Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking*

Ahmed Bouajjani[1], Peter Habermehl[1], Pierre Moro[1], and Tomáš Vojnar[2]

[1] LIAFA, University of Paris 7, Case 7014, 2 place Jussieu, F-75251 Paris 5, France.
e-mail: {abou,haberm,moro}@liafa.jussieu.fr
[2] FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic.
e-mail: vojnar@fit.vutbr.cz

Abstract. We address the problem of automatic verification of programs with dynamic data structures. We consider the case of sequential, non-recursive programs manipulating 1-selector-linked structures such as traditional linked lists (possibly sharing their tails) and circular lists. We propose an automata-based approach for a symbolic verification of such programs using the regular model checking framework. Given a program, the configurations of the memory are systematically encoded as words over a suitable finite alphabet, potentially infinite sets of configurations are represented by finite-state automata, and statements of the program are automatically translated into finite-state transducers defining regular relations between configurations. Then, abstract regular model checking techniques are applied in order to automatically check safety properties concerning the shape of the computed configurations or relating the input and output configurations. For this particular purpose, we introduce new techniques for the computation of abstractions of the set of reachable configurations and to refine these abstractions if spurious counterexamples are detected. Finally, we present experimental results showing the applicability of the approach and its efficiency.

1 Introduction

In this paper, we address the problem of automatic verification of programs with *dynamic linked data structures*. Such programs are in general difficult to write and understand, and so the possibility of their *formal verification* is highly desirable. Formal verification of such programs is, however, a very difficult task too. Dynamic allocation leads to a necessity of dealing with infinite state spaces. The objects to be dealt with are in general graphs whose shape is difficult to be restricted in advance. The problem is that the linked data structures may fulfil some shape invariants at certain program points, but these invariants may be temporarily broken in various ways while performing some operations over the data structures.

We consider in this work the case of sequential non-recursive programs manipulating structures with one next pointer such as traditional singly-linked lists and circular lists (possibly sharing their parts) that belong among the most commonly used structures in practice. We propose an automata-based approach for symbolic verification of

* This work was supported in part by the French Ministry of Research (ACI project Securité Informatique) and the Czech Grant Agency (projects GA CR 102/04/0780 and 102/03/D211).

such programs using the *regular model checking framework* [11, 19, 3]. To the best of our knowledge, this is the first time regular model checking is systematically used in this area—so far, there has only been an isolated ad-hoc attempt to do so in [2].

As our first contribution, we provide a *systematic encoding* of the configurations of considered programs as words over a suitable finite alphabet. Potentially infinite sets of configurations can then be represented by finite-state automata. Moreover, we propose an *automatic translation* of non-recursive sequential C-like programs (without pointer arithmetics and with suitably abstracted non-pointer data values) into finite-state transducers applicable to the sets of program configurations represented by automata and defining regular relations between these configurations. The translation is done statement-by-statement, and one can then either take a union of all statement transducers or use them separately.

By repeatedly applying the transducer (or transducers) representing a program to the automaton encoding a set of possible initial configurations, one can obtain the sets of configurations reachable in any finite number of steps. It is, however, usually impossible to obtain the set of all reachable configurations in this way—the computation will not stop for most programs with loops. One thus has to consider techniques that will accelerate the computation achieving termination as often as possible—a general termination result cannot be obtained as the verification problem considered is clearly undecidable.

In the literature, several different general-purpose techniques have been proposed to *accelerate the computation* of reachable states in regular model checking. They include, e.g., widening [3, 17], collapsing of automata states based on the history of their creation by composing transducers [10, 1], abstraction of automata [2], or inference of languages [6]. In this work, however, as our further contribution, we propose a new set of acceleration techniques that are more tailored for the given domain and thus promise much better performance results. These techniques are based on new language abstractions, which contrary to those introduced in [2], are not defined on the representation structures (i.e. the automata representing sets of configurations), but defined on words (corresponding to configurations). Such abstractions are defined by means of finite-state transducers following different generic schemas. The definitions of these abstractions are guided by the observation that in the configurations of the programs we consider there are some repeated patterns for which it is sufficient to remember their number of repetitions precisely up to some fixed bound. If the number of repetitions is higher than the bound, we abstract it to an arbitrary value. The abstraction schemas we define are refinable in the sense that they define infinite sequences of abstraction mappings with increasing precision. Therefore, our verification approach is based on computing abstractions of the sets of reachable configurations, and on refining the abstractions when spurious counterexamples are detected.

These techniques allow us to *fully automatically compute* safe overapproximations of the state space of programs with 1-selector-linked dynamic data structures from whose elements the non-pointer fields are abstracted away. In this way, we can automatically check many important safety properties related to a correct use of dynamically allocated memory—absence of null pointer dereferences, working with uninitialized pointers, memory leakage (i.e. checking that there does not arise any unfreed and unaccessible garbage), etc. Furthermore, we can automatically handle the cases where a finite number of elements of the considered dynamic data structures are allowed to carry other than pointer fields. Using this fact and a simple technique which we propose for describing the desired input/output configurations, we can then automatically verify various properties relating the input and output of the considered programs (e.g., that the output of a list reversing procedure is really exactly the reverse of the input list, etc.). Finally, we show how the techniques can be applied to dealing with linked dynamic data structures whose elements contain any data fields of finite type too.

We have implemented the proposed techniques in a prototype tool and tried it out on a number of procedures manipulating classical singly-linked lists as well as cyclic lists. The results are very encouraging and show the applicability of our approach.

Related Work. Out of the work on verification of programs with dynamic linked data structures published in the literature, the two approaches that are probably the closest to our approach are the ones related to the tools Pale [15] and TVLA [16].

Pale (or more precisely its version for singly-linked structures) based on [8] uses a similar encoding of configurations as the one we propose in the following. The possibility of sharing parts of the lists is, however, not considered there. Moreover, there is no translation of the programs to transducers for manipulating sets of configurations in the Pale approach. The effect of the program is expressed by manipulating a logical description of the configurations, and automata come into play only when deciding the resulting WS1S formulae in Mona [12]. The approach of Pale is not as automatic as ours—only loop-free code can be handled automatically; if there are loops in the code to be checked, the user has to manually provide their invariants. We adopt a different methodology based on abstract symbolic reachability analysis which can also be used to automatically generate invariants.

