \times (Q_C \times \mathbb{N}^C) \mid q = q' \wedge MG = MS(\mathbf{a})\}$$

is a bisimulation between the transition systems of PS and of $CS(PS)$. We can hence use the counter system $CS(PS)$ to analyze the pointer system PS.

The figure 2 gives an example of a connected component of the counter system $CS(PS)$ obtained from the pointer system $PS = \langle \{1\}, \{(1, x_1.succ = x_1, 1)\} \rangle$ with $V = \{x_1, x_2\}$. We see with this example that this translation does not preserve the flatness of systems. Table 1 gives a list of programs working over acyclic initial configurations (except `parse-cyclic-acyclic`) which we have translated into counter systems and successfully analyzed. Some of these programs are described in the appendix A. Most of them are classical programs, except the program `copy-and-delete` presented in the introduction, the program `split` which divides a single list in two lists and is safe only if the input list has an even length, and the program `parse-cyclic-acyclic` which parses a cyclic and an acyclic list in the same time. We remark that in most of

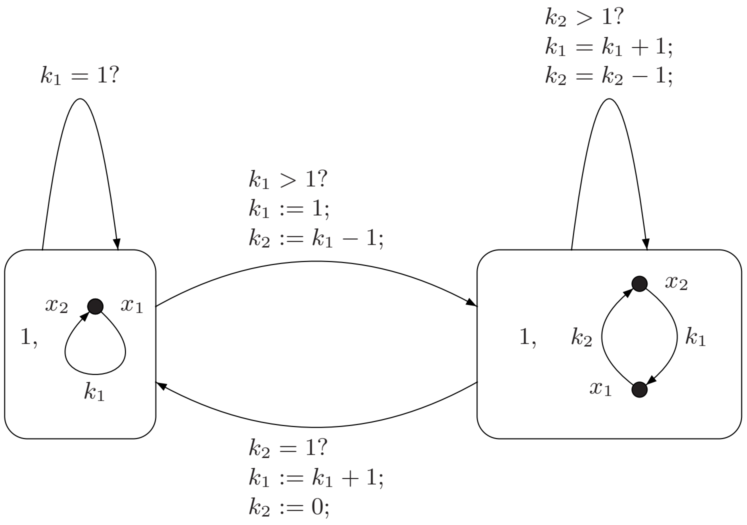


Fig. 2. An example of a counter system $CS(PS)$ obtained from a pointer system PS

the cases the corresponding counter system is flat (and has always, by definition of the translation, the finite monoid property) which corresponds to the hypothesis of the theorems 1 and 2. When the system is not flat, it can still be analyzed sometimes, as for the `merge` program, using a flattening procedure implemented in the tool `FAST`. But in other cases it might not be fully verified, as for the program `parse-cyclic-acyclic`.

Program	**Is PS flat ?**	**Is CS(PS) flat ?**	**Analyzed with FAST ?**
`reverse`	YES	YES	YES
`delete`	YES	YES	YES
`deleteALL`	YES	YES	YES
`merge`	NO	NO	YES
`copy-and-delete`	YES	YES	YES
`split`	YES	YES	YES
`delete(n)`	YES	YES	YES
`parse-cyclic-acyclic`	YES	NO	NO

Table 1. Examples of programs analyzed by `FAST`

Since the memory shapes appear in the control states of $CS(PS)$, it is possible to translate any temporal logic formula over the symbolic configurations of PS into a temporal logic formula over the configurations of $CS(PS)$. Hence :

Theorem 3. *Let Φ_P be a CTL^*_{mem} formula. Then there effectively exists a formula Φ_C of $FOCTL^*(Pr)$ such that for all pointer systems PS :*

$$PS \models \Phi_P \text{ if and only if } CS(PS) \models \Phi_C$$

Furthermore, the counter system $CS(PS)$ has the finite monoid property. In fact, in [5], we can see that all the matrices labeling the transitions of $CS(PS)$ are composed of columns in which all the elements are equal to 0 except one which is equal to 1. Using the theorem 2 and the previous result, we hence have the following result.

Corollary 1. *Let PS be a pointer system such that $CS(PS)$ is flat. Then the model-checking is decidable for PS.*

4 Decidability results for programs without destructive update

After having seen a general method to analyze pointer systems translating them into a counter system, we aim in this section at finding some classes of pointer systems for which the safety and the model-checking problems are decidable. We know that these problems are undecidable for pointer systems in general because it is easy to simulate a Minsky machine with a pointer system. Since there is no obvious notion equivalent to finite monoid property for pointer systems, and since, as we will see, flatness is in general not sufficient to decide reachability properties, we define other restrictions on pointer systems. We will say that a pointer system $PS = \langle Q, E \rangle$ is :

- *without destructive update* if the actions in E are all of the form $x := e$ with $x \in V$;
- *without alias test* if the guards in E do not contain any test like $x = y$ with $x, y \in V$.

In [11], the authors have studied whether flat programs without destructive update, working on a given special shape, could fail to satisfy some assert intructions inserted in the code. This problem reduces to a particular case of safety problem, which is the reachability of a control state. They proved, that this problem is undecidable for a flat pointer system without destructive update, if any initial symbolic configuration is considered, but is decidable for initial symbolic configurations with at most one cyclic list. Hence this result shows that even when we take strong restrictions such as flat pointer systems without destructive update, the problem of reachability of a control state is undecidable.

The work we present here extends and completes the results presented in [11]. In this section we establish the decidability of the safety and model-checking problems for two restricted classes of flat pointer systems without destructive update. It is true that these classes are very restricted, but we should see in the next section that it is hard to obtain decidability results without considering such restrictions.

Theorem 4 (Decidability of safety). *For flat pointer systems without destructive update and without alias test, the safety problem is decidable.*

With this theorem, we hence propose a new class of flat pointer systems without destructive update for which the safety problem is decidable, and for this class there is no need to put restriction on the initial configuration.

With the second theorem, we extend to general temporal properties the results expressed in [11] in the case of programs with an acyclic initial configuration.

Theorem 5 (Decidability of model-checking). *For flat pointer systems without destructive update and with an initial acyclic symbolic configuration, the model-checking is decidable.*

The proofs of these theorems rely both on a translation that maps a pointer system without destructive update to a bisimilar counter system. Moreover, this translation, unlike the translation presented previously, preserves the flatness of the systems, and relies on the notion of a new symbolic representation for memory graphs, called roots memory shapes. We first introduce this notion, then present the translation and sketch the proofs of the two theorems.

