Verifying Dynamic Pointer-Manipulating Threads

Thomas Noll and Stefan Rieger

RWTH Aachen University
Software Modeling and Verification Group
52056 Aachen, Germany
{noll,rieger}@cs.rwth-aachen.de

Abstract. We present a novel approach to the verification of concurrent pointer-manipulating programs with dynamic thread creation and memory allocation as well as destructive updates operating on arbitrary (possibly cyclic) singly-linked data structures. Correctness properties of such programs are expressed by combining a simple pointer logic for specifying heap properties with linear-time (LTL) operators for reasoning about system executions. To automatically solve the corresponding model-checking problem, which is undecidable in general, we abstract from non-interrupted sublists in the heap, resulting in a finite-state representation of the data space. We also show that the control flow of a concurrent program with unbounded thread creation can be characterized by a Petri net, making LTL model checking decidable (though not feasible in practice). In a second abstraction step we also derive a finite-state representation of the control flow, which then allows us to employ standard LTL model checking techniques.

1 Introduction

Techniques for the verification of elementary properties of pointer programs are highly desirable. Programming with pointers is error-prone with potential pitfalls such as dereferencing null pointers and the emergence of memory leaks. So far, the field of pointer analysis has primarily focused on sequential programs. But pointer programming becomes even more vulnerable in a concurrent setting where threads can be dynamically created, and where data structures such as linked lists are shared between several threads.

We present an approach to model checking concurrent programs that operate on singly-linked data structures. It stays within the realm of traditional (linear-time) model checking. This facilitates the usage of standard model checkers for validating temporal properties addressing absence of memory leaks, dereferencing of null pointers, dynamic creation of cells, and simple and position-dependent aliasing.

Our approach is illustrated by considering a simple concurrent programming language that besides the usual control structures offers primitives for thread creation, pointer manipulation, cell creation and destruction, and (guarded) atomic regions that allow to implement concurrency control constructs such as test-and-set primitives, semaphores, and monitors.

The operational semantics of our language is defined in a modular way. The *control-flow semantics* is given by a (finite) Petri net whose places correspond to the control locations of the program. The *heap semantics* is specified by transformation rules which describe the effect of executing single commands.

The combination of both yields a labeled transition system (modeled by a Petri net) which is generally infinite due to the unbounded creation of both control threads and heap cells. Its desirable properties are expressed in a first-order linear-time temporal logic (LTL) that is enriched with assertions on pointer structures such as reachability and freshness of cells, or pointer aliasing.

Since the model-checking problem is generally undecidable in this setting we introduce a first abstraction, which addresses the data space of the program. Our list abstraction exploits a variant of summary nodes [7] to obtain a finite representation of the heap and thus eliminates one potential source of undecidability. In fact, known results then allow us to conclude that the data abstract model-checking problem is decidable even though the underlying transition system is still infinite (see Thm. 5.7). However, its intractability forces us to apply a second abstraction step in which we also derive a finite-state representation of the control flow, which altogether yields a finite transition system. As a result, standard LTL model-checking algorithms can be employed. Both abstractions are obtained in a fully mechanized manner. Moreover they are sound in the sense that they over-approximate the concrete program behavior.

2 Related Work

Related work on the topic of analyzing pointer-manipulating programs can be classified into the following (often overlapping) categories, which mainly focus on sequential programming languages: *predicate abstraction* [1,8,23], *shape analysis* [2,26,27], *regular model checking* [3,5], *dataflow analysis* [21,33,34], *Hoare-style approaches* [6,18], and *separation logic* [22,24]. In summary, many of the characterizing features of our approach are already present in earlier papers: the restriction to singly-linked lists without data fields [1,3,11,16,19,20] which still allows to model many practical applications such as device drivers, the introduction of abstract entities which represent a potentially unbounded number of heap cells (called "summary nodes" in [7]), and the observation that, in this setting, the number of sharing points in heap structures is bounded by the number of program variables [1,4,20].

Pointer analysis in connection with concurrency is only considered in rather few places. Most publications concentrate on specific questions such as aliasing or escape analysis [25,28] or the analysis of safety properties [13,30], or particular applications such as concurrent garbage collection are studied [9,10,29]. To our knowledge, the only pointer logics allowing to specify liveness properties of concurrent systems are ETL [31] and NTL [11]. In contrast to these, however, we avoid the use of temporal operators inside quantification. In this way, involved mechanisms to keep track of the identities of individual heap nodes are not required.

Thus our approach is unique in that it supports concurrent programs with dynamic thread creation, memory allocation, and destructive updates operating on arbitrary (possibly cyclic) linked lists. Moreover it integrates both abstraction and model checking in a fully automated way and supports a linear-time logic in which both safety and liveness properties can be expressed, allowing to use standard LTL model checkers.

3 A List-Manipulating Programming Language

Given sets PV of program variables and $\mathcal{P}$ of thread names, a *dynamic list-manipulating program* (DLMP) π has the form ($v_i, v \in PV$ and $p_j, p \in \mathcal{P}$)

$$\pi = \textbf{var } v_1, ..., v_k;\ \textbf{proc main}(S_0);\ p_1(S_1); ...;\ p_l(S_l)$$

Here each S_i ($0 \leq i \leq l$) is of the form $s_{i1}; ...; s_{ir_i}$ with $s_{ij} \in$ CMD, where CMD is the set of the following commands:

PExp := PExp	pointer assignment	**new**(PExp)	object creation
if BExp **goto** n	conditional jump	**del**(PExp)	object destruction
goto n	unconditional jump	**spawn**(p)	spawn instance of thread p
atc(BExp)	guarded atomic region	**exit**	thread termination
end atc	end of atomic region		

Pointer expressions (PExp) comprise the special constant *nil* denoting an undefined pointer value, a program variable, or the (de)referencing of a program variable. Arbitrary dereferencing depths can be emulated using a sequence of atomic assignments. The *Boolean expressions* (BExp) are standard.

$$\text{PExp} ::= nil \mid v \mid *v \mid \&v \qquad \text{BExp} ::= \text{PExp} = \text{PExp} \mid \text{BExp} \wedge \text{BExp} \mid \neg\text{BExp}$$

Note that we do not allow nesting of atomic regions. In the following we assume for simplicity that π as above is globally given (if not mentioned otherwise).

