Who is Pointing When to Whom?

On the Automated Verification of Linked List Structures

Dino Distefano*, Joost-Pieter Katoen, and Arend Rensink

Department of Computer Science, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract. This paper introduces an extension of linear temporal logic that allows to express properties about systems that are composed of entities (like objects) that can refer to each other via pointers. Our logic is focused on specifying properties about the dynamic evolution (such as creation, adaptation, and removal) of such pointer structures. The semantics is based on automata on infinite words, extended with appropriate means to model evolving pointer structures in an abstract manner. A tableau-based model-checking algorithm is proposed to automatically verify these automata against formulae in our logic.

1 Introduction

Pointers are references to memory cells. Programming with pointers is an error-prone activity with potential pitfalls such as dereferencing null pointers and the creation of memory leaks. Unwanted side-effects may occur due to aliasing where apparently unaffected variables are modified by changing a shared memory cell – the so-called "complexity of pointer swing". The analysis of pointer programs has been a topic of continuous research interest since the early seventies [3, 7]. The purpose of this research is twofold: to assess the correctness of pointer programs, and to identify the potential values of pointers at compile time to allow more efficient memory management strategies and code optimization.

Properties of Pointer Programs. Alias analysis, i.e., checking whether pairs of pointers can be aliases, has received much attention initially (see, e.g., [6, 14]). [8] introduced and provided algorithms to check the class of so-called position-dependent alias properties, such as "the n-th cell of v's list is aliased to the m-th cell of list w". Recently, extensions of predicate calculus to reason about pointer programs have become *en vogue*: e.g., BI [12], separation logic [20], pointer assertion logic (PAL) [13], alias logic [2], local shape logic [19] and extensions of spatial logic [4]. These approaches are almost all focused on verifying pre- and postconditions in a Hoare-style manner.

Since our interest is in concurrent (object-oriented) programs and in expressing properties over dynamically evolving pointer structures, we use first-order

* Currently at Dept. of Computer Science, Queen Mary, University of London, UK.

K. Lodaya and M. Mahajan (Eds.): FSTTCS 2004, LNCS 3328, pp. 250–262, 2004.

linear-time *temporal logic* (LTL) as a basis and extend it with *pointer assertions* on single-reference structures, such as aliasing, as well as predicates to reason about the *birth* and *death* of cells (which provide a model for object references). The expressiveness of the resulting logic, called NTL (Navigation Temporal Logic), is similar to that of the recent Evolution Temporal Logic (ETL) [23]. Whereas ETL uses 3-valued logical structures as semantic models, we follow an automata-based approach: models of NTL are infinite runs that are accepted by Büchi automata where states are equipped with a representation of the heap. PAL contains similar pointer assertions as NTL (and goes beyond lists), but has neither primitives for the birth and death of entities nor temporal operators. Evolving heaps have been lately used to model mobile computations. In that view NTL combines both spatial and temporal features similar to the ambient logic introduced in [5].

Heap Abstraction. A major issue in analyzing pointer programs is the choice of an appropriate representation of the heap. As the number of memory cells for a program is not known a priori and in general is unpredictable, a concrete representation is inadequate. Analysis techniques for pointer programs therefore typically use abstract representations of heaps such as, e.g., location sets [22] (that only distinguish between single and multiple cells), k-limiting paths [14] (allowing up to k distinct cells for some fixed k), or summary nodes [21] in shape graphs. This paper uses an abstract representation of heaps that is tailored to *unbounded* linked list structures. The novelty of our abstraction is its *parameterization* in the pointer program as well as in the formula. Cells that represent up to M elements, where M is a formula-dependent constant, are exact whereas unbounded cells (akin to summary nodes) represent longer lists. The crux of our abstraction is that it guarantees each unbounded cell to be preceded by a chain of at least L exact cells, where L is a program-dependent constant. Parameters L and M depend on the longest pointer dereferencing in the program and formula, respectively. In contrast with the k-limiting approach, where an adequate general recipe to determine k is lacking, (minimal bounds on) the parameters L and M can be easily determined by a static analysis.

Pointer Program Analysis. Standard type-checking systems are not expressive enough to establish properties of pointers such as memory leaks and dereferencing null pointers. Instead, techniques for analyzing pointer programs are more powerful and include abstract interpretation [8], deduction techniques [2, 12, 13, 20], design by derivation à la Dijkstra [16], and shape analysis [21], or combinations of these techniques.

As our aim is to obtain a fully automated verification technique the approach in this paper is based on *model checking*. Our model-checking algorithm is a non-trivial extension of the tableau-based algorithm for LTL [15]. For given NTL-formula Φ, this algorithm is able to check whether the automaton-model of the concurrent pointer program at hand satisfies Φ. The algorithm, like the ETL approach [23], suffers from false negatives, i.e., a verification may wrongly conclude that the program refutes a formula. In such case, however, diagnostic information

can be provided (unlike ETL, and as for PAL [13]) that may be used for further analysis. Besides, by incrementing the parameters M and L, a more concrete model is obtained that is *guaranteed* to be a correct refinement of the (too coarse) abstract representation. This contrasts with the ETL approach where manually-provided instrumentation predicates are needed. Compared to the PAL approach which is fully automated for loop-free (sequential) programs, our technique is fully automated for concurrent pointer programs that may include loops.