4.1 Roots memory shape

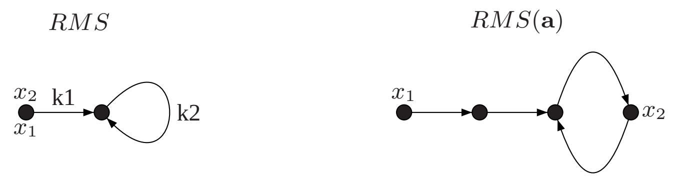
The basic ingredient of the new translation is the notion of roots memory shapes, that are memory shapes in which all the variables appear only on root nodes or on cyclic lists. They can be formally defined as follows :

Definition 9 (Roots memory shape). *A roots memory shape is a memory shape RMS $= (N, succ, loc, l)$ such that for all $n \in N$:*

- *either $loc^{-1}(\{n\}) \neq \emptyset$ and $succ^{-1}(\{n\}) = \emptyset$ (root node);*
- *or $loc^{-1}(\{n\}) = \emptyset$ and $|succ^{-1}(\{n\})| \geq 2$ (shared node);*
- *or $loc^{-1}(\{n\}) \neq \emptyset$ and $succ^{-1}(\{n\}) = \{n\}$ (unshared node on cyclic list).*

Note that we do not really need to impose $loc^{-1}(\{n\}) = \emptyset$ in the second condition, but we do it to simplify the definition of the translation we give later. We suppose that the set of variables is $V = \{x_1, ..., x_m\}$ and we associate with each pointer variable x_i a counter c_i. We define $C_V = \{c_1, ..., c_m\}$ and suppose that $C_V \subset C$. By giving a value for the counters labeling the nodes, and also a value for the counters c_i associated to pointer variables (which was not the case with memory shapes), we can obtain a memory graph from a roots memory shape. Given a roots memory shape RMS and a valuation $\mathbf{a} \in \mathbb{N}^C$, we define the memory graph $RMS(\mathbf{a})$ as follows (see figure 3 for an example): first, we consider the memory graph $MG = (N, succ, loc)$ obtained from the interpretation of RMS as a memory shape; then we define $RMS(\mathbf{a})$ to be the memory graph $MG' = (N', succ', loc')$ where $N' = N$, $succ' = succ$ and $loc'(x_i) = succ^{\mathbf{a}(c_i)}(loc(x_i))$ for all variables x_i (where $succ^j$ represents the j-th successor). Note that some valuations may not be admissible for this definition, as $loc'(x_i)$ may be undefined, or the condition on the absence of memory leaks in the definition of memory graphs may not be satisfied (if all variables located on the same node in RMS all have strictly positive values for their counters). We denote by $\mathcal{RMS}$ the set of roots memory shapes. Since for a finite number of variables the number of memory shapes

$\mathbf{a}(k1) = 2$
$\mathbf{a}(k2) = 2$
$\mathbf{a}(c_1) = 0$
$\mathbf{a}(c_2) = 5$



c_1 and c_2 are respectively associated to x_1 and x_2

Fig. 3. A memory graph associated with a roots memory shape and a valuation

is finite, we deduce that $\mathcal{RMS}$ is also a finite set. Before to give the definition of the new translation, we define here some useful notions on memory graphs. We consider a memory graph $MG = (N, succ, loc)$.

A node $n' \in N \cup \{\texttt{null}, \perp\}$ is said to be *reachable* in MG from an other node $n \in N$ if there exists a path in the graph from n to n', i.e. a finite ordered set of nodes $\{n_1, ..., n_r\} \subseteq N \cup \{\texttt{null}, \perp\}$ such that $n_1 = n$, $n_r = n'$ and $\forall i \in \{1, ..., r-1\}$, $succ(n_i) = n_{i+1}$. For a node $n \in N$, we denote by $\texttt{List}(MG, n)$ the set $\{n' \in N \cup \{\texttt{null}, \perp\} \mid n'$ is reachable from n in $MG\}$.

We introduce then the notion of shared nodes. Given two nodes $n, n' \in N$, we define the set $\texttt{Shared}(MG, n, n') = \texttt{List}(MG, n) \cap \texttt{List}(MG, n')$, which represents the nodes that are shared by the list beginning at the node n and the one beginning at the node n'.

We propose also a notation to represent the set of nodes which lay between two nodes. Let $n, n' \in N$, we define $\texttt{Between}(MG, n, n')$ such that:

– if $n' \notin \texttt{List}(MG, n)$, $\texttt{Between}(MG, n, n') = \emptyset$;
– else if $n' = n$, $\texttt{Between}(MG, n, n') = \emptyset$;
– else ($n' \in \texttt{List}(MG, n)$ and $n' \neq n$) $\texttt{Between}(MG, n, n')$ is the set of nodes $\{n_1, ..., n_r\}$ such that $n_1 = succ(n)$, $n' = succ(n_r)$, $\forall i \in \{1, ..., r-1\}$, $succ(n_i) = n_{i+1}$ and $\forall i \in \{1, ..., r\}$, $n_i \notin \{n, n'\}$.

Furthermore, we will say that a variable $v \in V$ is *single* in MG if:

– $loc^{-1}(loc(v)) = \{v\}$ (i.e. v is the only variable on the node $loc(v)$).

We recall that a node $n \in N$ is called a *root node* if $succ^{-1}(\{n\}) = \emptyset$.

4.2 From pointers to counters

We present now how to define a counter system that faithfully represents a pointer system without destructive update using a translation which preserves the flatness of the system. In this manner, we define two functions $TEST_2$ and $POST_2$, we will then use to build the counter system.

The partial function $TEST_2$ takes as argument a pointer guard $g \in \mathcal{G}$ and a roots memory shape RMS and returns a Presburger-formula ϕ. This function is defined such that the following property is verified : if $\phi = TEST_2(g, RMS)$, for all admissible $\mathbf{a} \in \mathbb{N}^C$ (according to RMS), we have :

$$RMS(\mathbf{a}) \models g \text{ if and only if } \mathbf{a} \models \phi$$

Remark that if RMS is a roots memory shape with a cyclic list and if g is a pointer guard of the form $x_i = x_j$ then the arithmetic formula given by the function $TEST_2$ should use propositions of the form y is divided by x which do not belong to the Presburger arithmetic. In fact, if x_i and x_j are two pointer variables pointing in RMS to the unique node of a cyclic list whose edge is labeled with the counter k, testing if $x_i = x_j$ could be done using the formula $(c_i \geq c_j \Rightarrow k|c_i - c_j) \vee (c_j \geq c_i \Rightarrow k|c_j - c_i)$. To avoid this situation, we restrict the definition domain of $TEST_2$ such that $dom(TEST_2) = \{(g, RMS) \in \mathcal{G} \times \mathcal{RMS} \mid g$ does not use alias test or RMS is acyclic$\}$. The table 2 gives the formal definition of the function $TEST_2$.