Fig. 1 shows a DLM-program that simulates a simple server/worker scenario. The server creates new objects in an infinite loop and inserts them into a list. For each object a new worker thread is spawned deleting one object from the list when it is executed. Without imposing fairness constraints this may lead to an infinite number of both objects and threads.

Petri Nets. We use Petri nets to describe the operational semantics of DLMPs.

Definition 3.1. *A* Petri net *is a tuple* $\mathfrak{P} = (P, T, src, tgt, \ell, m_0)$ *where* P *is a set of* places*,* T *a set of transitions,* $src, tgt : T \to 2^P$ *associate each* transition *with its source and target places,* $\ell : T \to L$ *is a transition* labeling function*, and* $m_0 : P \to \mathbb{N}$ *the* initial marking*. A* state *of* $\mathfrak{P}$ *is a marking* $m : P \to \mathbb{N}$*. The set of all markings is denoted by* $Mark(\mathfrak{P})$*.*

Petri nets are high-level representations of (infinite) transition systems whose transitions are characterized by the *token game*. If in a marking m a transition t is enabled, i.e. $m(p) > 0$ for all $p \in src(t)$, and if its firing yields m', we write $m \rhd_t m'$. $m \rhd m'$ means that there exists $t \in T$ such that $m \rhd_t m'$.

Definition 3.2 (Run). *Let $\mathfrak{P} = (P, T, src, tgt, \ell, m_0)$ be a Petri net. A* run *of $\mathfrak{P}$ is a (possibly infinite) sequence of markings $\rho = m_0 m_1 m_2 ... \in Mark(\mathfrak{P})^\star \cup Mark(\mathfrak{P})^\omega$ such that $m_i \rhd m_{i+1}$. The set of all those runs is denoted by $Runs(\mathfrak{P})$.*

For $\rho = m_0 m_1 ... \in Runs(\mathfrak{P})$ let $|\rho| \in \mathbb{N} \cup \{\infty\}$ be the length of ρ. We write $\rho[k]$ to denote the suffix starting from the k-th marking, i.e., $m_k m_{k+1} ... \in Runs(P, T, src, tgt, \ell, m_k)$ which implies $\rho[k] = \varepsilon$ for $|\rho| \leq k$, and we set $\rho_i := m_i$.

Finally we call a Petri net k-*safe* if at no time any place holds more than k tokens and *bounded* if there exists a k for which it is k-safe. Clearly only bounded Petri nets can be represented by *finite* transition systems.

Concrete Heap Semantics. Defining the semantics of DLM-programs requires a formal model of the heap.

Definition 3.3. *A* heap configuration *is a tuple $H = (N, A, \mu, F)$ with a set of nodes $N \supseteq PV$, a set of abstract nodes $A \subseteq N \setminus PV$, a successor function $\mu : N \to N_{nil}$ (where $N_{nil} := N \cup \{nil\}$), and a set of flags $F \subseteq \text{Flags} := \{\text{err}, \text{leak}, \text{del}\} \cup \{\text{new}_n \mid n \in N\} \cup \{\text{spawn}_p \mid p \in \mathcal{P}\}$.*

$\boldsymbol{H}$ denotes the set of all heap configurations; $\boldsymbol{H_\emptyset} \subseteq \boldsymbol{H}$ the set of all concrete ones (i.e., those with $A = \emptyset$).

The nodes represent both the dynamic objects at runtime and the static program variables (which cannot be deleted). Edges, as formalized by the μ-function, encode the *points-to* information of a specific program state. The set A of abstract nodes will later be used for our heap abstraction technique (see Sct. 4) and will be empty throughout the current section. Finally the flags give special information about a state, e.g., whether a runtime error or memory leak occurred, a node was created or deleted, or a thread has been spawned.

```
var x, y;
proc main(
01   new(x);
02   spawn(server); )

server(
11   spawn(worker);
12   atc(tt);
13        y := x;
14        new(x);
15        *x := y;
16   end atc;
17   goto 11; )

worker(
21   atc(x ≠ nil);
22        y := x;
23        x := *x;
24        del(y);
25   end atc; )
```

Fig. 1. Server/Worker

To delete unreachable nodes that do not influence program semantics a garbage collection mapping denoted by $\downarrow$ is applied. Whenever it removes an unreachable node, it sets the leak flag to indicate a potential memory leak.

Definition 3.4. *Let $H = (N, \emptyset, \mu, F) \in \boldsymbol{H_\emptyset}$. The semantics of pointer expressions is given by the partial function $[\![\cdot]\!] : \text{PExp} \rightharpoonup N_{nil}$, defined as follows (where $\bot$ denotes the undefined value).*[1]

$$[\![nil]\!] := nil \qquad [\![v]\!] := \mu(v) \qquad [\![*v]\!] := \mu([\![v]\!]) \qquad [\![\&v]\!] := v$$

The semantics of Boolean expressions, $[\![\cdot]\!] : \text{BExp} \rightharpoonup \mathbb{B}$, is standard but strict, i.e., it becomes undefined if at least one subexpression is undefined.

[1] The definition implies $\mu(nil) = \bot$ and so $[\![\cdot]\!]$ can indeed yield undefined results.

The effect of executing a program statement is captured by a transition relation which associates the source and target heap configuration with the given statement and an indicator from the set $\{0, 1, \perp\}$. Here 1 denotes the normal execution of a statement or the selection of the then-branch of an **if**-command, 0 only occurs in the else-branch of **if**-statements, and $\perp$ represents the failure of a command (e.g., dereferencing a null-pointer).