TVLA is based on abstractions of the arising pointer structures described in a 3-valued logic [16]. The approach is more automatic than the one of Pale, but still the user may be required to provide some instrumentation predicates (or simulation invariants in the later approach of [7]) to make the abstraction sufficiently precise. The recent work [13] presents the first steps towards automatically obtaining the necessary instrumentation predicates by an analysis of spurious counterexamples. Moreover, up to very recently, TVLA had difficulties with cyclic structures that were resolved in a way [14] which like our approach exploits the observation that singly-linked structures exhibit some internal repeated structural patterns.

Both Pale and TVLA are extended to handle structures with more than a single next pointer. We are preparing such an extension of our approach based on tree (or more general) automata too.

Finally, representations of linked memory structures based on automata were used in [9, 5, 18, 4] too. In [18] following the earlier work of [9, 5], the special problem of may-alias analysis is primarily considered and a different symbolic representation of memory structures is used—it is based on tuples of automata (one for each pointer variable) and alias relations (using constraints). In [4], an alias logic with a Hoare-like proof system is introduced. There, one memory structure is represented as a collection of automata whereas we represent a set of memory structures with one automaton.

Outline. The rest of the paper is organised as follows. In Section 2 we introduce basic concepts about automata and transducers. In Section 3 we describe our encoding of pointer programs with automata and transducer. Then, we give our verification method in Section 4. Finally, we describe our experimental results in Section 5 and conclude.

2 Automata and Transducers

A *finite-state automaton* is a 5-tuple $A = (Q, \Sigma, \delta, q_{init}, F)$ where Q is a finite set of states, Σ a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ a set of labelled transitions, $q_{init} \in Q$ the initial state and $F \subseteq Q$ a set of final states.

The transition relation $\rightarrow \subseteq Q \times \Sigma^* \times Q$ of A is defined as the smallest relation satisfying: (1) $\forall q \in Q : q \xrightarrow{\varepsilon} q$, (2) if $(q, a, q') \in \delta$, then $q \xrightarrow{a} q'$, and (3) if $q \xrightarrow{w} q'$ and $q' \xrightarrow{a} q''$, then $q \xrightarrow{wa} q''$. The (regular) language recognised by A from a state $q \in Q$ is $L(A, q) = \{w \;:\; \exists q' \in F.\, q \xrightarrow{w} q'\}$. The language of A is $L(A) = L(A, q_{init})$. We suppose here that automata are manipulated in their canonical (i.e. minimal deterministic) form.

A *finite-state transducer* over Σ is a 5-tuple $\tau = (Q, \Sigma_\varepsilon \times \Sigma_\varepsilon, \delta, q_{init}, F)$ where Q is a finite set of states, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $\delta \subseteq Q \times \Sigma_\varepsilon \times \Sigma_\varepsilon \times Q$ is a set of transitions, $q_{init} \in Q$ is the initial state, and $F \subseteq Q$ a set of final states. The transition relation $\rightarrow \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ is defined as the smallest relation satisfying: (1) $q \xrightarrow{\varepsilon,\varepsilon} q$ for every $q \in Q$, (2) if $(q, a, b, q') \in \delta$, then $q \xrightarrow{a,b} q'$ and (3) if $q \xrightarrow{u,v} q'$ and $q' \xrightarrow{a,b} q''$, then $q \xrightarrow{ua,vb} q''$. A transducer τ defines a (regular) relation $R_\tau = \{(u, v) \;:\; \exists q' \in F.\, q_{init} \xrightarrow{u,v} q'\}$.

Given a language $L \subseteq \Sigma^*$ and a relation $R \subseteq \Sigma^* \times \Sigma^*$, let $R(L)$ be the set $\{v \in \Sigma^* \;:\; \exists u \in L.\, (u, v) \in R\}$. Sometimes, we abuse the notation by identifying a transducer τ (resp. an automaton A) with the relation R_τ (resp. the language $L(A)$). For instance, we write $\tau(A)$ to denote $R_\tau(L(A))$.

Let $id \subseteq \Sigma^* \times \Sigma^*$ be the identity relation and $\circ$ the composition of relations. Given a transducer τ, let $\tau^0 = id$, $\tau^{i+1} = \tau \circ \tau^i$, and let $\tau^* = \cup_{i=0}^{\infty} \tau^i$ be the reflexive-transitive closure of τ.

3 From Programs to Transducers

In this section, we describe the translation we propose for automatic verification of sequential, non-recursive programs with 1-selector-linked dynamic data structures in the framework of regular model checking. Our translation is general enough to cover *any* program of this kind (not containing pointer arithmetics and not explicitly covering the possibly necessary abstraction of non-pointer data).

We first describe how to encode as words the so-called program *stores*, i.e. the dynamic memory part of program configurations containing dynamically allocated memory cells linked by pointers. This encoding is similar to the one used in [8], but extended with the possibility of lists sharing their parts. Then, we propose an encoding of the standard C pointer operations (apart from pointer arithmetics) in the form of transducers. This is different from [8] where operations are encoded by changing a logical description of the configurations. Some of the pointer operations cannot be translated directly to a single transducer, therefore we propose to simulate their effect by computing a limit of a repeated application of certain simple auxiliary transducers.

In the following, we will use as a running example the following procedure reversing a list l. We suppose the data fields normally present in the elements of the data type `List` to be abstracted away and just the next-pointer fields to be preserved.

```
List x,y,l;
l1: y = null;
l2: while (l != null) {  // i.e. if (l!=null) goto l3; else goto l7;
l3:      y = l->next;
l4:      l->next = x;
l5:      x = l;
l6:      l = y; }        // i.e. l = y; goto l2;
l7: l = x;
l8: // end of program
```

3.1 Encoding Stores as Words

Basically, a store is encoded as the concatenation of several words (separated by a special symbol), each of them representing a list of elements. Successive elements of these lists are given from the left to the right, with positions of pointer variables marked by special symbols. We suppose for the moment that list elements contain no data—later we show that adding data of a finite type is not a problem. We also suppose for the beginning that the store does not contain cycles nor shared parts (i.e. no two different next-pointers point to the same list element). To encode such stores as words, we use the following alphabet Σ: For every pointer variable x used in the program at hand, we have $x \in \Sigma$, and Σ further contains the letters $|$ to separate lists (and some special parts of the configurations), $/$ to separate list elements (i.e. $/$ represents a next-pointer), # to express that a next-pointer points to null, and ! to denote that the next-pointer value is undefined.