As it has been done for the first introduced translation, we now define the partial function $POST_2$ which takes as argument a pointer action $a \in \mathcal{A}$ that is not a destructive update and a root memory shape RMS and returns a pair $((A, \mathbf{b}, \phi), RMS')$ such that the following property is satisfied : if $((A, \mathbf{b}, \phi), RMS') = POST_2(a, RMS)$, for all admissible $\mathbf{a}$ and $\mathbf{a}'$ in $\mathbb{N}^C$ (according to RMS and RMS'), we have :

$$RMS'(\mathbf{a}) = [\![a]\!]_P(RMS(\mathbf{a}')) \text{ if and only if } \mathbf{a} \models \phi \text{ and } \mathbf{a}' = A.\mathbf{a} + \mathbf{b}$$

Note that whereas the function $POST$ was returning a set of pairs $((A, \mathbf{b}, \phi), MS)$ the function $POST_2$ returns an unique pair. This feature allows us to define a translation which preserves the flatness of systems. The table 3 gives the definition of the function $POST_2$ in the case of actions of the form $x_i :=$ NULL. The definition for the others actions is given in the appendix B. Note that sometimes the linear function (A, b) is denoted for instance $c_i := c_j$ which means that the function changes only the value of the counter c_i giving it the value of c_j. In these tables, in the column describing RMS', it is sometimes written for instance $loc'(x_i) = loc(x_j)$, it means that RMS' is obtained by moving in RMS the variable x_i to the node where x_j points on.
The figure 4 presents an example of the results of the computation of $POST_2$. Intuitively, the conditions ensuring some action will not yield memory fault can be defined with a guard on the counter variables, just as in figure 4 where the guard $(c_1 = 0 \vee c_2 = 0)$ ensures the absence of memory leak and the guard $(c_4 < k2)$ the absence of memory violation. A pointer action $x_i := x_j$ will correspond to moving x_i to the location of x_j and doing the linear transformation $c_i := c_j$ on counters. In the case of $x_i := x_j.succ$, it is the same, we move x_i to the location of x_j and we update the counter with the operation $c_i := c_j + 1$. Finally, for the action $x_i :=$ NULL, we do the following operations, we move x_i to `null` and we set the counter c_i to 0.

Using $TEST_2$ and $POST_2$, we can associate with a pointer system $PS = \langle Q_P, E_P \rangle$ the counter system $CS_2(PS) = \langle Q_C, E_C \rangle$ where :

Hypothesis	$TEST_2(g, RMS)$
$g := True$	$True$
$g := \neg g'$	$\neg TEST_2(g, RMS)$
$g := g' \wedge g''$	$TEST_2(g, RMS) \wedge TEST_2(g', RMS)$
$g := Isnull(x_i)$ and $\exists n$ such that $n \in \texttt{List}(RMS, x_j)$ and n is on a cyclic list	$False$
$g := Isnull(x_i)$ and $\texttt{null} \notin \texttt{List}(RMS, loc(x_i))$ and $\nexists n$ such that $n \in \texttt{List}(RMS, x_j)$ and n is on a cyclic list	$False$
$g := Isnull(x_i)$ and $\texttt{null} \in \texttt{List}(RMS, loc(x_i))$ and $\nexists n$ such that $n \in \texttt{List}(RMS, x_j)$ and n is on a cyclic list	$c_i = \Sigma_{n \in \texttt{List}(RMS, loc(x_i)) \cap N} l(n)$
$g := x_i = x_j$ and RMS is acyclic and $\texttt{Shared}(RMS, loc(x_i), loc(x_j)) = \emptyset$	$False$
$g := x_i = x_j$ and RMS is acyclic and $\texttt{Shared}(RMS, loc(x_i), loc(x_j)) \neq \emptyset$	$c_i > \Sigma_{n \in \texttt{List}(RMS, loc(x_i)) \setminus \texttt{Shared}(RMS, loc(x_i), loc(x_j))} l(n) \wedge$ $c_j > \Sigma_{n' \in \texttt{List}(RMS, loc(x_j)) \setminus \texttt{Shared}(RMS, loc(x_i), loc(x_j))} l(n') \wedge$ $c_i - \Sigma_{n \in \texttt{List}(RMS, loc(x_i)) \setminus \texttt{Shared}(RMS, loc(x_i), loc(x_j))} l(n) =$ $c_j - \Sigma_{n' \in \texttt{List}(RMS, loc(x_j)) \setminus \texttt{Shared}(RMS, loc(x_i), loc(x_j))} l(n')$

Table 2. Computation of $TEST_2(g, RMS)$

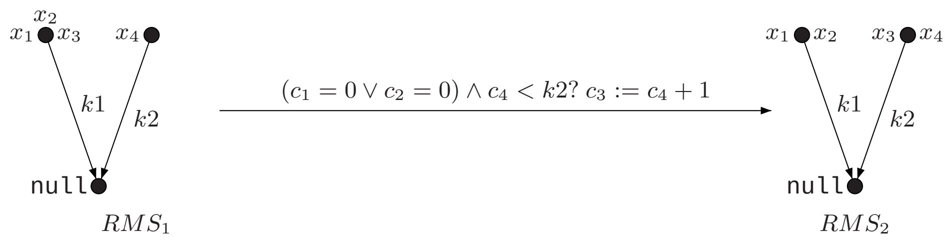


Fig. 4. Effect of the action $x_3 := x_4.succ$ over RMS_1

– $Q_C = Q_P \times \mathcal{RMS}$
– E_C is the transition relation defined by $((q, RMS), (A, b, \phi), (q', RMS')) \in E_C$ if and only if there exists a transition $(q, (g, a), q') \in E_P$ and two Presburger formulae ϕ_1 and ϕ_2 such that $\phi_1 = TEST_2(g, RMS)$ and $((A, \mathbf{b}, \phi_2), RMS') = POST_2(a, RMS)$ and $\phi = \phi_1 \wedge \phi_2$.

Note that since E_P and $Q_P \times \mathcal{RMS}$ are finite, we can effectively build the counter system $CS_2(PS)$.

We will show how PS and $CS_2(PS)$ are related. Let us consider the relation R_T between the configurations of the pointer system PS and the ones of its associated counter system $CS_2(PS)$ defined by :

$$R_T \ = \big\{ \ \big((q, MG)\ ,\ ((q, RMS), \mathbf{a})\big) \mid MG = RMS(\mathbf{a})\big\}.$$

and the relation R_T^{ac} which is the restriction of R_T to acyclic memory graphs :

$$R_T^{ac} \ = \big\{ \ \big((q, MG)\ ,\ ((q, RMS), \mathbf{a})\big) \mid MG = RMS(\mathbf{a}) \text{ and } MG \text{ is acyclic}\big\}.$$

Proposition 1. *For any pointer system without destructive update PS, $CS_2(PS)$ enjoys the following properties:*

1. *$CS_2(PS)$ has the finite monoid property.*
2. *If PS is flat then $CS_2(PS)$ is flat.*
3. *R_T^{ac} is a bisimulation.*
4. *If PS is without alias test, R_T is a bisimulation.*

Idea of the proof : $CS_2(PS)$ has the finite monoid property because all the matrices given by the function $POST_2$ are composed of lines in which all the elements are equal to 0 except one which is equal to 1, and the multiplicative monoid of such a set of matrices is finite. The other points of this proposition are direct consequences of the way we build $CS_2(PS)$ and of the properties of the functions $TEST_2$ and $POST_2$. $\square$

Properties 1 and 2 ensure that we will be able to use theorems 1 and 2 (and also the tool FAST), and properties 3 and 4 are essential to relate counter properties to pointer properties.