Definition 3.5. *The* heap transformation relation, $\rightarrow_h \subseteq (\boldsymbol{H_\emptyset} \setminus \{H_{\text{err}}\}) \times \text{CMD} \times \{0, 1, \perp\} \times \boldsymbol{H_\emptyset}$*, is given as follows. Here* $H_{\text{err}} := (PV, \emptyset, \{v \mapsto nil \mid v \in PV\}, \{\text{err}\})$*,* $H = (N, A, \mu, F) \in \boldsymbol{H_\emptyset} \setminus \{H_{\text{err}}\}$ *with* $A = \emptyset$*, and* $f[x/y]$ *denotes a function update where* y *is the new value of* x*. (We only show some example rules.)*

$$\frac{[\![\alpha]\!] \neq \perp}{H, v := \alpha \xrightarrow{1}_h (N, A, \mu[v/[\![\alpha]\!]], \emptyset)\downarrow} \qquad \frac{[\![\alpha]\!] = \perp}{H, v := \alpha \xrightarrow{\perp}_h H_{\text{err}}}$$

$$\frac{}{H, \mathbf{new}(v) \xrightarrow{1}_h (N \uplus \{n\}, A, \mu[v/n], \{\text{new}_n\})\downarrow}$$

$$\frac{[\![\alpha]\!] \neq nil}{H, \mathbf{del}(\alpha) \xrightarrow{1}_h (N \setminus \{[\![\alpha]\!]\}, A, \mu[[\![\alpha]\!]/\perp, \mu^{-1}([\![\alpha]\!])/nil], \{\text{del}\})\downarrow}$$

$$\frac{[\![b]\!] \neq \perp}{H, \mathbf{if}\ b\ \mathbf{goto}\ n \xrightarrow{[\![b]\!]}_h (N, A, \mu, \emptyset)} \qquad \frac{[\![b]\!] = \perp}{H, \mathbf{if}\ b\ \mathbf{goto}\ n \xrightarrow{\perp}_h H_{\text{err}}}$$

$$\frac{[\![b]\!] = 1}{H, \mathbf{atc}(b) \xrightarrow{1}_h \hat{H}} \qquad \frac{[\![b]\!] = \perp}{H, \mathbf{atc}(b) \xrightarrow{\perp}_h H_{\text{err}}}$$

Note that the heap flags (except err) are only active in the configuration directly following the corresponding event.

As our final goal is to combine the heap and control-flow semantics, we now represent the heap transformation relation by a Petri net. The labels will later be used for synchronizing the two nets.

Definition 3.6. *The* concrete heap semantics *is the (infinite, 1-safe) Petri net* $\mathfrak{P}^h := (P, T, src, tgt, \ell, m_0)$ *with* $P \subseteq \boldsymbol{H_\emptyset}$, $T = \{(H, H', c, x) \mid H, c \xrightarrow{x}_h H')\}$, $src(H, H', c, x) = \{H\}$, $tgt(H, H', c, x) = \{H'\}$, $\ell(H, H', c, x) = (c, x)$, $m_0(H_0) = 1$ *for a given* $H_0 \in P$ *(typically the empty heap), and* $m_0(H) = 0$ *for* $H \neq H_0$*.*

Control-Flow Semantics. In the context of concurrency and dynamic threading it does not suffice to only consider the effects of certain statements on the heap; the control flow of the program is also crucial. It can again be modeled by a Petri net.

Definition 3.7. *The* control-flow semantics *of* π *is given by the Petri net* $\mathfrak{P}^c := (P, T, src, tgt, \ell, m_0)$ *with* $P = \{\text{lock}\} \cup \bigcup_{i=0}^{l} \bigcup_{j=1}^{r_i} \{ij\}$, $\ell : T \rightarrow \text{CMD} \times \{0, 1, \perp\}$, $m_0(01) = 1,\ m_0(\text{lock}) = 1$ *and* $m_0(p) = 0$ *for all* $p \notin \{01, \text{lock}\}$*. For* $0 \leq i \leq l$ *and* $1 \leq j \leq r_i$ *let* lock_{ij} *be the singleton set containing* lock *if* s_{ij} *is* not *inside an atomic region and the empty set otherwise. The transitions (T, src and tgt) are then given as follows:*

s_{ij}	$\ell(t)$	$src(t)$	$tgt(t)$
if b **goto** n	$(s_{ij}, 0)$	$\{ij\} \cup \text{lock}_{ij}$	$\{i(j+1)\} \cup \text{lock}_{ij}$
	$(s_{ij}, 1)$	$\{ij\} \cup \text{lock}_{ij}$	$\{in\} \cup \text{lock}_{ij}$
	$(s_{ij}, \bot)$	$\{ij\} \cup \text{lock}_{ij}$	$\emptyset$
goto n	$(s_{ij}, 1)$	$\{ij\} \cup \text{lock}_{ij}$	$\{in\} \cup \text{lock}_{ij}$
atc(b)	$(s_{ij}, 1)$	$\{ij, \text{lock}\}$	$\{i(j+1)\}$
	$(s_{ij}, \bot)$	$\{ij, \text{lock}\}$	$\emptyset$
end atc	$(s_{ij}, 1)$	$\{ij\}$	$\{i(j+1), \text{lock}\}$
spawn(p_x)	$(s_{ij}, 1)$	$\{ij\} \cup \text{lock}_{ij}$	$\{i(j+1), x1\} \cup \text{lock}_{ij}$
exit	$(s_{ij}, 1)$	$\{ij\} \cup \text{lock}_{ij}$	lock
$\alpha := \alpha', \textbf{new}(\alpha), \textbf{del}(\alpha)$	$(s_{ij}, 1)$	$\{ij\} \cup \text{lock}_{ij}$	$\{i(j+1)\} \cup \text{lock}_{ij}$
	$(s_{ij}, \bot)$	$\{ij\} \cup \text{lock}_{ij}$	$\emptyset$

If one of the target places is not in P we omit the corresponding out-edge (e.g. in case of thread termination or a jump out of range).