Main Contributions. Summarizing, the main contributions of this paper are: *(i)* A first-order temporal logic that both contains pointer assertions as well as predicates referring to the birth or death of memory cells; *(ii)* An automaton-based model for pointer programs where states are abstract heap structures and transitions represent the dynamic evolvement of these heaps; the model deals finitely with unbounded allocations. *(iii)* A way of parameterizing the degree of "correctness" of abstract heap structures, on the basis of a straightforward static analysis of the program and formula at hand. On incrementing these parameters, refined heap structures are automatically obtained. *(iv)* A model-checking algorithm to check abstract representations of pointer programs against formulae in our logic.

The main advantage of our approach is that it is completely automated: given a program and a temporal logic property, the abstract automaton as well as the verification result for the property are determined completely algorithmically. Moreover, to our knowledge, this paper is the first to develop model-checking techniques for (possibly) unbounded evolving heaps of the kind described above[1].

Our current approach restricts to single outgoing pointers. This still allows us to consider many interesting structures such as acyclic, cyclic and unbounded lists (as in [16] and [8]), as well as hierarchies (by backpointers). Besides, several "resource managers" such as memory managers only work with lists [18]. Our abstract heap structures can also model mobile ambients (see [9]).

Details of the model checking algorithm and all proofs can be found in [10].

2 A Logic for Dynamic References

Syntax. Let LV be a countable set of logical variables ranged over by x, y, z, and Ent be a countable set of entities ranged over by e, e', e_1 etc. $\bot \notin Ent$ is used to represent "undefined"; we denote $E^{\bot} = E \cup \{\bot\}$ for arbitrary $E \subseteq Ent$. Navigation Temporal Logic (NTL) is a linear temporal logic with quantification of logical variables that range over entities, or may be undefined. The syntax of navigation expressions is defined by the grammar:

$$\alpha \ ::= \ nil \ \Big| \ x \ \Big| \ \alpha\!\uparrow$$

[1] In this respect, the recent paper [1] only introduces a symbolic representation for heaps intended for checking safety properties (but not liveness), and does not consider model checking algorithms.

where *nil* denotes the null reference, x denotes the entity (or nil) that is the value of x, and $\alpha\!\uparrow$ denotes the entity referred to by (the entity denoted by) α (if any). Let $x\!\uparrow^0 = x$ and $x\!\uparrow^{n+1} = (x\!\uparrow^n)\!\uparrow$ for natural n. The syntax of *NTL* is:

$$\Phi \;::=\; \alpha = \alpha \;\Big|\; \alpha\,\mathsf{new} \;\Big|\; \alpha \rightsquigarrow \alpha \;\Big|\; \Phi \wedge \Phi \;\Big|\; \neg\Phi \;\Big|\; \exists\, x.\, \Phi \;\Big|\; \mathsf{X}\,\Phi \;\Big|\; \Phi\,\mathsf{U}\,\Phi \;.$$

The basic proposition $\alpha\,\mathsf{new}$ states that the entity (referred to by) α is fresh, $\alpha = \beta$ states that α and β are aliases, and $\alpha \rightsquigarrow \beta$ expresses that (the entity denoted by) β is reachable from (the entity denoted by) α. The boolean connectives, quantification, and the linear temporal connectives X (next) and U (until) have the usual interpretation. We denote $\alpha \neq \beta$ for $\neg\,(\alpha = \beta)$, α *dead* for $\alpha = \mathit{nil}$, α *alive* for $\neg\,(\alpha$ *dead*$)$, $\alpha \not\rightsquigarrow \beta$ for $\neg\,(\alpha \rightsquigarrow \beta)$ and $\forall x.\, \Phi$ for $\neg\,(\exists\, x.\, \neg\Phi)$. The other boolean connectives and temporal operators $\Diamond$ (eventually) and $\Box$ (always) are standard. For example, $\Diamond(\exists x.\, x \neq v \;\wedge\; x \rightsquigarrow v \;\wedge\; v \rightsquigarrow x)$ expresses that eventually v will point to a non-empty cycle.

Semantics. Logical formulae are interpreted over infinite sequences of sets of entities that are equipped with information concerning the linking structure between these entities. Formally, an *allocation sequence* σ is an infinite sequence of pairs $(E_0, \mu_0)(E_1, \mu_1)(E_2, \mu_2)\ldots$ where for all $i \geqslant 0$, $E_i \subseteq \mathit{Ent}$ and $\mu_i : E_i^{\perp} \to E_i^{\perp}$ such that $\mu_i(\perp) = \perp$; μ_i encodes the pointer structure of E_i. Let $\theta : \mathit{LV} \rightharpoonup \mathit{Ent}^{\perp}$ be a partial valuation of logical variables. The semantics of navigation expression α is given by:

$$\begin{aligned}
&[\![\, \mathit{nil} \,]\!]_{\mu,\theta} = \perp \\
&[\![\, x \,]\!]_{\mu,\theta} \;\; = \theta(x) \text{ if } \theta(x) \neq \perp, \text{ and } \perp \text{ otherwise} \\
&[\![\, \alpha\!\uparrow \,]\!]_{\mu,\theta} = \mu\,([\![\, \alpha \,]\!]_{\mu,\theta})
\end{aligned}$$