Hypothesis	$(A, \mathbf{b})$	ϕ	RMS'
x_i is single in RMS and $loc(x_i) \notin \{\texttt{null}, \perp\}$	$(Id, \overrightarrow{0})$	$False$	RMS
$loc(x_i) \in \{\texttt{null}, \perp\}$	$(Id, \overrightarrow{0})$	$True$	$loc'(x_i) = \texttt{null}$
x_i is not single in RMS and $loc(x_i)$ belongs to a cyclic list	$c_i := 0$	$True$	$loc'(x_i) = \texttt{null}$
x_i is not single in RMS and $loc(x_i)$ does not belong to a cyclic list	$c_i := 0$	$\bigvee_{x_j \in loc^{-1}(\{loc(x_i)\}) \setminus \{x_i\}} c_j = 0$	$loc'(x_i) = \texttt{null}$

Table 3. Computation of $((A, \mathbf{b}, \phi), RMS') = POST_2(a, RMS)$ for the action a of the form $x_i :=$ NULL.

4.3 Translating the symbolic configurations

To conclude the proofs of theorems 4 and 5, we have to extend the translation to symbolic configurations and temporal formulae. We shall define $T(\texttt{INIT})$, $T(\texttt{BAD})$ as two symbolic configurations of $CS_2(PS)$ that correspond to $\texttt{INIT}$ and $\texttt{BAD}$ in PS, and we must moreover define $T(\Phi) \in$ FOCTL*(Pr) that corresponds to $\Phi \in$ CTL$^*_{mem}$. The key of this translation is to find an effective symbolic representation $T(q, (MS, \psi))$ for the set of (counter systems) configurations that are bisimilar to (pointer systems) configurations in $[\![(q, (MS, \psi))]\!]$. That is, for all roots memory shape, we should represent the set of counters values :

$$T_{RMS}(MS, \psi) \ = \ \{\mathbf{a} \mid \ RMS(\mathbf{a}) \in [\![MS, \psi]\!]\}.$$

This set is not Presburger definable in general, due to cyclic lists, but, as we will see with the next proposition, it is definable in the logic $\mathcal{L}^{\exists}_{|} = \langle \mathbb{N}, +, |, = \rangle^{\exists}$ of the existentially quantified formulae of the Presburger arithmetic with divisibilty. An essential result for our proofs is that the satisfiability problem for this logic is decidable [23].

Proposition 2. *$T_{RMS}(MS, \psi)$ is definable in $\mathcal{L}^{\exists}_{|}$. Moreover, if MS is acyclic, then it is definable in Presburger arithmetic.*

Before to give the proof of this proposition, we introduce some preliminary notions. We will say that a memory shape MS is *compatible* with a roots memory shape RMS (denoted $MS \sqsubset RMS$) if and only if $[\![(MS, True)]\!] \cap [\![(RMS, True)]\!] \neq \emptyset$. Intuitively a memory shape MS is compatible with a roots memory shape RMS if it is possible to obtain a graph isomorphic to RMS from MS moving the pointer variables to a root node or to a node in a cyclic list they are connected to in MS. To be more formal, we introduce the notion of *compatibility function*.

Let $MS = (N, succ, loc)$ be a memory shape and $RMS = (N', succ', loc', l')$. We denote by N_R (resp. N'_R) the set of root nodes in MS (resp. in RMS), N_2 (resp. N'_2) the set of nodes with at least two predecessors in MS (resp. in RMS) and N_C (resp. N'_C) the set of nodes belonging to a cyclic list not reachable from a root node in MS (resp. in RMS). We say that a function $g : N' \rightarrow N$ is a *compatibility function* between MS and RMS if g is a total injective function such that:

- $g(N'_R) = N_R$, $g(N'_2) = N_2$ and $g(N'_C) \subseteq N_C$;
- for all nodes $n, n' \in N'$, $n' \in \texttt{List}(RMS, n)$ if and only if $g(n') \in \texttt{List}(MS, g(n))$;
- for all nodes $n \in N'$, $\texttt{null} \in \texttt{List}(RMS, n)$ if and only if $\texttt{null} \in \texttt{List}(MS, g(n))$;
- for all nodes $n \in N'$, $\perp \in \texttt{List}(RMS, n)$ if and only if $\perp \in \texttt{List}(MS, g(n))$;
- for all variables $v \in V$, $loc(v) \in \texttt{List}(MS, g(loc'(v))$.

We can then deduce the following lemma.

Lemma 1. *Let MS be a memory shape and RMS be a roots memory shape. $MS \sqsubset RMS$ if and only if there exists a compatibility function between MS and RMS.*

We now give the proof of proposition 2.

Proof : In this proof we associate the set $T_{RMS}(MS, \psi)$ with the arithmetic formula that characterizes it. Let (MS, ψ) be a symbolic memory shape and RMS a roots memory shape. We suppose $C_{MS} = \{k_1, ..., k_m\}$, $MS = (N, succ, loc, l)$ and $RMS = (N', succ', loc', l')$. We build a logic formula $T_{RMS}(MS, \psi)$ over $C_{RMS} \cup C_V$ as follows:

- If $MS \not\sqsubset RMS$, $T_{RMS}(MS, \psi) = False$;
- Otherwise Let g be a function of compatibility between MS and RMS;
  1. Rename in ψ and in MS all the counters k_i with $\bar{k}_i$ Such that for all $i \in \{1, ..., m\}$, $\bar{k}_i \notin C$ to obtain a formula $\bar{\psi}$, and we denote by $\bar{l}$ the function which associates to each node $n \in N$ the counter $\bar{k}_i$ such that $l(n) = k_i$.
  2. For each node n in N', we define the formula ϕ_n to ensure that the length on the graphs correspond:
  $$\phi_n := l'(n) = \bar{l}(g(n)) + \sum_{n' \in \texttt{Between}(MS, g(n), g(succ'(n)))} \bar{l}(n')$$
  3. Let $N_{Cl} \subseteq N$ (resp. $N'_{Cl} \subseteq N'$) the set of nodes of MS (resp. of RMS) which belong to a cyclic list. For each variable $x_i \in V$, we define a formula ϕ_i to ensure the pointer variables are located at the same position.
     - If $loc(x_i) \in \{\texttt{null}, \perp\}$, then
     $$\phi_i := c_i = 0$$

- If $loc(x_i) \notin N_{Cl}$:

$$\phi_i := \bar{l}(g(loc'(x_i))) + \sum_{n \in \texttt{Between}(MS, g(loc'(x_i)), loc(x_i))} \bar{l}(n) = c_i$$

- Otherwise if $loc(x_i) \in N_{Cl}$ and $loc'(x_i) \in N'_{Cl}$ (in this case, the cyclic list where x_i points to is necessarily not reachable from a root node, we write $L = \sum_{n \in \texttt{List}(MS, loc(x_i))} \bar{l}(n)$ the size of the cyclic list x_i is on then:

$$\begin{gathered}\phi_i := \bigwedge_{\{x_h | loc(x_h) = loc(x_l)\}} \big(c_i.\texttt{mod}(L) = c_h.\texttt{mod}(L)\big) \wedge \\ \bigwedge_{\{x_j | loc(x_j) \in \texttt{List}(MS, loc(x_i) \setminus loc(x_i)\}} \big(c_j.\texttt{mod}(L) = \\ (c_i + \bar{l}(loc(x_i)) + \sum_{n \in \texttt{Between}(MS, loc(x_i), loc(x_j))} \bar{l}(n)).\texttt{mod}(L)\big)\end{gathered}$$