Example 3.8. The graph in Fig. 2 shows the Petri net modeling the control-flow semantics of the program from Fig. 1. The round nodes represent the places and the rectangles the (labeled) transitions of the net. If there are incoming and outgoing edges to the same place they are drawn as bidirectional arrows. In the initial state there are only tokens in the places 01 and lock.

Concrete DLMP-Semantics. Now that we defined the heap as well as the control flow semantics of our programming language we have to combine both.

Definition 3.9. *Let* $\mathfrak{P}^c = (P^c, T^c, src^c, tgt^c, \ell^c, m_0^c)$ *be the control flow and* $\mathfrak{P}^h = (P^h, T^h, src^c, tgt^c, \ell^h, m_0^h)$ *the heap semantics of* π*. The* concrete semantics *of* π *is the Petri net* $\mathfrak{P} := \mathfrak{P}^c \otimes \mathfrak{P}^h := (P^c \cup P^h, T, src, tgt, \ell, m_0)$ *where*

$$T = \{((t^c, t^h) \in T^c \times T^h \mid \ell^c(t^c) = \ell^h(t^h)\} \qquad \ell(t^c, t^h) = \ell^c(t^c)$$

$$src(t^c, t^h) = src(t^c) \cup src(t^h)$$
$$tgt(t^c, t^h) = tgt(t^c) \cup tgt(t^h) \qquad m_0(p) = \begin{cases} m_0^c(p) & \textit{if } p \in P^c \\ m_0^h(p) & \textit{otherwise} \end{cases}$$

As one might suspect the concrete semantics cannot be used as-is in verification techniques since DLM-programs are *Turing complete*[2].

4 Data Abstraction

To tackle the verification problem we use heap abstraction techniques to generate a *data abstract semantics* that over-approximates the behavior of the concrete one, i.e., whose runs cover all concrete ones. In our setting this approach is correct but generally incomplete: although we can conclude from the satisfaction of a

[2] DLM-programs can simulate a counter machine (the values of the counters are represented by lists of the corresponding length).

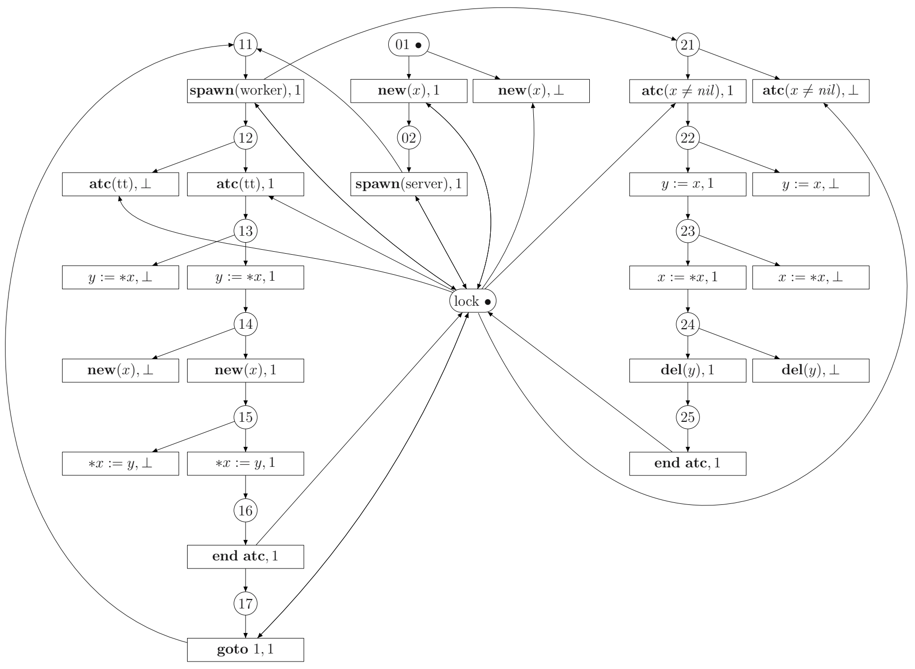

Fig. 2. Control-flow semantics for the server/worker example

property in the abstract state space the validity in the concrete case, the inverse is not possible anymore. Our heap abstraction is parameterized via a global constant $M \in \mathbb{N}$ which allows a systematic refinement. For a given $M > 0$ we set $\mathbb{M} := \{0, 1, ..., M, \star\}$, where $\star$ represents all values greater than M.

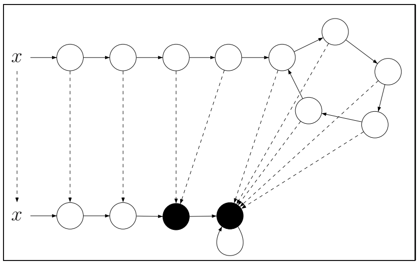

Fig. 3. An Abstraction Morphism

Chain Abstraction and Canonical Configurations. For heap abstraction we adopt the idea of *summary nodes*. Summary nodes are not allowed to represent arbitrary structures but only so-called *chains* which are non-interrupted sublists, i.e., list segments where only the head node is allowed to have more than one predecessor. This abstraction technique is well known [7,11,26]. For further details you may also refer to [15].

Based on the concept of chains one can define so-called *abstraction morphisms* which are surjective functions of the type $h : N_1 \to N_2$ for $H_i = (N_i, A_i, \mu_i, F_i) \in \boldsymbol{H}$ that retain the graph structure while collapsing chains of length greater than M to abstract nodes. In Fig. 3 an example is depicted.