For a given allocation sequence σ, E_i^{σ} and μ_i^{σ} denote the set of entities, respectively the pointer structure, in the i-th state of σ. The semantics of *NTL*-formulae is defined by satisfaction relation $\sigma, N, \theta \models \Phi$ where σ is an allocation sequence, $N \subseteq E_0^{\sigma}$ is the set of entities that are initially new, and θ is a valuation of the free variables in Φ. Let N_i^{σ} denote the set of new entities in state i, i.e., $N_0^{\sigma} = N$ and $N_{i+1}^{\sigma} = E_{i+1}^{\sigma} \backslash E_i^{\sigma}$, and let θ_i^{σ} denote the valuation at state i, where $\theta_i^{\sigma}(x) = \theta(x)$ if $\theta(x) \in E_k^{\sigma}$ for all $k \leqslant i$, and is $\perp$ otherwise. The latter condition prevents that contradictions like $\exists x.\mathsf{X}\,(x$ *dead* $\Rightarrow \mathsf{X}\,x$ *alive*$)$ are satisfiable. Note that once a logical variable is mapped to an entity, this association remains valid along σ until the entity is deallocated. The satisfaction relation $\models$ is defined as follows:

$$\begin{aligned}
&\sigma, N, \theta \models \alpha\,\mathsf{new} \quad \text{iff} \;\; [\![\, \alpha \,]\!]_{\mu_0^{\sigma},\theta} \in N \\
&\sigma, N, \theta \models \alpha = \beta \quad \text{iff} \;\; [\![\, \alpha \,]\!]_{\mu_0^{\sigma},\theta} = [\![\, \beta \,]\!]_{\mu_0^{\sigma},\theta} \\
&\sigma, N, \theta \models \alpha \rightsquigarrow \beta \quad \text{iff} \;\; \exists\, k \geqslant 0.\, [\![\, \alpha\!\uparrow^k \,]\!]_{\mu_0^{\sigma},\theta} = [\![\, \beta \,]\!]_{\mu_0^{\sigma},\theta} \\
&\sigma, N, \theta \models \exists x.\Phi \quad \text{iff} \;\; \exists e \in E_0^{\sigma} : \sigma, N, \theta\{\, e/x \,\} \models \Phi \\
&\sigma, N, \theta \models \neg\Phi \quad \text{iff} \;\; \sigma, N, \theta \not\models \Phi \\
&\sigma, N, \theta \models \Phi \vee \Psi \quad \text{iff} \;\; \text{either } \sigma, N, \theta \models \Phi \text{ or } \sigma, N, \theta \models \Psi \\
&\sigma, N, \theta \models \mathsf{X}\,\phi \quad \text{iff} \;\; \sigma^1, N_1^{\sigma}, \theta_1^{\sigma} \models \Phi \\
&\sigma, N, \theta \models \Phi\,\mathsf{U}\,\Psi \quad \text{iff} \;\; \exists i.\, (\sigma^i, N_i^{\sigma}, \theta_i^{\sigma} \models \Psi \text{ and } \forall j < i.\, \sigma^j, N_j^{\sigma}, \theta_j^{\sigma} \models \Phi).
\end{aligned}$$

Here, $\theta\{\, e/x \,\}$ is defined as usual, i.e., $\theta\{\, e/x \,\}(x) = e$ and $\theta\{\, e/x \,\}(y) = \theta(y)$ for $y \neq x$. Note that the proposition $\alpha \rightsquigarrow \beta$ is satisfied if $[\![\, \beta \,]\!] = \bot$ and $[\![\, \alpha \,]\!]$ can reach some entity with an undefined outgoing reference.

Program Variables. To enable the specification of properties over entities pointed to by program variables (rather than just logical ones), we introduce for each program variable v_i a logical variable x_{v_i}. This variable always points to a distinguished entity e_{v_i} which exists in every state. For convenience in *NTL*-formulae let v_i denote $x_{v_i}\!\uparrow$ and let $\exists\, x.\, \Phi$ abbreviate $\exists\, x.\, (x \neq x_{v_1} \wedge \ldots \wedge x \neq x_{v_n}) \;\Rightarrow\; \Phi$.

Example 1. Consider the following list-reversal program (see, e.g., [2, 20, 21]):

$$\mathsf{decl}\ v, w, t : \quad w := \mathit{nil}; \mathsf{while}\ (v \neq \mathit{nil})\ \mathsf{do}\ \ t := w; w := v; v := v\!\uparrow; w\!\uparrow := t\ \ \mathsf{od}$$

Properties of interest of this program include, for instance: "v and w always point to distinct lists (heap non-interference)": $\Box(\forall x.\, v \rightsquigarrow x \;\Rightarrow\; w \not\rightsquigarrow x)$. "$v$'s list will be (and remains to be) reversed" [2]: $\forall x.\, \forall y.\, \big((v \rightsquigarrow x \wedge x\!\uparrow = y) \;\Rightarrow\; \Diamond\Box(y\!\uparrow = x)\big)$. "None of the elements in v's list will ever be deleted": $\forall x.\, (v \rightsquigarrow x \;\Rightarrow\; \Box x\ \mathit{alive})$.