- Otherwise ($loc(x_i) \in N_{Cl}$ and $loc'(x_i) \notin N'_{Cl}$), and we denote by $L = \sum_{n \in \texttt{List}(MS, loc(x_i))} \bar{l}(n)$, intuitively L encodes the size of the cyclic list pointed to by x_i and $S = \sum_{n \in \texttt{List}(MS, g(loc'(x_i))) \setminus N_{Cl}} \bar{l}(n)$ represents the length of the segment leaving from $g(loc'(x_i))$ and finishing on the cyclic list, we then have :

$$\begin{gathered}\phi_i = \big(c_i \geq S \wedge \\ \bar{l}(g(loc'(x_i))) + \sum_{n \in \texttt{Between}(MS, g(loc'(x_i)), loc(x_i))} \bar{l}(n) = \\ S + (c_i - S).\texttt{mod}(L)\big)\end{gathered}$$

4. Finally we obtain:

$$T_{RMS}(MS, \psi) := \exists \bar{k}_1 ... \exists \bar{k}_m . \bar{\psi} \wedge \bigwedge_{n \in N'} \phi_n \bigwedge_{x_i \in V} \phi_i$$

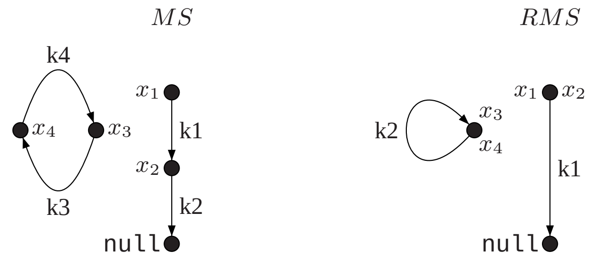
Remark that if MS is acyclic, then by construction $T_{RMS}(MS, \psi)$ is a Presburger formula. And in the other cases, $T_{RMS}(MS, \psi)$ can be rewritten into an equivalent formula of the logic $\mathcal{L}^{\exists}_{|}$. This is due to the fact that $a.mod[c] = b.mod[c]$ is equivalent to the formula $c \mid (a - b)$ and that any Presburger formula can be rewritten into an equivalent Presburger formula with only existential quantifiers (by elimination of the quantifiers [18]). $\square$

When we associate an arithmetic formula φ over the counters $C_{RMS} \cup C_V$ with a roots memory shape RMS, we denote by $[\![(RMS, \varphi)]\!]$ the set of memory graphs, $\{RMS(\boldsymbol{a}) \mid \boldsymbol{a} \models \varphi\}$ We have then the following result by construction of T_{RMS} :

Lemma 2. *Let (MS, ψ) be a symbolic memory shape and RMS a roots memory shape.*

1. *If $MS \sqsubset RMS$, $[\![(RMS, T_{RMS}(MS, \psi))]\!] = [\![(MS, \psi)]\!]$.*
2. *If $MS \not\sqsubset RMS$, $[\![(RMS, T_{RMS}(MS, \psi))]\!] = \emptyset$.*

We will now see how we use this different results to prove the theorems 4 and 5. The main idea consists in reducing the problems of safety and model-checking over pointer systems without destructive update to similar problems over counter systems.



$$T_{RMS}(MS,\psi) := \exists \bar{k}_1.\exists \bar{k}_2.\exists \bar{k}_3.\exists \bar{k}_4.\bar{\psi}\wedge$$
$$k1 = \bar{k}1 + \bar{k}2 \wedge k2 = \bar{k}3 + \bar{k}4\wedge$$
$$c_1 = 0 \wedge c2 = \bar{k}1\wedge$$
$$c_4.\mathtt{mod}[k2] = (c_3 + \bar{k}3).\mathtt{mod}[k2]\wedge$$
$$c_3.mod[k2] = (c_4 + \bar{k}4).\mathtt{mod}[k2]$$

Fig. 5. An example of the computation of the formula $T_{RMS}(MS,\psi)$

4.4 Proof of theorem 4

Let $(q,(MS,\psi))$ be a symbolic configuration of a pointer system. We define :

$$T(q,(MS,\psi)) = \bigcup_{RMS\in\mathcal{RMS}} ((q,RMS),T_{RMS}(MS,\psi))$$

$T(q,(MS,\psi))$ represents a symbolic configuration for $CS_2(PS)$ and furthermore by lemma 2, we have that $[\![(MS,\psi)]\!] = \bigcup_{RMS\in\mathcal{RMS}}[\![(RMS,T_{RMS}(MS,\psi))]\!]$. The properties of the arithmetic formula $T_{RMS}(MS,\psi)$ and the fact that the relation R_T is a bisimulation between the transition system of PS and the one of $CS_2(PS)$ allows us to state the following lemma.

Lemma 3. *Let PS be a pointer system without destructive update and without alias test, $(q_0,(MS_0,\psi_0))$ an initial symbolic configuration, and $(q_B,(MS_B,\psi_B))$ a bad symbolic configuration for PS. Then :*

$$\mathtt{Reach}(PS,[\![(q_0,(MS_0,\psi_0))]\!]) \cap [\![(q_B,(MS_B,\psi_B))]\!] = \emptyset$$

if and only if

$$\mathtt{Reach}(CS_2(PS),[\![T(q_0,(MS_0,\psi_0))]\!]) \cap [\![T(q_B,(MS_B,\psi_B))]\!] = \emptyset$$

Using this last lemma and previous results, we can prove the theorem 4.

Proof of theorem 4
Let PS be a flat pointer system without destructive update and without alias test, $(q_0,(MS_0,\psi_0))$ an initial symbolic configuration, and $(q_B,(MS_B,\psi_B))$ a bad symbolic configuration for PS. By proposition 1, the counter system $CS_2(PS)$ is flat and

has the finite monoid property. By lemma 3, the considered safety problem reduces to the safety problem for $CS_2(PS)$ with $T(q_0, (MS_0, \psi_0))$ as initial symbolic configuration and $T(q_B, (MS_B, \psi_B))$ as bad symbolic configuration. Furthermore, since for all memory shapes MS, for all Presburger formula ψ over C_{MS} and for all roots memory shape RMS , $T_{RMS}(MS, \psi)$ is a formula of $\mathcal{L}^{\exists}_{|}$, we can deduce from the theorem 1, that this last problem reduces to the satisfiability problem of a formula of $\mathcal{L}^{\exists}_{|}$, which is a decidable problem [23].□

4.5 Proof of theorem 5

We consider then a formula Φ of CTL*(MS). We define the formula $T(\Phi)$ by induction as follows:

– if $\Phi := (q, (MS, \psi))$ with MS acyclic then :
  $T(\Phi) := \bigvee_{RMS \in \mathcal{RMS}}((q, RMS), T_{RMS}(MS, \psi))$;
– if $\Phi := (q, (MS, \psi))$ with MS not acyclic then :
  $T(\Phi) := \bigvee_{RMS \in \mathcal{RMS}}((q, RMS), False)$;
– if $\Phi := \neg\Phi'$ then $T(\Phi) := \neg T(\Phi')$;
– if $\Phi := \Phi' \wedge \Phi''$ then $T(\Phi) := T(\Phi') \wedge T(\Phi'')$;
– if $\Phi := \mathsf{X}\Phi'$ then $T(\Phi) := \mathsf{X}T(\Phi')$;
– if $\Phi := \Phi'\mathsf{U}\Phi''$ then $T(\Phi) := T(\Phi')\mathsf{U}T(\Phi'')$;
– if $\Phi := \mathsf{A}\Phi'$ then $T(\Phi) := \mathsf{A}T(\Phi')$.