We write $H_2 \leq H_1$ to denote that there is an abstraction morphism that abstracts H_1 to H_2. In this context we will also write $h(H_1) = H_2$ lifting h to heap configurations. If $|N_1| = |N_2|$, h is an isomorphism. We then write $H_1 \cong H_2$.

Note that a given source configuration can give rise to different abstractions. To obtain a unique canonical representation of a concrete heap configuration we collapse only *maximal* chains and do not abstract nodes that are closer than three μ-steps to a program variable. This yields a concrete expression semantics.

The set of all such canonical configurations is denoted by $\boldsymbol{H}_\natural$. It can be shown that for every concrete heap configuration a unique canonical configuration exists which is related to the former by a morphism $h_\natural$ [15]. We will use it as *abstraction function* in the following. The lower graph in Fig. 3 is a canonical configuration.

Abstract Heap Semantics. Regarding the expression semantics nothing needs to be modified in the data-abstract setting: in a canonical configuration, abstract nodes have a distance greater than two from the variable nodes such that every pointer expression refers to a concrete node. The expression semantics can therefore be chosen identical to the concrete case (Def. 3.4), now interpreted on canonical configurations.

Definition 4.1 (Abstract Heap Transformation Relation). *The abstract heap transformation relation* $\Rightarrow_h \subseteq (\boldsymbol{H}_\natural/_\cong \setminus \{\{H_{\mathrm{err}}\}\}) \times \mathrm{CMD} \times \{0, 1, \bot\} \times \boldsymbol{H}_\natural/_\cong$ *is depicted in Fig. 4 for* $H = (N, A, \mu, F) \in \boldsymbol{H}_\natural$. *We focus on assignments since the other rules are analogous to the concrete case. For simplicity we use representatives of the isomorphism classes.*

In Fig. 4 the semantic rules are visualized by examples. Rules 1 and 2 lead to a potential increase in the distance from variables to abstract nodes: consider an assignment of the form $y := nil$. If y points into a list whose head is referred to by another variable, we possibly increase the distance from that variable to abstract nodes. The assignment therefore potentially yields a non-canonical configuration making a re-abstraction necessary.

In rule 3 there might be the necessity for both concretization and abstraction. The execution of the assignment yields an intermediate configuration which is generally not canonical since the variable v could now be too close to an abstract node. Therefore we have to find a more concrete configuration H' whose abstraction yields the intermediate configuration. There might be more than one solution, thus this rule is nondeterministic (indicated by dashed arrows). After the concretization a re-abstraction is used to obtain the canonical form.

Due to our canonical representation, $\boldsymbol{H}_\natural/_\cong$ is finite and its size depends (linearly) on the number of program variables and the value of the precision constant M[3]. This implies the finiteness of the abstract heap semantics but not the boundedness of the data abstract program semantics as defined below.

[3] The number of nodes is bounded by $(2M + 3) \cdot |PV|$.

$$(1)\quad \frac{\alpha \notin *PV}{H, v := \alpha \overset{1}{\Rightarrow}_h h_\natural((N, A, \mu[v/[\![\alpha]\!]], \emptyset)\downarrow), \varepsilon}$$

$$(2)\quad \frac{[\![v]\!] \neq nil \qquad [\![\alpha]\!] \neq \bot}{H, *v := \alpha \overset{1}{\Rightarrow}_h h_\natural((N, A, \mu[\mu(v)/[\![\alpha]\!]], \emptyset)\downarrow), \varepsilon}$$

$$(3)\quad \frac{H' \in \boldsymbol{H} \text{ s.t. } (N, A, \mu[v/[\![*w]\!]], \emptyset)\downarrow \leq H' \text{ and } h_\natural(H') \in \boldsymbol{H}_\natural \qquad [\![w]\!] \neq nil}{H, v := *w \overset{1}{\Rightarrow}_h h_\natural(H'), \varepsilon}$$

$$(4)\quad \frac{[\![\alpha]\!] = \bot \ \vee [\![\alpha']\!] = \bot}{H, \alpha := \alpha' \overset{\bot}{\Rightarrow}_h H_{\text{err}}, \varepsilon}$$

(1,2) $v := w$ (analogously: $v = nil$, $v = \&w$, $*v := w$, $*v = *w$, $*v = \&w$)