Example 2. The following program consists of two processes that concurrently produce and add entities to the tail tl of a buffer, respectively remove and consume them from the head hd of that buffer:

$$\begin{aligned}\mathsf{decl}\ hd, tl, t : \ &\big(\mathsf{new}(tl); hd := tl; \mathsf{while}\ (\mathit{true})\ \mathsf{do}\ \ \mathsf{new}(tl\!\uparrow); tl := tl\!\uparrow\ \ \mathsf{od}\\ &\quad \parallel\ \mathsf{while}\ (\mathit{true})\ \mathsf{do}\ \ \mathsf{if}\ (hd \neq tl)\ \mathsf{then}\ t := hd; hd := hd\!\uparrow; \mathsf{del}(t)\ \mathsf{fi}\ \mathsf{od}\big)\end{aligned}$$

For navigation expression α, $\mathsf{new}(\alpha)$ creates (i.e., allocates) a new entity that will be referred to by the expression α. The old value of α is lost. Thus, if α is the only pointer to entity e, say, then after the execution of $\mathsf{new}(\alpha)$, e is automatically garbage collected together with the entities that are only reachable from e. $\mathsf{del}(\alpha)$ destroys (i.e., deallocates) the entity associated to α, so that α and every pointer referring to it becomes undefined. Some example properties: "Every element in the buffer is eventually consumed": $\Box(hd \neq tl \;\Rightarrow\; \exists x.\, (x = hd \;\wedge\; \Diamond x\ \mathit{dead}))$. "The tail is never deleted or disconnected from the head": $\Box(tl\ \mathit{alive} \wedge hd \rightsquigarrow tl)$.

3 Abstracting Linked List Structures

The most obvious way to model pointer structures is to represent each entity and each pointer individually. For most programs, like, e.g., the producer/consumer program, this will give rise to infinite representations. To obtain more abstract (and compact) views of pointer structures, in this paper chains of entities will be aggregated and represented by one (or more) entities. We consider the abstraction of *pure chains* (and not of arbitrary graphs) in order to be able to keep the "topology" of pointer structures invariant in a more straightforward manner.

[2] If one is interested in only checking whether v's list is reversed at the end of the program, program locations can be added and referred to in the standard way.

Pure Chains. Let $\prec$ be the binary relation on entities (excluding $\bot$) representing μ, i.e., $e \prec e'$ iff $\mu(e) = e'$. A sequence $e_1, \ldots, e_k$ is a chain (of length k) if $e_i \prec e_{i+1}$, for $0 < i < k$. The non-empty set E of entities is a chain of length $|E|$ iff there exists a bijection $f : \{1, \ldots, k\} \to E$ such that $f(1), \ldots, f(k)$ is a chain; let $\mathit{first}(E) = f(1)$ and $\mathit{last}(E) = f(k)$. E is a *pure chain* if $|\{ e' \mid e' \prec e \}| = 1$ for all $e \in f(2), f(3), \ldots, f(k)$ and f is unique (which may fail to be the case if the chain is a cycle). Note that chains consisting of a single element are trivially pure.

Abstracting Pure Chains. An abstract entity may represent a pure chain of "concrete" entities. The concrete representation of abstract entity e is indicated by its *cardinality* $\mathcal{C}(e) \in \mathbb{M} = \{1, \ldots, M\} \cup \{*\}$, for some fixed constant $M > 0$. Entity e for which $\mathcal{C}(e) = m \leqslant M$ represents a chain of m "concrete" entities; if $\mathcal{C}(e) = *$, e represents a chain that is longer than M. In the latter case, the entity is called *unbounded*. (Such entities are similar to summary nodes [21], with the specific property that they always abstract from pure chains.) The special cardinality function **1** yields one for each entity. The precision of the abstraction is improved on increasing M.

Configurations and Morphisms. States in our automata are triples $(E, \mu, \mathcal{C})$, called *configurations*. Configurations representing pure chains at different abstraction levels are related by morphisms, defined as follows. Let *Cnf* denote the set of all configurations ranged over by c and c', and $\mathcal{C}(\{e_1, \ldots, e_n\}) = \mathcal{C}(e_1) \oplus \ldots \oplus \mathcal{C}(e_n)$ where $n \oplus m = n+m$ if $n+m \leqslant M$ and $*$ otherwise.

Definition 1. *For $c, c' \in$ Cnf, surjective function $h : E \to E'$ is a* morphism *if:*

1. *for all $e \in E'$, $h^{-1}(e)$ is a pure chain and $\mathcal{C}'(e) = \mathcal{C}(h^{-1}(e))$*
2. *$e \prec' e' \;\Rightarrow\; \mathit{last}(h^{-1}(e)) \prec \mathit{first}(h^{-1}(e'))$*
3. *$e \prec e' \;\Rightarrow\; h(e) \preceq' h(e')$ where $\preceq'$ denotes the reflexive closure of $\prec'$.*

According to the first condition only pure chains may be abstracted by a single entity while keeping the cardinalities invariant. The last two conditions enforce the preservation of the pointer structure under h. Intuitively speaking, by means of a morphism the abstract shape of the pointer dependencies represented by the two related configurations is maintained. The identity function *id* is a morphism and morphisms are closed under composition. Configurations c and c' are isomorphic, denoted $c \cong c'$, iff there exist morphisms from c to c' and from c' to c such that their composition is *id*.

4 An Automaton-Based Model for Pointer Evolution

Evolving Pointer Structures. Morphisms relate configurations that model the pointer structure at distinct abstraction levels. They do not model the dynamic evolution of such linking structures. To reflect the execution of pointer-manipulating statements, such as either the creation or deletion of entities (e.g., `new` in Java and `delete` in C++), or the change of pointers by assignments (e.g., $x = x\!\uparrow\!\uparrow$), we use *reallocations*.