Since, for all acyclic memory shapes MS, for all Presburger formulae ψ over C_{MS} and for all roots memory shape RMS, $T_{RMS}(MS, \psi)$ is a Presburger formula, we deduce that:

Remark 1. For all formulae Φ of CTL*(MS), $T(\Phi)$ is a formula of CTL*(Pr).

Moreover, the following lemma holds.

Lemma 4. *Let PS be a pointer system without destructive update, $(q_0, (MS_0, \psi_0))$ an initial acyclic symbolic configuration of PS, and Φ a formula of CTL*(MS). We have that $\pi, 0 \models \Phi$ for all configurations paths π of $CS_2(PS)$ such that $\pi(0) \in [\![(q_0, (MS_0, \psi_0))]\!]$, if and only if $\pi', 0 \models T(\Phi)$ for all configuration paths π' of the transition system $TS(CS_2(PS))$ such that $\pi'(0) \in [\![T(q_0, (MS_0, \psi_0))]\!]$.*

Proof
This lemma can be proved by induction on the length of the formula Φ using the definition of $T(q_0, (MS_0, \psi_0))$. The first case of the induction is when Φ is of the form $(q, (MS, \psi))$, and it is proved using lemma 2. The other cases are then proved using that the relation R_T^{ac} is a bisimulation between the transition system of PS and the one of $CS_2(PS)$.□

This allows us to conclude the result of theorem 5.

Proof of theorem 5
Let PS be a flat pointer system without destructive update, $(q_0, (MS_0, \psi_0))$ an initial

acyclic symbolic configuration of PS, and Φ a formula of CTL$^*_{mem}$. By proposition 1, the counter system $CS_2(PS)$ is flat and has the finite monoid property. Besides, by lemma 4, the considered problem reduces to the model-checking problem for $CS_2(PS)$ with the initial symbolic configuration $T(q_0, (MS_0, \psi_0))$ and the FOCTL*(Pr) formula $T(\Phi)$. Hence using theorem 2, we can deduce that the model-checking problem for PS with $(q_0, (MS_0, \psi_0))$ as symbolic initial configuration and Φ as temporal formula is decidable. $\Box$

5 Undecidability results

In this section, we show that the decidability results we obtained for safety and temporal properties are tight. In particular, these results become false if one relaxes any hypothesis. For instance, theorem 4 does not hold without the hypothesis of absence of alias test (this is proved in [11]). We list here some new decidability results for some classes of pointer systems very close to the ones we studied in the previous section. All our undecidability results are based on a reduction from satisfiability of Diophantine equations, which is known as undecidable.
Diophantine equations are equations of the form $P(\mathbf{k}) = 0$ where P is some polynomial over naturals and $\mathbf{k}$ a vector of positive integer variables. As explained in [11], Diophantine equations can be encoded as a conjunction of arithmetic formulae of the form $k = k' + k''$ or $k = lcm(k', k'')$ or $k = j$ where k, k', k'' are counter variables and j is a positive integer; the satisfiability problem of such formulae is then undecidable. Below, we use the term of Diophantine equations to describe such conjunctions of arithmetic formulae. Following [11], we now associate with a Diophantine equation $\mathcal{E}$ a pointer system $PS_{\mathcal{E}}$ for which a certain safety (resp. temporal) property holds if and only if $\mathcal{E}$ has a solution.
Our first result shows that theorem 4 does not extend to model-checking :

Theorem 6. *The model-checking problem is undecidable for flat pointer systems without destructive update and without alias test.*

The proof of this result is an adaption of the undecidability proof of [11]. Let us consider $\mathcal{E} = \bigwedge \mathcal{E}_i$ a Diophantine equation. We define the tuple $(MS_{\mathcal{E}}, PS_{\mathcal{E}}, \Phi_E)$ such that: $MS_{\mathcal{E}}$ is a memory shape defined by $MS_{\mathcal{E}} = MS_1 \uplus .. \uplus MS_n$ (where $\uplus$ represents the disjunctive union of memory shapes and n coincides with the number of conjuncts in $\mathcal{E}$); $\Phi_{\mathcal{E}} = \bigwedge \Phi_i$ is a temporal property; and $PS_{\mathcal{E}}$ is a flat pointer system without destructive update and without alias test defined by $PS_{\mathcal{E}} = PS_1; ..; PS_n$, where ; is the sequential composition and each PS_i is a flat program (without destructive update and without alias test) that either exits correctly (`exit(0)`) and launches the execution of PS_{i+1} or aborts (`exit(1)`):

- if $\mathcal{E}_i$ is $k = k' + k''$, MS_i is the memory shape with three disjoint list segments of length k, k', k'', whose heads are pointed to by some set of fresh variables $\{\mathtt{x}, \mathtt{x}_0\}, \{\mathtt{y}, \mathtt{y}_0\}, \{\mathtt{z}, \mathtt{z}_0\}$ respectively, and whose tails are on `null`. The pointer system PS_i can be described by the program :

```
while (y≠NULL) do x=x.succ; y=y.succ;end while;
while (z≠NULL) do x=x.succ; z=z.succ;end while;
if (x=NULL) then exit(0); else exit(1);
```

The temporal property Φ_i expresses that the `exit(0)` is reached (which means without error).

- if $\mathcal{E}_i$ is $k = lcm(k', k'')$, we define MS_i to be the memory shape with two disjoint cyclic lists pointed to by sets of fresh variables $\{\mathtt{y}, \mathtt{y}_0\}, \{\mathtt{z}, \mathtt{z}_0\}$ respectively, of lengths k', k'', and a single list of length k whose head is pointed by some set of fresh variables $\{\mathtt{x}, \mathtt{x}_0\}$ and ends on `null`. We moreover define PS_i by the program :

```
while (x≠NULL) do x=x.succ;y=y.succ;z=z.succ;end while;
exit(0);
```

  The temporal property Φ_i expresses that, each state of the loop verifies $(\mathtt{y}, \mathtt{z}) \neq (\mathtt{y}_0, \mathtt{z}_0)$ until $(\mathtt{x}, \mathtt{y}, \mathtt{z}) = (\mathtt{null}, \mathtt{y}_0, \mathtt{z}_0)$.
- if $\mathcal{E}_i$ is $k = j$, MS_i is the memory shape with one list segment of length k, whose head is pointed on by some set of fresh variables $\{\mathtt{x}, \mathtt{x}_0\}$ and whose tail is on `null`. The pointer system PS_i can be described by a program without loop which performs i times `x:=x.succ` and does `exit(0)` at the end if x points on `null` and `exit(1)` otherwise. The temporal property Φ_i expresses that the `exit(0)` is reached (which means without error).