(3) $v := *w$

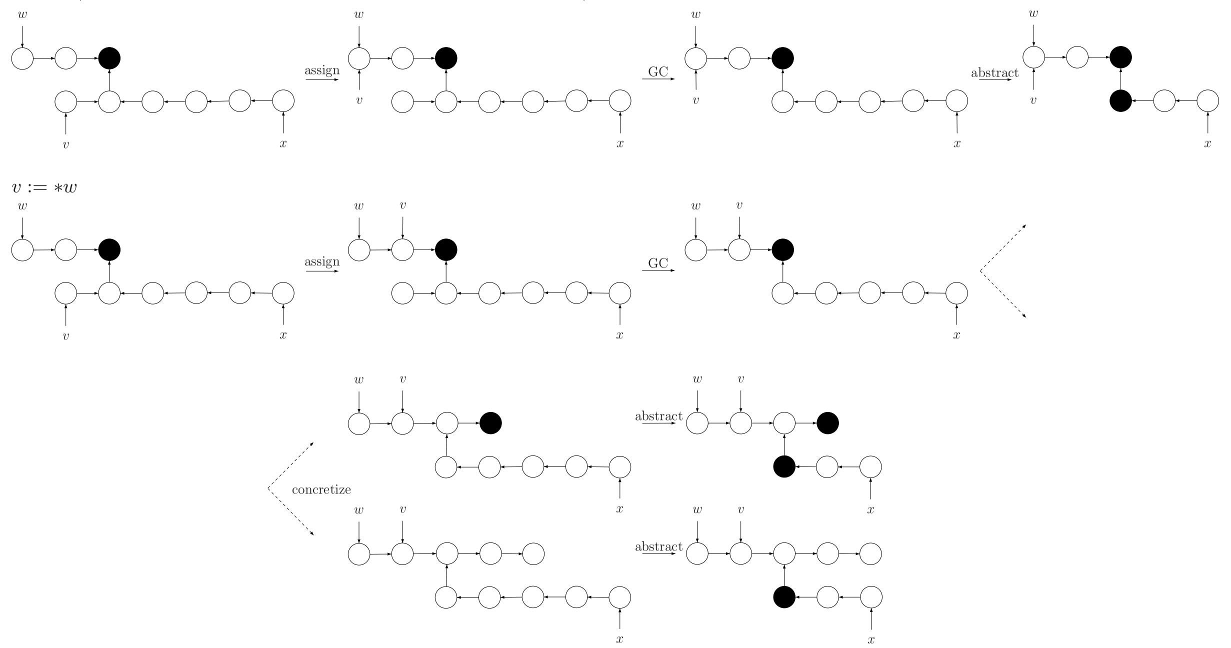

Fig. 4. Abstract rules and exemplary visualization ($M = 2$)

Definition 4.2. *The* abstract heap semantics *is the Petri net* $\mathfrak{P}^h_\natural := (P, T, src, tgt, \ell, m_0)$ *with* $P \subseteq \boldsymbol{H}_\natural/_\cong$, $T = \{(K, K', c, x) \mid K, c \overset{x}{\Rightarrow}_h K'\}$, $\ell(K, K', c, x) = (c, x)$, $src(K, K', c, x) = \{K\}$, $tgt(K, K', c, x) = \{K'\}$ *and* $m_0(K_0) = 1$ *for a* $K_0 \in P$ *(e.g. the empty heap congruence class) and* $m_0(K) = 0$ *for* $K \neq K_0$.

The data abstract program semantics *is given by the Petri net* $\mathfrak{P}_\natural := \mathfrak{P}^c \otimes \mathfrak{P}^h_\natural$ *where* $\otimes$ *is as in Def. 3.9.*

5 A Logic for Pointer Programs

In the previous sections we have defined our programming language and both its concrete and abstract semantics. In this section we will present a logic which allows us to reason about heap configurations and program behavior. In the following LV denotes a set of logical variables with $LV \cap PV = \emptyset$.

Pointer Logic. Pointer Logic deals with single configurations, and can be used to express graph properties as well as to inspect the special heap flags.

Definition 5.1. *We define the set of* Pointer Logic formulae *(PL-formulae) as follows:*

$$\begin{aligned} \text{NExp} &::= nil \mid v\ (\in PV) \mid x\ (\in LV) \mid *\text{NExp} \\ \text{Atomic} &::= \text{tt} \mid \text{ff} \mid f\ (\in \text{Flags}) \mid \text{NExp} = \text{NExp} \mid \text{NExp} \rightsquigarrow \text{NExp} \\ \text{PL} &::= \text{Atomic} \mid \neg\text{PL} \mid \text{PL} \wedge \text{PL} \mid \exists x : \text{PL} \end{aligned}$$

As usual we will use the logical operations $\vee$, $\rightarrow$, and $\forall$ as abbreviations. In contrast to pointer expressions in DLM-programs, the logic supports dereferencing operations of arbitrary depth. The predicate $\rightsquigarrow$ expresses the reachability of heap objects, whereas $=$ is true iff both expressions refer to the same object.

Definition 5.2. *Let* $\beta : LV \rightharpoonup N$ *be a valuation function instantiating logical variables with heap nodes and* $(N, \emptyset, \mu, F) \in \boldsymbol{H}_{\boldsymbol{\emptyset}}$ *a concrete heap configuration. Then we define* $[\![\cdot]\!] : \text{NExp} \rightharpoonup N_{nil}$ *for* $x \in LV$, $v \in PV$ *and* $\alpha \in \text{NExp}$ *by:*

$$[\![nil]\!] := nil \qquad [\![v]\!] := v \qquad [\![x]\!] := \beta(x) \qquad [\![*\alpha]\!] := \mu([\![\alpha]\!])$$

Note the semantic difference in comparison to the programming language. In the logic a variable v is interpreted by itself and not by the node it is referencing. This allows the check for identity of program variables without introducing a reference operator. The semantics of PL with respect to concrete heap configurations is quite standard and therefore omitted.

Reasoning about Abstract Computations. When switching to abstract configurations we run into several complications since logical variables can be bound to both concrete and abstract nodes. In the latter case we have to record to *which* concrete node, represented by the abstract node, a variable is bound. This could lead to undefinedness of PL-formulae. The problem mainly occurs in direct comparisons of the form $\alpha = \alpha'$. To solve it we choose the global precision constant M in dependence of the formula $\varphi \in \text{PL}$, assuming from now on that

$$M \geq \sum_{x \in \text{Variables}(\varphi)} \{j + 1 \mid *^j x \text{ occurs in } \varphi\},$$

and introduce the concept of *abstract valuations*.

Given $H \in \boldsymbol{H}_\natural$ and $\varphi \in \mathrm{PL}$, an abstract valuation is of the form $\eta = (\beta, o, \delta)$ where $\beta : PV \to N$ maps logical variables to (abstract) nodes, $o : PV \to \mathbb{M}$ denotes the offset of a variable "inside" an abstract node, and $\delta : PV \to PV \rightharpoonup \mathbb{M}$ is a "distance matrix" for the logical variables (referring to the same abstract node). δ is only defined if both arguments are mapped to the same entity, and o is only different from 1 if the corresponding variable is mapped to an abstract node. The set of all such valuations will be denoted by $\mathrm{Val}_{H,\varphi}$.