Definition 2. *For $c, c' \in Cnf$, $\lambda : (E^{\perp} \times E'^{\perp}) \to \mathbb{M}$ is a* reallocation *if:*

1. *(a)* $\mathcal{C}(e) = \bigoplus \lambda(e, e')$ *and (b)* $\mathcal{C}'(e') = \bigoplus \lambda(e, e')$
2. *(a) for all* $e \in E$, $|\{\, e' \mid \lambda(e, e') = * \,\}| \leqslant 1$ *and (b)* $\{\, e' \mid \lambda(\perp, e') = * \,\} = \varnothing$
3. *(a) for all* $e \in E$, $\{\, e' \mid \lambda(e, e') \neq 0 \,\}$ *and (b) for all* $e' \in E'$, $\{\, e \mid \lambda(e, e') \neq 0 \,\}$ *are chains.*

We write $c \overset{\lambda}{\rightsquigarrow} c'$ *if there is a reallocation (named* λ*) from* c *to* c'*.*

The special entity $\perp$ is used to model birth and death: $\lambda(\perp, e) \neq 0$ denotes the birth of (some instances of) e whereas $\lambda(e, \perp) \neq 0$ denotes the death of (some instances of) e. Intuitively speaking, reallocation λ redistributes cardinalities on E to E' such that (1a) the total cardinality allocated by λ to $e \in E$ equals $\mathcal{C}(e)$ and (1b) the total cardinality assigned to $e' \in E'$ equals $\mathcal{C}'(e')$. Moreover, (2a) for each entity e unbounded cardinalities (i.e., equal to $*$) are assigned only once (according to (1b) to an unbounded entity in E'), and (2b) no unbounded entities can be born. The last condition is self-explanatory. Note that the identity function *id* is a reallocation. The concept of reallocation can be considered as a generalisation of the idea of identity change as, for instance, present in history-dependent automata [17]: besides the possible change of the abstract identity of concrete entities, it allows for the evolution of pointer structures. Reallocations allow "extraction" of concrete entities from abstract entities by a redistribution of cardinalities between entities. Extraction is analogous to *materialisation* [21]. Reallocations ensure that entities that are born cannot be reallocated from any other entity. Moreover, entities that die can only be reallocated to $\perp$.

Relating Abstract and Concrete Evolutions. As a next step we relate transitions between abstract representations of pointer structures to transitions between their corresponding concrete representations. To that end, "abstract" reallocations are related to "concrete" ones. These are called *concretions*.

Definition 3. *Let* $c \overset{\lambda}{\rightsquigarrow} c'$ *and* $\widehat{c} \overset{\widehat{\lambda}}{\rightsquigarrow} \widehat{c}'$ *with* $\mathcal{C}_{\widehat{c}} = \mathcal{C}_{\widehat{c}'} = \mathbf{1}$*.* $\widehat{\lambda}$ *is a* concretion *of* λ*, denoted* $\widehat{\lambda} \rhd \lambda$*, iff there exist* h *and* h' *such that:*

1. h *is a morphism between* $\widehat{c}$ *and* c*, and* h' *is a morphism between* $\widehat{c}'$ *and* c'
2. $\lambda(e, e') = \bigoplus \{\, \widehat{\lambda}(\widehat{e}, \widehat{e}') \mid (h(\widehat{e}), h'(\widehat{e}')) = (e, e') \,\}$
3. $h(e) = h(e') \;\vee\; (h' \circ \widehat{\lambda})(e) = (h' \circ \widehat{\lambda})(e')$ *implies* $e \prec_{\widehat{c}} e' \Leftrightarrow \widehat{\lambda}(e) \prec_{\widehat{c}'} \widehat{\lambda}(e')$
4. $(\mathcal{C}_{\widehat{c}} \circ h')(e) = * \;\Rightarrow\; e \in cod(\widehat{\lambda})$*, the co-domain of* $\widehat{\lambda}$*.*

The first condition states that the concrete source-configuration $\widehat{c}$ and its abstract source c are related by a morphism, and the same applies to their target configurations $\widehat{c}'$ and c'. (Stated differently, reallocations and morphisms commute in this case.) The second condition requires the multiplicity of λ and $\widehat{\lambda}$ to correspond, while the third condition forbids the change of order (according to $\prec$) between concrete entities and their abstract counterparts (unlike reallocations). Hence, the order of entities in a chain should remain identical. The last condition says that entities that are mapped onto unbounded ones in the target states are not fresh. Due to the third condition all concrete entities represented by an abstract entity enjoy a common fate: either all of them "survive" the reallocation or all die.

Automaton-Based Model. In order to model the dynamic evolution of programs manipulating (abstract) linked lists, we use a generalisation of Büchi automata (extending [11]) where each state is a configuration and transitions exist between states iff these states can be related by means of a reallocation reflecting the possible change in the pointer structure.