Then, we can encode a store without sharing and cycles as a sequence of three parts separated by the symbol $|$ as follows:

- The first part contains a sequence of pointer variables whose values are undefined. In order not to have to consider all their possible orderings, we fix in advance a certain ordering on Σ that is respected here as well as in similar situations below.
- The second part contains pointer variables pointing to null.
- Finally, the third part contains the list sequences separated again by the symbol $|$. Each list sequence is encoded as follows: Every list element is represented by a (possibly empty) sequence of pointer variables pointing to it, lists elements are separated by the symbol $/$, and lists end either with the symbol # (null) or ! (undefined).

For example, the word $x\ y \mid \mid l\ /\ /\ \# \mid$ encodes a possible initial configuration of the list reversion example: x and y are undefined, no variable points to null, and l points to a list with two elements.

Now, regular expressions (or alternatively finite-state automata) can be used to describe sets of stores. For instance, the regular expression $(x\ y \mid \mid l\ /^+\ \# \mid) + (x\ y \mid l \mid)$ encodes all possible initial stores for our list reversion example.

Notice that in our encoding, we do not allow garbage (parts of the memory not accessible from pointer variables). As soon as an operation creates garbage, an error is

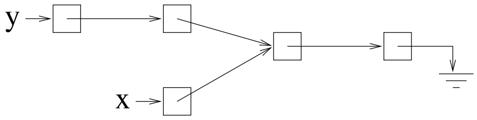

Fig. 1. A store with sharing

reported. In fact, such a situation corresponds to a memory leak in C (in Java, on the other hand, we can always perform "garbage collection" and remove the garbage).

Remark: Clearly, pointer variables appear exactly once in every word. The separator | and the symbols # and ! appear a bounded number of times since we do not consider stores with garbage. Finally, the symbol / can appear an unbounded number of times.

Lists with Sharing and/or Loops. To encode sharing of parts of lists as, for example, in Figure 1, we extend the alphabet Σ by a finite set of pairs of markers (m_f, m_t, n_f, n_t, etc.). A "from" marker X_f may be used after a next-pointer sign / to indicate that the given next-pointer points to an element marked by X_t (the corresponding "to" marker). Then, e.g., the word $|\;|\;x\;/\;m_f\;|\;y\;/\;/\;n_f\;|\;n_t\;m_t\;/\;/\;\#\;|$ encodes the store of Figure 1.

As one can easily see, the above store could be encoded in several other ways too (for instance, as $|\;|\;x\;/\;n_t\;/\;/\;\#\;|\;y\;/\;/\;n_f\;|$). Although we partially normalize the encoding by imposing a certain ordering on the symbols that are attached to the same memory location, we do not define a canonical representative of the store. However, our experimental results (see Section 5) show that this is not an obstacle to a practical applicability of our method. Furthermore, using a canonical form would complicate the encoding of program statements.

Notice also that markers allow us to encode circular lists (as, e.g., $|\;|\;x\;n_t\;/\;/\;n_f\;|$ corresponding to a circular list of two elements pointed to by x).

It is not difficult to see that given a store with k pointer variables encoded with more than k pairs of markers, one can encode the same store with at most k markers provided that no garbage is allowed: If a "to" marker is at the beginning of a sequence of cells that is not accessible without using markers, we can put these sequence directly in place of the corresponding "from" marker and save one pair of markers. For example, the store $|\;|\;x\;/\;m_f\;|\;y\;/\;/\;n_f\;|\;n_t\;m_t\;/\;/\;\#\;|$ of Figure 1 can be described with one pair of markers as $|\;|\;x\;/\;n_t\;/\;/\;\#\;|\;y\;/\;/\;n_f\;|$ or also as $|\;|\;x\;/\;m_f\;|\;y\;/\;/\;m_t\;/\;/\;\#|$.

Typically, the number of markers that is really needed is even smaller than k as we will demonstrate in our experiments.

3.2 Encoding Program Statements as Transducers

We now describe our encoding of program statements as transducers. We consider non-recursive C programs without pointer arithmetics. Initially, we also suppose all non-pointer data manipulations to be abstracted away—we briefly return to handling them later. Such programs may easily be pre-processed to contain only statements of the form `pointer_assignment; goto l;` or `if (pointer_test) goto l1; else goto l2;`. Moreover, by introducing auxiliary variables, we can eliminate multiple pointer dereferences of the form `x->next->next` and consider single dereferences only.

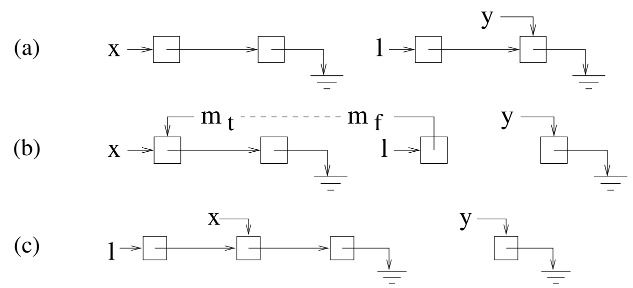

Fig. 2. An example store, the store after the statement `l->next=x`, and after a rearrangement

To encode full configurations of the considered programs, we extend the encoding of stores by adding a letter for the line of the program the control is currently at (followed by a separator $|$). Moreover, for the needs of our verification procedure, we add a single letter indicating the so-called computation mode. The mode is either n (normal), e (error—a null pointer dereference or working with an undefined pointer has been detected), s (shifting, used later for implementing the pointer manipulation statements that cannot be implemented as a single transducer), and u (unknown result that arises when an insufficient number of markers is used). For instance, the initial configurations of the list reversion example are then $(n\, l_1 \mid x\, y \mid \mid l\, /^+\, \# \mid) + (n\, l_1 \mid x\, y \mid l \mid)$.