Using this concept one can define a function $d_{H,\eta} : \mathrm{NExp} \times \mathrm{NExp} \to \{0, 1, \infty\}$ measuring the "distance" of pointer expressions, where distance here means either 0 if the expressions are mapped onto the same (concrete) node, 1 if the second argument is reachable from the first, or ∞ if neither is the case (see [15] for details).

Definition 5.3. *Let $H = (N, A, \mu, F) \in \boldsymbol{H}_\natural$ and $\eta = (\beta, o, \delta) \in \mathrm{Val}_{H,\varphi}$. The satisfaction relation $\models$ for PL-formulae on canonical configurations is then given as follows (omitting the trivial cases):*

$$\begin{array}{rl}
H, \eta \models f & \textit{iff } f \in F, \textit{ where } f \in \mathrm{Flags} \\
H, \eta \models \alpha_1 = \alpha_2 & \textit{iff } d_{H,\eta}(\alpha_1, \alpha_2) = 0 \\
H, \eta \models \alpha_1 \rightsquigarrow \alpha_2 & \textit{iff } d_{H,\eta}(\alpha_1, \alpha_2) \leq 1 \\
H, \eta \models \exists x : \varphi & \textit{iff } \exists n \in N,\ \textit{off} \in \mathbb{M},\ \textit{dist} : V(\varphi) \rightharpoonup \mathbb{M} \textit{ s.t.} \\
& \quad H, (\beta_\eta[x/n], o_\eta[x/\textit{off}], \delta_\eta[x/\textit{dist}]) \models \varphi \\
H \models \varphi & \textit{iff } \exists \eta \in \mathrm{Val}_{H,\varphi} \textit{ s.t. } H, \eta \models \varphi \\
[H]_\cong \models \varphi & \textit{iff } H \models \varphi
\end{array}$$

Temporal Pointer Logic. Pointer Logic enables us to express properties of single configurations. However it cannot be used to specify (ongoing) computations, i.e., configuration sequences. To this aim we extend it by temporal operators.

Definition 5.4. *The set of* Temporal Pointer Logic formulae *(TPL-formulae) is given as follows:*

$$\mathrm{TPL} ::= \mathrm{PL} \mid \neg\mathrm{TPL} \mid \mathrm{TPL} \wedge \mathrm{TPL} \mid \boldsymbol{X}\,\mathrm{TPL} \mid \mathrm{TPL}\ \boldsymbol{U}\,\mathrm{TPL}$$

For $\varphi \in \mathrm{TPL}$ we use the the abbreviations $\boldsymbol{F}\varphi := \mathrm{tt}\,\boldsymbol{U}\varphi$ and $\boldsymbol{G}\varphi := \neg\boldsymbol{F}\neg\varphi$.

Note that it is *not* possible to nest PL-quantifiers and temporal operators. To do so it would be necessary to keep track of the object identities between states, which is difficult in the presence of abstract nodes. In addition it would blow up the state space and exclude the use of standard model checking algorithms. Only a few approaches support this idea [11,31]; most other works in the area consider only the *shape* of the heap. Clearly this restriction results in a loss of expressivity, nonetheless we can specify many interesting properties.

Example 5.5. For our server/worker system from Fig. 1 it holds true:

1. $\mathbf{GX}$ tt (never deadlock, i.e., there is always a successor state)
2. $\neg\mathbf{F}$ err (no pointer errors)
3. $\mathbf{GF}\ \exists n : \mathrm{new}_n$ (new objects are created infinitely often)
4. $\mathbf{GF}\ \mathrm{spawn}_{\mathrm{worker}}$ (infinitely often worker processes are spawned)

5. $\mathbf{G}(\exists n : \text{new}_n \to \mathbf{F}\ \text{spawn}_{\text{worker}})$
   (for every new object a worker thread is spawned)
6. $\neg\mathbf{G}(\text{spawn}_{\text{worker}} \to \mathbf{F}\ \text{del})$
   (the creation of a worker process does not necessarily result in the deletion of a node, i.e., fairness is not guaranteed)

More general correctness properties are:

7. $\mathbf{F}\ *v = *w$ (v and w will eventually become aliases)
8. $\mathbf{G}\neg(\exists x : (v \rightsquigarrow x \wedge w \rightsquigarrow x))$ (v and w always point to disjoint heap parts)
9. $\mathbf{G}(\forall x : (v \rightsquigarrow x \to (\neg\exists y : (x \rightsquigarrow y \wedge *y \rightsquigarrow x))))$
   (v always points to a non-cyclic list)
10. $\mathbf{FG}(\neg\text{leak})$ (only finitely many memory leaks can occur)
11. $\mathbf{G}(\forall x : (v \rightsquigarrow x \to (\forall y : (y \rightsquigarrow x \to v \rightsquigarrow y))))$ (v always points to a chain)

As mentioned before TPL specifies computation paths. These are given as sequences of heap configurations according to the Petri net representing the program semantics. By construction, for each marking m there is exactly one $p \in \boldsymbol{H}_\natural/_\cong \cup \boldsymbol{H}_\emptyset$ such that $m(p) = 1$.