Definition 4. *A* high-level allocation Büchi automaton *(HABA) H is a tuple $\langle X, C, \rightarrow, I, \mathcal{F}\rangle$ with:*

- $X \subseteq LV$*, a finite set of logical variables;*
- $C \subseteq Cnf$*, a set of configurations (also called* states*);*
- $\rightarrow \subseteq C \times (Ent \times Ent \times \mathbb{M}) \times C$*, a* transition relation*, s.t.* $c \rightarrow_\lambda c' \Rightarrow c \overset{\lambda}{\rightsquigarrow} c'$*;*
- $I : C \rightharpoonup 2^{Ent} \times (X \rightharpoonup Ent)$*, an* initialisation *function such that for all c with $I(c) = (N, \theta)$ we have $N \subseteq E$ and $\theta : X \rightharpoonup E$.*
- $\mathcal{F} \subseteq 2^C$ *a set of sets of* accept *states.*

Note that initial new entities cannot be unbounded. HABA can be used to model the behaviour of pointer-manipulating programs at different levels of abstraction. In particular, when all entities in any state are concrete (i.e., $\mathcal{C}(e) = 1$ for all e), and states are related by the identity reallocation, a concrete automaton is obtained that is very close to the actual program behaviour.

Automata for Pointer-Manipulating Programs. As a start, we determine by means of a static analysis of the program p, the "longest" navigation expression that occurs in it and fix constant L such that $L > \max\{\, n \mid (v\uparrow)^n \text{ occurs in } p \,\}$. Besides the formula-dependent constant M, the program-dependent constant L can be used to tune the precision of the symbolic representation, i.e., by increasing L the model becomes less abstract. Unbounded entities (i.e., those with cardinality $*$) will be exploited in the semantics to keep the model finite. The basic intuition of our symbolic semantics is that unbounded entities should always be preceded by a chain of at least L concrete entities. This principle allows us to precisely determine the concrete entity that is referred to by any assignment, new and del-statement. As assignments may yield unsafe configurations (due to program variables that are "shifted" too close to an unbounded entity), these statements require some special treatment (see [10]).

Folded Allocation Sequences. In *NTL*-formulae, entities can only be addressed through logical variables, and logical variables can only be compared in the same state. These observations allow a mapping of entities from one state in an allocation sequence onto entities in its next state, as long as this preserves the conditions of being a reallocation. A *folded allocation sequence* is an infinite alternating sequence $(E_0, \mu_0, \mathbf{1}_0)\lambda_0(E_1, \mu_1, \mathbf{1}_1)\lambda_1 \cdots$, where λ_i is a reallocation from $(E_i, \mu_i, \mathbf{1}_i)$ to $(E_{i+1}, \mu_{i+1}, \mathbf{1}_{i+1})$ for $i \geqslant 0$. Due to the unitary cardinality functions, λ_i associates at most one entity in E_{i+1} to an entity in E_i. We write λ_i^σ for the reallocation function of σ in state i, and we define $N_0^\sigma = N$, and $N_{i+1}^\sigma = E_{i+1}^\sigma \backslash cod(\lambda_i^\sigma)$. Similarly, $\theta_0^\sigma = \theta$ and $\theta_{i+1}^\sigma = \lambda_i^\sigma \circ \theta_i^\sigma$ where $\lambda \circ \theta_i(x)$ equals e if $\theta_i(x) \neq \perp$ and $\lambda(\theta_i(x), e) = 1$, and $\perp$ otherwise. Using these adapted

definitions of N and θ, a semantics of *NTL* can be defined in terms of folded allocation sequences that is equivalent to $\models$ (see [10]). Runs of our symbolic HABA automata "generate" folded allocation sequences in the following way:

Definition 5. *HABA-run* $q_0\lambda_0 q_1\lambda_1 \cdots$ *generates an allocation triple* (σ, N, θ) *where* $\sigma = c_0^\sigma \lambda_0^\sigma c_1^\sigma \lambda_1^\sigma \cdots$ *is a folded allocation sequence, if there exists a family of morphisms* h_i *(called a generator) from* c_i^σ *to* c_{q_i} *such that, for all* $i \geqslant 0$*:*

$$\lambda_i^\sigma \rhd \lambda_i \text{ (via } h_i \text{ and } h_{i+1}\text{), and } I(q_0) = (N', h_0 \circ \theta) \text{ where } N = h_0^{-1}(N').$$

We adopt the generalised Büchi acceptance condition, i.e, $c_0 c_1 c_2 \cdots$ is a *run* of HABA H if $c_i \rightharpoonup c_{i+1}$ for all $i \geqslant 0$ and $|\{\, i \mid c_i \in F \,\}| = \omega$ for all $F \in \mathcal{F}$. Let *runs*(H) denote the set of runs of H. Then $\mathcal{L}(H) = \{\, (\sigma, N, \theta) \mid \exists \rho \in \textit{runs}(H).\ \rho \text{ generates } (\sigma, N, \theta) \,\}$.

Relating the Concrete and Symbolic Model. A given HABA abstracts a *set* of concrete automata. We formally define this by first defining an implementation relation over HABA and then using the correspondence of concrete automata to a certain class of HABA. We say that a given HABA abstracts another one if there exists a so-called simulation relation (denoted $\lesssim$) between their state sets.