Conditional jumps based on tests like `x==null` or `x==y` are now quite easy to encode. The transducer just checks whether `x` is in the null section or in the same section as `y` (taking $/$ and $|$ as section separators), and according to this changes the letter encoding the current line. If `x` or `y` is in the undefined section, we go to the error mode. Similarly, assignments of the form `x=null` or `x=y` are easy to handle—`x` is deleted from its current position (using an x,ε transition) and put to the section of y (using an ε,x transition).

A slightly more involved case is the one of tests based on the `x->next` construct and the one of the `y=x->next` assignment. Apart from generating an error when `x` is undefined or null, one has to consider the successor of `x`, which may involve going from a "from" marker to the appropriate "to" marker. However, as the number of markers is finite, the transducer can easily remember from which marker to which it is going and skip the part of the configuration between these markers.

Adding/removing markers. The most difficult case is then the one of the `l->next=x` assignment. The transducer first tries to commit the operation by using a pair of unused markers (say m_f/m_t) out of the in advance chosen set of marker pairs (an unused marker pair is one that does not appear in the current configuration word). Then, behind the section of `l`, the transducer puts m_f, and marks the section of `x` by m_t. For instance, in the list reversion procedure, $n\, l_4 \mid \mid \mid x\, /\, /\, \# \mid l\, /\, y\, /\, \# \mid$ is transformed via `l->next=x` into $n\, l_5 \mid \mid \mid m_t\, x\, /\, /\, \# \mid l\, /\, m_f \mid y\, /\, \# \mid$ as shown in Figure 2 (a), (b).

However, there may not be any unused markers left. In such a case, the transducer tries to reclaim some by re-arranging the configuration. This can be done by moving some sequence of cells that starts with a "to" marker directly into the place of the corresponding "from" marker (provided these markers do not constitute a loop). As explained in Section 3.1, this is always possible provided the chosen number of pairs of markers is sufficiently big (more than the number of pointer variables). For example,

$n\ l_5 \mid \mid \mid m_t\ x\ /\ /\ \#\ \mid l\ /\ m_f \mid y\ /\ \# \mid$ can be re-arranged to $n\ l_5 \mid \mid \mid l\ /\ x\ /\ /\ \# \mid y\ /\ \# \mid$ as sketched in Figure 2 (c).

The above operation, however, cannot be encoded as a single transducer as it may require an unbounded sequence (such as the list after x in our example) to be shifted to another place, and a finite-state transducer is incapable of remembering such sequences. To circumvent this problem, we use a very simple transducer τ which does one step of the shifting—i.e. it shifts a single element of the sequence by deleting it from its current location and re-producing it at its required location. The desired result is then the limit $\tau^*(Conf)$ where $Conf$ is a regular set of configurations on which the operation is applied. The limit (or an upper approximation of it) is computed using our abstract reachability analysis techniques. In order not to mix half-shifted sequences with the ready-to-use ones, the shifting is done in a special computation mode when no other operations are possible. [3]

If some marker has to be eliminated but this cannot be done, we go to the u mode and stop the computation. Such a situation cannot happen when we use as many markers as pointer variables. Nevertheless, it may happen when the user tries to use a smaller number of them with the aim of reducing the verification time (which is often, but not always possible). If one does not want to use markers at all, the two operations of introducing and eliminating a pair of markers (including shifting) are done at once.

Finally, the remaining `malloc(x)` and `free(x)` operations are again easy to encode. The `malloc(x)` operation introduces a sequence of elements with a single element, pointed to by `x`, and with an undefined successor. The `free(x)` operation removes an element, makes `x` and all its aliases undefined, and possibly makes undefined the next-pointer originally leading to `x`.

Adding Data Values to List Elements. The encoding can easily be extended to handle list elements containing data of a finite type. Their values are added into Σ and then every memory cell encoded as a sequence surrounded by $/$ and/or $\mid$ contains not only the pointers (markers) pointing to it, but also the appropriate data value. The tests and assignments on `*x` may then easily be added by testing whether the appropriate data letter is in the section of `x` or changing the data letter in this section.

4 Automatic Verification Techniques

We introduce in this section infinite-state verification techniques based on the regular model checking framework. These techniques combine automata-based reachability analysis with abstraction techniques. We concentrate in this work on the verification of safety properties. In the context of regular model checking, given a transducer τ modelling some infinite-state system, an initial set of configurations $Init$, and a set of bad configurations Bad, the safety verification problem consists in deciding whether

$$\tau^*(Init) \cap Bad = \emptyset \qquad (1)$$

Since the problem is undecidable in general (the transitions of any Turing machine can be straightforwardly encoded by a finite-state transducer), we adopt an approach

[3] Shifting could be implemented as an atomic, special purpose (and rather complex) operation directly on the automata too.

based on computing abstractions of the set $\tau^*(Init)$ and refining these abstractions when spurious counterexamples are detected.

4.1 Abstract Regular Model Checking

A *language abstraction* is a mapping $\alpha : 2^{\Sigma^*} \to 2^{\Sigma^*}$ such that $\forall L \in 2^{\Sigma^*}.\, L \subseteq \alpha(L)$. An abstraction α' *refines* (or is a *refinement* of) an abstraction α if $\forall L \in 2^{\Sigma^*}.\, \alpha'(L) \subseteq \alpha(L)$. An abstraction α is *finite-range* if the set $\{L \in 2^{\Sigma^*} \,:\, \exists L' \in 2^{\Sigma^*}.\, \alpha(L') = L\}$ is finite. We say that an abstraction mapping is *regular* if it can be defined by a finite-state transducer.

Given a transducer τ and a language abstraction α, let τ_α be the mapping such that $\forall L \in 2^{\Sigma^*}.\, \tau_\alpha(L) = \alpha(\tau(L))$.

The first step of our approach is to define a language abstraction α and compute the set $\tau_\alpha^*(Init)$. Clearly, if α is a finite-range abstraction, the iterative computation of $\tau_\alpha^*(Init)$ as $\tau_\alpha(Init) \cup \tau_\alpha^2(Init) \cup \ldots$ eventually terminates. By definition of α, the obtained set $\tau_\alpha^*(Init)$ is an overapproximation of $\tau^*(Init)$, and therefore, if $\tau_\alpha^*(Init) \cap Bad = \emptyset$, the problem (1) has a *positive answer*. Otherwise, the answer to the problem (1) is not necessarily negative since during the computation of $\tau_\alpha^*(Init)$, the abstraction α may introduce extra behaviours leading to Bad.