By construction the Diophantine equation represented by the formula $\mathcal{E}$ has a solution if and only if there exists a configuration path π in $PS_{\mathcal{E}}$ such that $\pi(0) \in [\![(MS_{\mathcal{E}}, True)]\!]$ and $\pi, 0 \models \Phi_{\mathcal{E}}$, which is true if and only if the answer to the model-checking problem on the inputs $PS_{\mathcal{E}}$, $(MS_{\mathcal{E}}, True)$ and $\neg\Phi_{\mathcal{E}}$ is no. We deduce from this the result of theorem 6.

This last proof and the proof of undecidability in [11] build flat pointer systems working over cyclic lists. Hence one may think that the key point of these undecidability results is the use of cyclic lists. We show here that this is not the case.
First, we say that a pointer system $PS = \langle Q, E\rangle$ associated with an initial symbolic configuration $(q_0, (MS_0, \psi_0))$ is an *acyclic pointer system* if the memory shape MS_0 is acyclic and all reachable memory graphs MG (i.e. $\exists q \in Q$ such that $(q, MG) \in \mathtt{Reach}(PS, [\![(q_0, (MS_0, \psi_0))]\!])$) are acyclic.

Theorem 7. *The safety problem is undecidable for acyclic flat pointer systems.*

Proof : We adapt the previous proof, that works with cyclic lists, to a system that does not work with cyclic lists but can use destructive updates. Note that we only need to adapt the program that tests the equation $k = lcm(k', k'')$, the rest of the proof being then similar. Consider a list segment l whose head is pointed to by $\{\mathtt{h}, \mathtt{h}'\}$, whose tail is on `null`, and whose last node before `null` is pointed by `t`; we thus define the subprogram :
`rotate`(l)=`h=h.succ;h'.succ=NULL;t.succ=h';t=h';h'=h;`
This program moves the first element of the list at the tail of the list. We now consider the memory shape with two such disjoint list segments l_1, l_2 with some pairs of extra

variables $\{\mathtt{y}, \mathtt{y}_0\}$, $\{\mathtt{z}, \mathtt{z}_0\}$ at some point in the middle of l_1 and l_2 respectively, and a disjoint standard list segment l whose head is pointed by $\{\mathtt{x}, \mathtt{x}_0\}$ and whose tail is pointed to `null`. Counters k, k', k'' represent the total length of the lists l, l_1, l_2 respectively. Then the following program exits normally if and only if $k = lcm(k', k'')$:

```
while (x≠null and not((y=y0) and (z=z0))) do
x=x.succ;y=y.succ;z=z.succ;
rotate(l1);rotate(l2);end while;
if (x=null and y=y0 and z=z0) then exit(0) else exit(1);
```

which ends the proof.□

Note that this last result answers a problem that was stated as open in [11].

6 Conclusion

Flat pointer systems	Initial symbolic configuration	Safety problem	Model-checking problem
Without destructive update	Acyclic	Decidable	Decidable
Without destructive update	No Restriction	Undecidable	Undecidable
Without destructive update and without alias test	No Restriction	Decidable	Undecidable
Acyclic	Acyclic	Unecidable	Undecidable

Table 4. Summary of main results

We have proposed a framework for model-checking pointer systems without destructive update. Given any pointer system without destructive update, one may translate it into a bisimilar counter system having the finite monoid property. Then a counter model-checker, FAST for instance, may verify it; if the counter system is flat (or flattable [4]), then FAST will terminate computing the Presburger representation of the reachability relation.

It was known that safety was undecidable for flat pointer systems without destructive update. We have completed the classification of flat pointer systems without destructive update in showing that the model-checking problem becomes decidable for flat pointer systems without destructive update with an initial acyclic configuration. We prove that if we replace the acyclic hypothesis by the hypothesis of absence of alias test, then safety remains decidable but model-checking becomes undecidable. Moreover, if we remove the hypothesis that the system is without destructive update, for even acyclic flat pointer systems the safety problem is undecidable. The table 4 contains a summary of the main decidability results when considering flat pointer systems

References

1. A. Annichini, A. Bouajjani, and M. Sighireanu. Trex: A tool for reachability analysis of complex systems. In *CAV'01*, volume 2102 of *LNCS*, pages 368–372. Springer, 2001.
2. S. Bardin, A. Finkel, and J. Leroux. FASTer acceleration of counter automata in practice. In *TACAS'04*, volume 2988 of *LNCS*, pages 576–590. Springer, 2004.
3. S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer*, 2008. To appear.
4. S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *ATVA'05*, volume 3707 of *LNCS*, pages 474–488. Springer, 2005.
5. S. Bardin, A. Finkel, É. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. In *AVIS'06*, 2006.
6. S. Bardin, A. Finkel, and D. Nowak. Toward symbolic verification of programs handling pointers. In *AVIS'04*, 2004.
7. J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV'06*, volume 4144 of *LNCS*, pages 386–400, 2006.
8. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV'06*, volume 4144 of *LNCS*, pages 517–531. Springer, 2006.
9. A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *TACAS'05*, volume 3440 of *LNCS*, pages 13–29. Springer, 2005.
10. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *SAS'06*, volume 4134 of *LNCS*, pages 52–70. Springer, 2006.
11. M. Bozga and R. Iosif. On flat programs with lists. In *VMCAI'07*, volume 4349 of *LNCS*, pages 122–136. Springer, 2007.
12. R. Brochenin, S. Demri, and É. Lozes. Reasoning about sequences of memory states. In *LFCS'07*, volume 4514 of *LNCS*, pages 100–114. Springer, June 2007.
13. S. Demri, A. Finkel, V. Goranko, and G. van Drimmelen. Towards a model-checker for counter systems. In *ATVA'06*, volume 4218 of *LNCS*, pages 493–507. Springer, 2006.
14. D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? In *FSTTCS'04*, volume 3328 of *LNCS*, pages 250–262. Springer, 2004.
15. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
16. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS'99*, pages 352–359. IEEE Computer Society Press, 1999.