Definition 6. *Let* H *and* H' *be two HABAs such that* $\mathcal{C}(e) = 1$ *for all* e *in* H*.* H' abstracts H*, denoted* $H \sqsubseteq H'$*, iff there exists a* simulation relation $\lesssim \subseteq C \times (\mathit{Ent} \rightharpoonup \mathit{Ent}) \times C'$ *between their state sets such that:*

1. $c_1 \lesssim_h c_1'$ *implies that* h *is a morphism between* c_1 *and* c_1'*;*
2. $c_1 \lesssim_h c_1'$ *with* $c_1 \rightharpoonup_\lambda c_2$ *implies* $c_1' \rightharpoonup_{\lambda'} c_2'$ *for some* λ' *and* c_2' *such that* $c_2 \lesssim_{h'} c_2'$ *and* $\lambda \rhd \lambda'$ *via* h *and* h'*;*
3. $c \in \mathit{dom}(I)$ *implies* $I'(c') = (N, h \circ \theta)$ *for some* $c' \in C'$ *and* h *such that* $c \lesssim_h c'$ *and* $I(c) = (h^{-1}(N), \theta)$*;*
4. *there exists a bijection* $\psi : \mathcal{F} \to \mathcal{F}'$ *such that for all* $F \in \mathcal{F}$ *and* $c \in F$*,* $c \lesssim_h c'$ *for some* $c' \in \psi(F)$ *and* h*.*

$c \lesssim_h c'$ denotes that c' simulates c, according to a given morphism h. This implies (*1*) that c' is an abstraction of the pointer structure in c (due to the morphism h), and (*2*) that every λ-transition of c is mimicked by a λ'-transition of c' such that $\lambda' \rhd \lambda$ and the resulting target states are again in the simulation relation. H is abstracted by H' if there is a simulation relation between their states such that (*3*) initial states and (*4*) accept conditions correspond.

Let $H \models \Phi$ if for all $(\sigma, N, \theta) \in \mathcal{L}(H)$ we have $\sigma, N, \theta \models \Phi$, where $\models$ is the satisfaction relation for NTL defined on folded allocation sequences.

Theorem 1. *For* $H \sqsubseteq H'$*:* $\mathcal{L}(H) \subseteq \mathcal{L}(H')$ *and* $(H' \models \Phi \;\Rightarrow\; H \models \Phi)$.

From this result it follows that all positive verification results on the (typically finite) abstraction H' carry over to the (mostly infinite) concrete automaton H.

Note that false negatives may occur as the refutation of a formula by H' does not necessarily has to imply that H refutes it as well[3].

Example 3. Consider the following NTL-formulae: *tl alive* $\Rightarrow \Box$(*tl alive*) and $\Box$(*hd alive* $\Rightarrow$ *hd* $\rightsquigarrow$ *tl*). It turns out that both formulae are valid in the abstract HABA ($L = M = 1$) modelling the producer/consumer program. By Theorem 1 we conclude that they are valid also on the infinite corresponding concrete model.

5 Model Checking

The Parameters M and L. The precision of automaton H is ruled by two parameters: L, which controls the distance between entities before they are collected into unbounded entities, and M, which controls the information we have about unbounded entities. L is used in the generation of models from programs; it is no longer of importance in the model checking stage. M is a formula-dependent constant exceeding $\sum_{x \in \Phi} \max\{\, i \mid (x\uparrow)^i \text{ occurs in } \Phi \,\}$ *for the formula Φ to check.* This may mean that the model H at hand is not (yet) suitable for checking a given formula Φ, namely if M for that model does not meet this lower bound. In that case we have to *stretch* the model. Fortunately, we can stretch a given model without loss of information (but with loss of compactness, and hence increase of complexity of the model checking). In fact, in [10] we define an operation $H \Uparrow \widehat{M}$, which stretches H so that in the resulting model the corresponding constant is $\widehat{M}$, and we have the following:

Theorem 2. *For all HABA H such that $\mathcal{C}(H) < \widehat{M}$: $\mathcal{L}(H) = \mathcal{L}(H \Uparrow \widehat{M})$.*

Here, $\mathcal{C}(H)$ is the maximal cardinality of some entity in H. The automaton $H \Uparrow \widehat{M}$ is a factor $\widehat{M} - M$ times as large as H.

The Tableau Graph. The next step is to construct a *tableau graph* $G_H(\Phi)$ for Φ from a given model H, assuming that stretching has been done, so M satisfies the given lower bound for Φ. $G_H(\Phi)$ enriches H, for each of its states q, with information about the collections of formulae relevant to the validity of Φ that possibly hold in q. These "relevant formulae" are essentially sub-formulae of Φ and their negations; they are collected into the so-called *closure* of Φ [15]. The states of $G_H(\Phi)$ are now so-called *atoms* (q, D) where q is a state of H and D a consistent and complete set of valuations of formulae from the closure of Φ on (the entities of) q. Consistency and completeness approximately mean that, for instance, if Ψ_1 is in the closure then exactly one of Ψ_1 and $\neg\Psi_1$ is "included in" D (i.e., D contains a valuation for it), and if $\Psi_1 \vee \Psi_2$ is in the closure then it

[3] In [11] where we only considered the birth and death of entities we obtained a stronger relationship between (a somewhat simpler variant of) the concrete and symbolic model. Here, the abstraction is less precise and permits the abstracted model to exhibit spurious behaviours that do not occur in the concrete model.

is "in" D iff Ψ_1 or Ψ_2 is "in" D, etc. For any q, the number of atoms on q is exponential in the size of the closure and in the number of entities in q.

A transition from (q, D) to (q', D') exists in the tableau graph $G_H(\Phi)$ if $q \rightarrow_\lambda q'$ in H and, moreover, to the valuation of each sub-formula $\mathsf{X}\Psi$ in D there exists a corresponding valuation of Ψ in D' — where the correspondence is defined modulo the reallocation λ. A *fulfilling path* in $G_H(\Phi)$ is then an infinite sequence of transitions, starting from an initial state, that also satisfies all the "until" sub-formulae $\Psi_1 \mathsf{U} \Psi_2$ in the atoms, in the sense that if a valuation of $\Psi_1 \mathsf{U} \Psi_2$ is in a given atom in the sequence, then a corresponding valuation of Ψ_2 (modulo a sequence of reallocations) occurs in a later atom.