Let us examine this case. Assume that $\tau_\alpha^*(Init) \cap Bad \neq \emptyset$, which means that there is a symbolic path:

$$Init,\ \tau_\alpha(Init),\ \tau_\alpha^2(Init), \cdots \tau_\alpha^{n-1}(Init),\ \tau_\alpha^n(Init) \tag{2}$$

such that $\tau_\alpha^n(Init) \cap Bad \neq \emptyset$. We analyze this path by computing the sets $X_n = \tau_\alpha^n(Init) \cap Bad$, and for every $k \geq 0$, $X_k = \tau_\alpha^k(Init) \cap \tau^{-1}(X_{k+1})$. Two cases may occur: (*i*) either $X_0 = Init \cap (\tau^{-1})^n(X_n) \neq \emptyset$, which means that the problem (1) has a *negative answer*, or (*ii*) there is a $k \geq 0$ such that $X_k = \emptyset$, and this means that the symbolic path (2) is actually a *spurious counterexample* due to the fact that α is too coarse. In this last situation, we need to refine α and iterate the procedure. Therefore, our approach is based on the definition of abstraction schemas allowing to compute families of (automatically) refinable abstractions.

In a previous work [2], we have proposed *representation-oriented* abstractions which consist in defining finite-range abstractions on automata (used as symbolic representation structures for sets of configurations). The general principle of these abstractions is to collapse automata according to some given equivalence relation on their states, regardless of the kind of the represented configurations or the analyzed system.

In this work, we adopt an alternative approach by considering *configuration-oriented* abstractions which are defined on configurations. This approach allows us to define abstraction techniques which are more adapted to the application domain we are considering here. In the next subsections, we propose generic schemas for defining families of refinable configuration-oriented abstractions. Instances of these schemas have been implemented in a prototype tool and used in several experiments (see Section 5).

4.2 Piecewise 0-k counter abstractions

The idea behind the first abstraction schema we introduce is to abstract each word by considering some finite decomposition of it, and by applying 0-k counter abstraction (which looses the information about the ordering between symbols and only keeps track of their numbers of occurrences up to k) to each piece of the word in this decomposition. Formally, for $w \in \Sigma^*$, let $dec(w) = (a_1, w_1, a_2, w_2, \cdots, a_n, w_n)$ such that $w = a_1 w_1 a_2 w_2 \cdots a_n w_n$, $\forall i, j \in \{1, \ldots, n\}$. $a_i \in \Sigma$ and $a_i \neq a_j$, and $\forall i \in \{1, \ldots, n\}$. $w_i \in \{a_1, \ldots, a_i\}^*$. Intuitively, $dec(w)$ corresponds to the unique decomposition of w according to the first occurrences in w of each of the symbols in Σ.

Given a word w and a symbol a, let $|w|_a$ denote the number of occurrences of a in w. Given $k \in \mathbb{N}^{>0}$, we define a mapping α_k from words to languages such that for every $w \in \Sigma^*$, if $dec(w) = (a_1, w_1, a_2, w_2, \cdots, a_n, w_n)$, then $\alpha_k(w) = a_1 L_1 a_2 L_2 \cdots a_n L_n$ where $\forall i \in \{1, \ldots, n\}$. $L_i = \{u \in \{a_1, \ldots, a_i\}^* \ : \ \forall j \in \{1, \ldots, i\}. \ |w_i|_{a_j} < k$ and $|u|_{a_j} = |w_i|_{a_j}$, or $|w_i|_{a_j} \geq k$ and $|u|_{a_j} \geq k\}$. We generalize α_k from words to languages in the straightforward way in order to obtain a language abstraction. We can easily prove that:

Proposition 1. *For every $k \geq 0$, α_k is regular and effectively representable by a finite-state transducer.*

Clearly, for every given alphabet Σ, the set of possible 0-k abstractions is finite, and therefore, the number of piecewise 0-k abstractions is also finite since they consist in concatenations of a bounded number of symbols and 0-k abstractions.

Proposition 2. *For every $k \in \mathbb{N}$, the abstraction α_k is finite-range.*

In fact, below, we consider a generalization of the above schema obtained as follows. We allow that decompositions may be computed according to the first occurrences of *only a subset* of the alphabet, called *decomposition symbols*. Furthermore, we allow that the abstraction does not concern some symbols, called *strong symbols*, i.e. all their occurrences are preserved at their original positions. Typically, strong symbols are those which are known to have a bounded number of occurrences in all considered words. For instance, in words corresponding to encodings of program configurations, strong symbols correspond to markers, separators, and pointer variables which are known to have either a fixed or a bounded number of occurrences in all configurations.

Formally, let $\Sigma_1, \Sigma_2 \subseteq \Sigma$ be two sets of symbols such that $\Sigma_1 \cap \Sigma_2 = \emptyset$, where Σ_1 is the set of decomposition symbols and Σ_2 is the set of strong symbols. (Notice that there may be symbols which are neither in Σ_1 nor in Σ_2.) Then, given $w \in \Sigma^*$, we define $dec(w)$ to be the decomposition $(a_1, w_1, a_2, w_2, \cdots, a_n, w_n)$ such that (1) $w = a_1 w_1 a_2 w_2 \cdots a_n w_n$, (2) $\forall i \in \{1, \ldots, n\}$. $a_i \in \Sigma_1 \cup \Sigma_2$ and, $a_i \in \Sigma_1 \Rightarrow |a_1 a_2 \cdots a_n|_{a_i} = 1$, and (3) $\forall i \in \{1, \ldots, n\}$. $w_i \in (\{a_1, \ldots, a_i\} \setminus \Sigma_2)^*$. Then, for each given k, the abstraction α_k is defined precisely as before.

The previous proposition still holds if the number of occurrences of each strong symbol is bounded. Let us call p-Σ_2-bounded language any set of words L such that $\forall w \in L. \ \forall a \in \Sigma_2. \ |w|_a \leq p$.