17. A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FSTTCS'02*, volume 2556 of *LNCS*, pages 145–156. Springer, 2002.
18. S. Ginsburg and E. H. Spanier. Semigroups, presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.
19. L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *SAS'06*, volume 4134 of *LNCS*, pages 144–160. Springer, 2006.
20. Homepage of LASH. http://www.montefiore.ulg.ac.be/∼boigelot/research/lash .
21. J. Leroux and G. Sutre. Flat counter automata almost everywhere! In *ATVA'05*, volume 3707 of *LNCS*, pages 489–503. Springer, 2005.
22. T. Lev-Ami and M. Sagiv. Tvla: A system for implementing static analyses. In *SAS'00*, volume 1824 of *LNCS*, pages 280–301. Springer, 2000.
23. L. Lipshitz. The diophantine problem for addition and divisibility. *Transactions of the American Mathematical Society*, 235:271–283, 1978.
24. S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening of shape analyses based on separation logic. In *SAS'07*, LNCS. Springer, 2007. To appear.
25. A. Mandel and I. Simon. On finite semigroups of matrices. *Theoretical Computer Science*, 5(2):101–111, 1977.
26. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI'01*, pages 221–231. ACM, 2001.
27. A. Podelski and T. Wies. Boolean heaps. In *SAS'05*, volume 3672 of *LNCS*, pages 268–283. Springer, 2005.
28. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *CAV'98*, volume 1427 of *LNCS*, pages 88–97. Springer, 1998.
29. E. Yahav, T. W. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP'03*, volume 2618 of *LNCS*, pages 204–222. Springer, 2003.

A Examples of programs

In this section, we give the description of some of the programs which feature in the table 1.

The program `split` :

```
void split(List x){
 List y,z,t,u;
 u=NULL;
 y=x;
 while(y!=NULL){
  t=y->next;
  z=t->next;
  t->next=u;
  y->next=z;
  u=t;
  y=z;}
 }
```

This program is safe when the acyclic list given in input (pointed to by x) has an even length.

The program `delete(n)` :

```
void delete(List x,int n){
 List y;
 int i=n;
 while(i!=0){
  y=x->next;
  free(x);
  x=y;
  i--;
 }
```

This program is safe when the acyclic list given in input (pointed to by x) has a length greater than the integer n.

The program `parse-cyclic-acyclic` :

```
void parse-cyclic-acyclic(List x,List t){
 List y,u;
 y=x;
 u=t;
 while(y!=NULL){
  y=y->next;
  u=u->next;}
 u->next=NULL;
 }
```

For this last program, we assume that the variable x is pointed to an acyclic list, and the variable t to a cyclic list. One can check that this program yields a memory leak when the number of elements in the list pointed to by t does not divides the number of elements in the list pointed to by x to which we add 1.

B Description of the translation $POST_2$

Hypothesis	$(A, \mathbf{b})$	ϕ	RMS'
x_i is single in RMS and $loc(x_i) \notin \{\texttt{null}, \perp\}$ and $x_i \neq x_j$	$(Id, \overrightarrow{0})$	$False$	RMS
x_i is single in RMS and $loc(x_i) \in \{\texttt{null}, \perp\}$ and $x_i \neq x_j$	$c_i := c_j$	$True$	$loc'(x_i) = loc(x_j)$
$x_i = x_j$	$(Id, \overrightarrow{0})$	$True$	RMS
x_i is not single in RMS and $x_i \neq x_j$ and $loc(x_i)$ belongs to a cyclic list	$c_i := c_j$	$True$	$loc'(x_i) = loc(x_j)$
x_i is not single in RMS and $x_i \neq x_j$ and $loc(x_i)$ does not belong to a cyclic list	$c_i := c_j$	$\bigvee_{x_l \in loc^{-1}(\{loc(x_i)\}) \setminus \{loc(x_i)\}} c_l = 0$	$loc'(x_i) = loc(x_j)$

Table 5. Computation of $((A, \mathbf{b}, \phi), RMS') = POST_2(a, RMS)$ for the action a of the form $x_i := x_j$

Hypothesis	$(A, \mathbf{b})$	ϕ	RMS'
$loc(x_j) \in \{\texttt{null}, \bot\}$	$(Id, \overrightarrow{0})$	$False$	RMS
x_i is single in RMS and $loc(x_i)$ is not on a cyclic list and $loc(x_i) \notin \{\texttt{null}, \bot\}$	$(Id, \overrightarrow{0})$	$False$	RMS
x_i is single in RMS and $loc(x_i)$ is on a cyclic list and $x_i = x_j$	$(Id, \overrightarrow{0})$	$True$	RMS
x_i is single in RMS and $loc(x_i)$ is on a cyclic list and $x_i \neq x_j$	$(Id, \overrightarrow{0})$	$False$	RMS
$loc(x_i) \in \{\texttt{null}, \bot\}$ and $\exists n$ such that $n \in \texttt{List}(RMS, x_j)$ and n is on a cyclic list	$c_i := c_j + 1$	$True$	$loc'(x_i) = loc(x_j)$
$loc(x_i) \in \{\texttt{null}, \bot\}$ and $\nexists n$ such that $n \in \texttt{List}(RMS, x_j)$ and n is on a cyclic list and $loc(x_j) \notin \{\texttt{null}, \bot\}$	$c_i := c_j + 1$	$c_j < \Sigma_{n \in \texttt{List}(RMS, x_j) \cap N} l(n)$	$loc'(x_i) = loc(x_j)$

Table 6. Computation of $((A, \mathbf{b}, \phi), RMS') = POST_2(a, RMS)$ for the action a of the form $x_i := x_j.succ$ (I)

Hypothesis	$(A, \mathbf{b})$	ϕ	RMS'
x_i is not single in RMS and x_i does belong to a cyclic list and $\exists n$ such that $n \in \texttt{List}(RMS, x_j)$ and n is on a cyclic list	$c_i := c_j + 1$	$\bigvee_{x_l \in loc^{-1}(\{loc(x_i)) \setminus \{loc(x_i)\}\}} c_l = 0$	$loc'(x_i) = loc(x_j)$
x_i is not single in RMS and x_i does not belong to a cyclic list and $\nexists n$ such that $n \in \texttt{List}(RMS, x_j)$ and n is on a cyclic list and $loc(x_j) \notin \{\texttt{null}, \perp\}$	$c_i := c_j + 1$	$c_j < \Sigma_{n \in \texttt{List}(RMS, x_j) \cap N} l(n) \wedge \bigvee_{x_l \in loc^{-1}(\{loc(x_i)\}) \setminus \{loc(x_i)\}} c_l = 0$	$loc'(x_i) = loc(x_j)$
x_i is not single in RMS and x_i belongs to a cyclic list and $\exists n$ such that $n \in \texttt{List}(RMS, x_j)$ and n is on a cyclic list	$c_i := c_j + 1$	$True$	$loc'(x_i) = loc(x_j)$
x_i is not single in RMS and x_i belongs to a cyclic list and $\nexists n$ such that $n \in \texttt{List}(RMS, x_j)$ and n is on a cyclic list and $loc(x_j) \notin \{\texttt{null}, \perp\}$	$c_i := c_j + 1$	$c_j < \Sigma_{n \in \texttt{List}(RMS, x_j) \cap N} l(n)$	$loc'(x_i) = loc(x_j)$

Table 7. Computation of $((A, \mathbf{b}, \phi), RMS') = POST_2(a, RMS)$ for the action a of the form $x_i := x_j.succ$ (II)