Proposition 1. *$H \models \Phi$ iff there does not exist a fulfilling path in $G_H(\neg \Phi)$.*

Unfortunately, in contrast to the case of propositional logic (in [15]) and our own earlier results in the absence of pointers (in [11]), in the setting of this paper we have not found a decision procedure for the existence of a fulfilling path. In fact, the existence of a *self-fulfilling strongly connected sub-component* (SCS) of the tableau graph, which is the technique used in these other papers, gives only a necessary criterion for the existence of a fulfilling path. To be precise, if we use $Inf(\pi)$ to denote the set of atoms that occur infinitely often in an (arbitrary) infinite path π in $G_H(\Phi)$, then we have:

Proposition 2. *$Inf(\pi)$ is not a self-fulfilling SCS $\Rightarrow$ π is not a fulfilling path.*

Since the number of SCSs of any finite tableau graph is finite, and the property of self-fulfillment is decidable, this gives rise to a mechanical procedure for verifying the satisfiability of formulae.

Theorem 3. *$H \models \Phi$ can be verified mechanically for any finite HABA H.*

This, combined with Th. 1, implies that, for any concrete automaton A of which H is an abstraction, it is also possible to verify mechanically whether $A \models \Phi$. Note that this theorem leaves the possibility of *false negatives*, as usual in model checking in the presence of abstraction. This means that if the algorithm fails to show $H \models \Phi$ then it cannot be concluded that Φ is *not* satisfiable (by some run of H). However, since such a failure is always accompanied by a "prospective" fulfilling path of Φ, further analysis or testing may be used to come to a more precise conclusion.

6 Conclusions

Although our heap structures are less general than those used in shape analysis, our abstractions are less non-deterministic, and therefore, are potentially more exact. Experimental research is needed to validate this claim. Although *NTL* is essentially a first-order logic, it contains two second-order features: the reachability predicate $\alpha \rightsquigarrow \beta$ (which computes the transitive closure of pointers), and the freshness predicate $\alpha\,\mathsf{new}$. The latter is second-order because it essentially

expresses that the entity denoted by α did not occur in the *set* of entities existing in the directly preceding state. In fact it would be very useful to extend the latter to freshness with respect to an arbitrary previous state, for instance by introducing formulae $\sigma X.\Phi$ which bind X to the set of entities existing in the current state, and predicates $\alpha \in X$ which express that the entity denoted by α is in the set X. We conjecture that the results of this paper can be lifted to such an extension without essential changes.

References

1. S. Bardin, A. Finkel, and D. Nowak. Towards symbolic verification of programs handling pointers. In: *AVIS 2004*. ENTCS 2004 to appear.
2. M. Bozga, R. Iosif, and Y. Lakhnech. Storeless semantics and alias logic. In: *PEPM*, pp. 55–65. ACM Press, 2003.
3. R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence* **6**: 23–50, 1971.
4. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In: *ICALP*, LNCS 2380, pp. 597–610. Springer, 2002.
5. L. Cardelli and A.D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In: *POPL*, pp. 365–377. ACM Press, 2000.
6. D.R. Chase, M. Wegman and F. Zadeck. Analysis of pointers and structures. In *PLDI*, pp. 296–310. ACM Press, 1990.
7. S.A. Cook and D. Oppen. An assertion language for data structures. In: *POPL*, pp. 160–166. ACM Press, 1975.
8. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In: *PLDI*, pp. 230–241. ACM Press, 1994.
9. D. Distefano. On model checking the dynamics of object-based software: a foundational approach. PhD. Thesis, Univ. of Twente, 2003.
10. D. Distefano, A. Rensink and J.-P. Katoen. Who is pointing when to whom? CTIT Tech. Rep. 03-12, 2003.
11. D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In: *TCS*, pp. 435–447. Kluwer, 2002.
12. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, pp. 14–26, ACM Press, 2001.
13. J. Jensen, M. Jørgensen, M. Schwartzbach and N. Klarlund. Automatic verification of pointer programs using monadic second-order logic. In: *PLDI*, pp. 226–236. ACM Press, 1997.
14. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, Chapter 4, pp. 102-131, Prentice-Hall, 1981.
15. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In: *POPL*, pp. 97–107. ACM Press, 1985.
16. G. Nelson. Verifying reachability invariants of linked structures. In: *POPL*, pp. 38–47. ACM Press, 1983.
17. U. Montanari and M. Pistore. An introduction to history-dependent automata. *ENTCS* **10**, 1998.
18. P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. In: *POPL*, pp. 268–280. ACM Press, 2004.

19. A. Rensink. Canonical graph shapes. In: *ESOP*, LNCS 2986, pp. 401–415. Springer,2004.
20. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pp. 55–74. IEEE CS Press, 2002.
21. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, **20**(1): 1–50, 1998.
22. L. Séméria, K. Sato and G. de Micheli. Resolution of dynamic memory allocation and pointers for the behavioural synthesis from C. In *DATE*, pp. 312–319. ACM Press, 2000.
23. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In: *ESOP*, LNCS 2618, pp. 204–222. Springer, 2003.