Proposition 3. *For every bound $p \geq 0$, and for every $k \in \mathbb{N}$, the abstraction α_k is finite-range when it is applied to p-Σ_2-bounded languages.*

As for the abstraction refinement issue, it is easy to see that the abstraction schema introduced above defines a family of refinable abstractions.

Proposition 4. *For every p-Σ_2-bounded language L, and for every $k \geq 0$, we have $\alpha_{k+1}(L) \subseteq \alpha_k(L)$. Moreover, if L is infinite, then $\alpha_{k+1}(L) \subsetneq \alpha_k(L)$.*

4.3 Closure abstractions

We introduce hereafter another family of regular abstractions. Now, the idea is to apply iteratively extrapolation rules which may be seen as rewriting rules replacing words of the form u^k, for some given word u and positive integer k, by the language $u^k u^*$.

Let $u \in \Sigma^*$ and let $k \in \mathbb{N}^{>0}$ be a strictly positive integer. A relation $R \subseteq \Sigma^* \times \Sigma^*$ is an *extrapolation rule* wrt. the pair (u,k) if $R = \{(w,w') \in \Sigma^* \times \Sigma^* \;:\; w = u_1 u^k u_2 \text{ and } w' \in u_1 u^k u^* u_2\}$. An *extrapolation system* is a finite union of extrapolation rules.

Clearly, for every language L, we have $L \subseteq R(L)$ (i.e. R defines a language abstraction). In fact, we are interested in abstractions which are the result of *iterating extrapolation systems*. Therefore, let us define a *closure abstraction* as the reflexive-transitive closure R^* of some extrapolation system R.

It is easy to see that every extrapolation system corresponds to a regular relation (i.e. definable by a finite-state transducer). The question is whether closure abstractions of regular languages are still regular and effectively computable. In the general case, the answer is not known. However, we provide a reasonable condition on extrapolation systems which guarantees the effective regularity of closure abstractions.

First of all, we can prove that if we consider a single extrapolation rule, the corresponding closure abstraction if effectively computable.

Lemma 1. *For every extrapolation rule R, and for every regular language L, the set $R^*(L)$ is regular and effectively constructible.*

Proof. Let A be an automaton recognizing L. Let B be an automaton recognizing $u^k u^*$, and let q_i (resp. q_f) be its initial (resp. final) state. Then, for every pair of states (q,q') of A that are related by u^k, we extend A by a unique copy of B and two ε transitions $q \xrightarrow{\varepsilon} q_i$ and $q_f \xrightarrow{\varepsilon} q'$ (which can then be removed by the classical algorithms). □

Now, let $R = R_1 \cup \cdots \cup R_n$ be an extrapolation system where each of the R_i's is an extrapolation rule wrt. a pair $(u_i,k_i) \in \Sigma^* \times \mathbb{N}^{>0}$. Our idea is to define a condition on R such that the computation of $R^*(L)$ can be done for every language L by computing sequentially closures wrt. each of the extrapolation rules R_i in some ordering. Let $\prec \subseteq \Sigma^* \times \Sigma^*$ be the smallest relation such that for every $u,v \in \Sigma^*$, $u \prec v$ if (1) u is not a factor of v (i.e. u does not appear as a subword of v), and (2) u cannot be written as $w_1 v^p w_2$ for any $p \in \mathbb{N}$ and two words w_1, w_2 such that w_1 is a suffix of v and w_2 is a prefix of v. We can prove the following lemma which says that if $u \prec v$, then u can never appear in any power of v.

Lemma 2. $\forall u,v \in \Sigma^*$, *if* $u \prec v$ *then* $\forall p \geq 0.\ \forall w_1,w_2 \in \Sigma^*.\ v^p \neq w_1 u w_2$

Proof. Immediate from the definition of $\prec$: The fact that u can appear in some power of v implies that one of the two conditions defining $u \prec v$ is false. □

We say that the extrapolation system R is *serialisable* if the reflexive closure of the relation $\prec$ (i.e. $\prec \cup id$) defines a partial ordering on the set $\{u_1^{k_1}, \ldots, u_n^{k_n}\}$ (i.e. $\prec$ is antisymmetric and transitive on this set).

Lemma 3. *Let R be a serialisable extrapolation system and let $R_{i_1} R_{i_2} \ldots R_{i_n}$ be a total ordering of the rules of R which is compatible with $\prec$. Then, $R^* = R^*_{i_n} \circ R^*_{i_{n-1}} \cdots \circ R^*_{i_1}$.*

Proof. Follows from Lemma 2: Closing by some R_{i_j} never creates new rewriting contexts for any of the R_{i_ℓ} with $\ell < j$. □

From the two lemmas 1 and 3 we deduce the following fact:

Theorem 1. *For every serialisable extrapolation system R and for every regular language L, the set $R^*(L)$ is regular and effectively constructible.*

Closure abstractions (even serialisable ones) are not finite-range in general. To see this, consider the infinite family of (finite) languages $L_n = (ab)^n$ for $n \geq 0$ and the extrapolation rule R with $U = \{a\}$ and $k = 1$. Then, the images of the languages above form an infinite family of languages defined by $R^*(L_n) = (a^+b)^n$ for every $n \geq 0$.

Therefore, in the verification framework described in Section 4.1, the use of a closure abstraction α does not guarantee the termination of the computation $\tau_\alpha^*(Init)$. However, as our experiments show (see Section 5) the extrapolation principle used in these abstractions is powerful enough to force termination in many practical cases while preserving the necessary accuracy of the analysis of complex properties.

Let us finally mention that the abstraction schema introduced above defines a family of refinable abstractions.

Proposition 5. *Let R be an extrapolation system wrt. a set of pairs $\{(u_1,k_1),...,(u_n,k_n)\}$, let $k'_1,\ldots,k'_n$ be integers such that $\forall i.\ k'_i \geq k_i$, and let S be the extrapolation system wrt. $\{(u_1,k'_1),\ldots,(u_n,k'_n)\}$. Then, for every language L, we have $S^*(L) \subseteq R^*(L)$. Moreover, if L is infinite, then $S^*(L) \subsetneq R^*(L)$.*

5 Applications and Experimental Results

We have experimented with a prototype implementation of our techniques on several procedures manipulating linked lists. We have implemented a prototype compiler translating programs into transducers as explained in Section 3. As shown in Table 1, we have considered procedures for reversing a list, inserting an element into a list at a given position, deleting an element of a list at a given position, merging two lists element-by-element, and the procedure of Bubblesort over a list. Let us note that although these procedures primarily work with simple linear lists, temporarily they may yield several lists sharing their tails or create circular links. Moreover, we have considered working directly with circular lists too, namely a procedure for reversing such lists and a procedure for removing a segment of a circular list (the motivating example of [14]).

As mentioned in Section 3, a store can have several encodings. Thus, to perform correctly the check $\tau_\alpha^*(Init) \cap Bad = \emptyset$, we require the set *Bad* to contain *all* possible encodings of bad stores. In all properties we consider below, this can be easily achieved.

5.1 Checking Consistency of Working with the Dynamic Memory

For all the examples, we have firstly checked a basic consistency property that consisted in checking that there is no null pointer dereference, no work with undefined pointers, no memory leak (i.e. there does not arise any undeleted and inaccessible garbage), and that the result is a single list pointed to by the appropriate variable. The specification of such a property for a given procedure is easy and can be derived automatically. For the list reversion example, the set of bad states can be specified using the below extended regular expression[4] where $V = x?\ y?$:

$$(((e+u)\ \Sigma^*) + (\Sigma\ l_8\ \Sigma^*))\ \&\ \neg(n\ l_8 \mid V \mid ((l\ V \mid) + (V \mid l\ V\ (/\ V)^*\ /\ \# \mid)))$$

The expression says that it is bad when we try to do a null pointer dereference or work with an undefined pointer value—this is recognized automatically in the transducers and signalized by the first letter of the resulting configuration set to e. If the first letter becomes u (for unknown), the program cannot be verified using the given number of markers and we have to add some. Finally, it is bad when we reach the final line l_8, and the result is not an empty list (represented by l behind #) nor a single list pointed to by l. We do not care about the values of x and y.

The above property of course holds for the correct versions of all the considered procedures. In such a case, our tool provides the user with a safe overapproximation of all the configurations reachable at every line. In this way, we, e.g., automatically obtain the following invariant of the loop of the list reversion procedure:

$$(nl_2 \mid y \mid lx \mid) + (nl_2 \mid y \mid x \mid l(/)^+\# \mid) + (nl_2 \mid \ \mid ly \mid x(/)^+\# \mid) + (nl_2 \mid \ \mid \ \mid x(/)^+\# \mid ly(/)^+\# \mid)$$

Roughly, this invariant says that the list is either empty, is pointed to from l, from x, or partially from x and partially from l.

To try out the ability of our techniques to generate counterexamples, we have also tried to examine a faulty version of the list reversion procedure where lines 4 and 5 were swapped. In this case, an error is reported and we are told that from a list with one element (i.e. from a configuration $n\ l_1 \mid x\ y \mid \mid l\ /\ \# \mid$), we can obtain a circular list (a configuration $n\ l_8 \mid y \mid m_t\ l\ x\ /\ m_f \mid$ where m_f and m_t represent the "from" and "to" versions of a marker m). The user can then also trace the program forwards from the initial configuration or backwards from the erroneous one.

5.2 Checking More Complex Properties

Further, we have tried to verify some more complex properties of the considered programs. Let us start, e.g., with the Bubblesort procedure. When checking just its basic consistency property, we have completely abstracted away the data values stored in the list and made all the conditional jumps fully nondeterministic. To check that the procedure really sorts, we used a technique inspired by [15]. We considered the values of the list elements to be abstracted to being either greater or less than or equal than their successors. The abstracted data values were represented by two special letters (gt and lte) associated with every list item. We supposed lte and gt to be distributed arbitrarily in the initial configurations. We then checked that the basic consistency property holds and, moreover, the result is a sorted list (i.e. a sequence of elements labelled—up to the last element—by lte).

In the case of the merge procedure, we let all elements of the first list be labelled as a elements in the initial configuration and all elements of the other list as b elements. Then we checked that the output list contains a regular mixture of a and b elements.

[4] We use "?" to denote zero or one occurrences and "&" to denote intersection.

Finally, for the list reversion and insertion and circular list reversion procedures, we did a fully precise verification of their effect. In the case of list reversion, this means that the output contains exactly the same elements as before, but in a reversed order. For the insertion procedure, the required property is that the output list is precisely the input list up to one new element added into the appropriate place.

To check the above rather strong property, we have proposed a simple, yet efficient technique. Let us explain it on the case of list reversion. In the initial configurations, we let the first and last element be labelled by special labels *bgn* and *end*. Next, we consider as initial all the configurations that can arise from the original initial configurations by attaching two further labels—namely fst and *snd*—to an arbitrary pair of successive elements. The labels are invisible for the unmodified program—they stay attached to their initial elements. Then, to check the desired property, it suffices that every reachable final configuration starts with *end*, ends with *bgn*, and contains a sequence snd/fst. This guarantees that no element can be dropped (then, there would be a way to obtain a configuration without some of the labels), no element can be added (either *end* would not be the first, *bgn* the last, or some snd/fst pair would get separated by another element), and the elements must be re-arranged in the given way (otherwise the required resulting ordering of the labels could be broken).

5.3 The Results of the Experiments

For each verification example, we applied one instance of the abstractions presented in Section 4. For checking the basic consistency properties, we used the piecewise 0-2 counter abstraction with no decomposition symbols ($\Sigma_1 = \emptyset$) and with strong symbols Σ_2 containing the pointer variables, the separator $|$, and the symbol #. Therefore, just the parts of words containing exclusively the $/$ symbols are abstracted. As remarked in Section 3.1, $/$ is the only symbol which can appear an unbounded number of times in lists without data. Therefore, our abstraction is finite range by Proposition 3. For the more complex properties, we used closure abstractions. The extrapolation rules we applied correspond to the loops one naturally expects to possibly arise in the considered structures (e.g., $(/a,2)$, $(/b,2)$, $(/a/b,2)$ for the list merge procedure)—providing such information seems to be easy in many practical situations. In all the cases, the abstractions we used are defined by serialisable extrapolation systems. Therefore, by Theorem 1, they are regular and effectively computable.

We tried out both verification over programs described by a single transducer as well as over programs described by a set of transducers (one per arc of the program control flow graph). Column T of Table 1 shows the running times obtained in the latter case. They were about 1.6 to 6 times better than in the former case. The computation times are presented for the minimum number of markers necessary not to run into the "do not known" result. In the case of inserting into a list, we, however, indicate that sometimes it may be advantageous to use more than a necessary number of markers, which is especially the case of loop-free procedures where it may completely eliminate the need for the complex operation of shifting. For every experiment, we also indicate the number of states and transitions of the biggest encountered automaton (or transducer).

We further made a comparison with the abstract regular model checking techniques based on automata abstraction introduced in [2]. We observed an equal performance on

Table 1. Some results of experimenting with classical and circular linked lists (obtained at 2.4GHz Intel Pentium 4 from an early prototype tool based on Yap Prolog and the FSA library)

Program	Markers	$\lvert M\rvert^{max}_{st.+tr.}$	T_{sec}	Program	Markers	$\lvert M\rvert^{max}_{st.+tr.}$	T_{sec}
Reverse, bas.cons.	0	51+105	0.3	Merge, bas.cons.	0	209+279	2.7
Reverse, full	0	281+369	4.2	Merge, corr.mix.	0	1080+1415	40.4
Faulty reverse	1	61+138	0.2	Bubblesort, bas.cons.	2	2095+2872	305
Insert, bas.cons.	0	81+102	0.5	Bubblesort, full	2	2339+2887	279
Insert, bas.cons.	2	165+577	0.15	Circ.list rev., bas.cons.	3	655+764	5.4
Insert, full	0	755+936	10.8	Circ.list reverse, full	3	2349+2822	50.6
Delete, bas.cons.	0	55+113	0.3	Circ.l. rem.seg., bas.c.	2	116+291	1.0

the faulty reverse example, but on the other examples, the new techniques were about 2.9 to 88 times better (not taking into account the Bubblesort example and checking of the correct mixture property for the list merge example where we stopped the tool based on [2] after 2000 seconds).

We believe that the verification times obtained from our prototype are very encouraging. Some of the verification times that can be found in the literature for similar verification experiments (especially the ones obtained from Pale) are lower but that is partly due to an incomparable degree of automation (especially in Pale where a significant amount of user intervention is needed) and partly due to the fact that our tool is just an early Prolog-based prototype. We expect much better times from a more solid implementation of our tool, which we are now working on.

6 Conclusion

We have proposed a new approach to automatic verification of programs with dynamic linked structures based on a combination of automata-based symbolic reachability analysis with abstraction techniques.

Our approach applies to C-like sequential programs with 1-selector linked structures, for which it allows to verify automatically (safety) properties concerning their data structures. The same techniques can also be used for automatic invariant generation for these programs. Notice that our approach is not restricted to C programs but can be adapted to other languages with similar operations on linked structures too.

The techniques we define are based on simple abstractions of regular sets of configurations which, on one hand, are abstract enough to force termination in many practical cases and, on the other hand, are accurate enough to handle complex properties of the considered data structures. The experimental results are quite encouraging and show the applicability of our approach at least to particular pointer-intensive library routines.

The techniques we propose in this paper are defined in a general way which makes them not restricted to the application domain we consider here. In fact, they can be used as efficient acceleration techniques in the generic framework of regular model checking for the verification of various classes of infinite-state systems as well.

A certain deficiency of the closure abstraction technique as presented above is the need to manually provide the extrapolation rules when non-pointer data fields are not abstracted away. However, very recently, we have proposed a heuristic for automatically deriving such rules based on on-the-fly monitoring of non-looping sequences of states in the encountered automata and on trying to divide them to a given number of equal subsequences, which can then be used as a basis for extrapolation. This heuristic was successful in all the considered examples with a similar time and space efficiency as presented above (the verification times being sometimes worse but sometimes even better). A proper theoretical as well practical investigation of this technique is a part of our future work.

For the future, we plan an extension of our framework to the case of more general linked data structures using representations based on more general classes of automata.

References

1. P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Algorithmic Improvements in Regular Model Checking. In *Proc. of CAV'03*, volume 2725 of *LNCS*. Springer, 2003.
2. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.
3. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.
4. M. Bozga, R. Iosif, and Y. Lakhnech. Storeless Semantics and Alias Logic. In *Proc. of PEPM'03*. ACM Press, 2003.
5. A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting. In *Proc. of PLDI'94*. ACM Press, 1994.
6. P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. In *Proc. of the 6th Infinity Workshop*, 2004.
7. N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. Verification via Structure Simulation. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.
8. J.L. Jensen, M.E. Jørgensen, N. Klarlund, and M.I. Schwartzbach. Automatic Verification of Pointer Programs Using Monadic Second-order Logic. In *Proc. of PLDI '97*, 1997.
9. H.B.M. Jonkers. Abstract Storage Structures. In *Algorithmic Languages*. IFIP, 1981.
10. B. Jonsson and M. Nilsson. Transitive Closures of Regular Relations for Verifying Infinite-State Systems. In *Proc. of TACAS'00*, volume 1785 of *LNCS*. Springer, 2000.
11. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. *Theoretical Computer Science*, 256(1–2), 2001.
12. N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, Denmark, 2001.
13. A. Loginov, T. Reps, and M. Sagiv. Abstraction Refinement for 3-Valued-Logic Analysis. Technical Report 1504, Computer Science Dept., University of Wisconsin, USA, 2004.
14. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proc. of VMCAI'05*. Springer, 2005.
15. A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI '01*, 2001. Also in SIGPLAN Notices 36(5) (May 2001).
16. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *TOPLAS*, 24(3), 2002.
17. T. Touili. Widening Techniques for Regular Model Checking. *ENTCS*, 50, 2001.
18. A. Venet. Automatic Analysis of Pointer Aliasing for Untyped Programs. *Science of Computer Programming*, 35(2), 1999.
19. P. Wolper and B. Boigelot. Verifying Systems with Infinite but Regular State Spaces. In *Proc. of CAV'98*, volume 1427, 1998.