Definition 5.6. *Let $\mathfrak{P}_\natural = (P, T, src, tgt, \ell, m_0)$ be the abstract (or concrete) semantics of π. For a given run $\rho \in Runs(\mathfrak{P}_\natural)$ the satisfaction relation $\models$ for $\varphi \in$ TPL, assuming w.l.o.g. that the maximal PL-subformulae in φ are closed, is defined as follows (again omitting the trivial cases):*

$$\begin{aligned}
&\varepsilon \not\models \varphi \\
&\rho \models \varphi\ (\in \text{PL}) && \textit{iff}\ \rho \neq \varepsilon\ \wedge\ \exists p \in P \cap (\boldsymbol{H}_\natural/_\cong \cup \boldsymbol{H}_\emptyset) : \rho_0(p) = 1\ \wedge\ p \models_{\text{PL}} \varphi \\
&\rho \models \boldsymbol{X}\varphi && \textit{iff}\ \rho[1] \models \varphi \\
&\rho \models \varphi \boldsymbol{U} \psi && \textit{iff}\ \exists k \leq |\rho| : \rho[k] \models \psi\ \textit{and}\ \forall j < k : \rho[j] \models \varphi
\end{aligned}$$

We write $\mathfrak{P}_\natural \models \varphi$ iff $\rho \models \varphi$ for all $\rho \in Runs(\mathfrak{P}_\natural)$ and $\pi \models \varphi$ iff $\mathfrak{P}^c \otimes \mathfrak{P}^h_\natural \models \varphi$.

Note that finite traces are included in the semantics of TPL. This implies that the equivalence $\neg\mathbf{X}\varphi \leftrightarrow \mathbf{X}\neg\varphi$ does generally *not* hold.

Model Checking Temporal Pointer Logic. The Turing completeness of DLM-programs implies that the model checking problem for TPL-formulae is undecidable. The following theorem shows that it suffices to employ data abstraction to obtain a positive result.

Theorem 5.7. *The data-abstract model checking problem is decidable, i.e., we can decide whether $\mathfrak{P}^c \otimes \mathfrak{P}^h_\natural \models \varphi$.*

Proof. The idea is to evaluate all maximal PL-subformulae on the heap configurations, to label (the transitions of) $\mathfrak{P}_\natural$ by atomic propositions and accordingly eliminate the PL-subformulae in φ to obtain an LTL-formula φ' (see Algorithm 6.4). The next step is to construct two automata $\mathfrak{A}$ and $\mathfrak{B}$ where $\mathfrak{A}$ is a finite automaton recognizing the finite words, and $\mathfrak{B}$ a nondeterministic Büchi-automaton accepting the infinite words satisfying φ'. Then according to [12] the model checking problem is decidable using a formula of the type defined in [32] to formulate the Büchi acceptance condition for $\mathfrak{B}$ and a reduction to the reachability problem for Petri net markings that is decidable in EXPSPACE [17]. □

The result is important but more of theoretical interest due to the high complexity of the problem. Thus we have to apply further simplifications to obtain practically feasible results.

6 Control-Flow Abstraction

The idea of the control-flow abstraction is similar to the data abstraction. Instead of recording for each Petri net place the exact number of tokens we only do this up to a certain resolution. A global constant $C \in \mathbb{N}$ parameterizes the resolution bound. $\mathbb{C} := \{0, ..., C, \star\}$ is used analogously to $\mathbb{M}$. What we obtain is an over-approximation $\mathfrak{P}^c_\natural$ of the concrete control-flow semantics $\mathfrak{P}^c$. The first step is the modification of the Petri net semantics.

Definition 6.1. *An* abstract Petri net *is of the form* $\mathfrak{P} = (P, T, src, tgt, \ell, \boldsymbol{m_0})$ *with* abstract markings *that are functions of the type* $\boldsymbol{m} : P \to \mathbb{C}$.

Definition 6.2. *Let* $\mathfrak{P} = (P, T, src, tgt, \ell, \boldsymbol{m_0})$ *be an abstract Petri net,* $\boldsymbol{m}, \boldsymbol{m'} \in$ *Mark*$(\mathfrak{P})$ *and* $t \in T$. *Then* $\blacktriangleright_t \subseteq Mark(\mathfrak{P}) \times T \times Mark(\mathfrak{P})$ *is given by*[4]*:*

$$\boldsymbol{m} \blacktriangleright_t \boldsymbol{m'} \;\Leftrightarrow \forall p \in srct : \boldsymbol{m}(p) > 0 \;\wedge\; \forall p \in P :$$

$$\boldsymbol{m'}(p) = \begin{cases} \boldsymbol{m}(p) - 1 & \textit{if } p \in src(t) \setminus tgt(t) \textit{ and } \boldsymbol{m}(p) \neq \star \\ C \textit{ or } \star & \textit{if } p \in src(t) \setminus tgt(t) \textit{ and } \boldsymbol{m}(p) = \star \\ \boldsymbol{m}(p) + 1 & \textit{if } p \in tgt(t) \setminus src(t) \\ \boldsymbol{m}(p) & \textit{otherwise} \end{cases}$$

The abstract control-flow semantics $\mathfrak{P}^c_\natural$ is defined as the concrete one, but using the abstract transition relation $\blacktriangleright$.

Definition 6.3. *The* abstract semantics *of* π *is the Petri net* $\mathfrak{P}_{\natural\natural} := \mathfrak{P}^c_\natural \otimes \mathfrak{P}^h_\natural$.

If we now want to apply model checking, i.e., verify that a TPL-formula φ is satisfied by $\mathfrak{P}_{\natural\natural}$, we evaluate all maximal PL-subformulae of φ on the heaps in $\mathfrak{P}_{\natural\natural}$, substitute them by atomic propositions, generate the underlying (finite) transition system, label it with atomic propositions according to the evaluation of subformulae, and solve the resulting model checking problem for LTL with finite traces [14].

Algorithm 6.4. *Let* $\mathfrak{P}_{\natural\natural} = (P, T, src, tgt, \ell, \boldsymbol{m_0})$ *be given and* $\varphi \in$ TPL *the formula to verify. Let* $\Psi := \{\psi \in \text{PL} \mid \psi \textit{ is a maximal subformula of } \varphi\} = \{\psi_1, ..., \psi_r\}$ *and* $a_1, ..., a_r$ *be atomic propositions.*

1. *Generate a finite transition system* $\mathcal{T} := (\{\boldsymbol{m} \mid \boldsymbol{m_0} \blacktriangleright^\star \boldsymbol{m}\}, \boldsymbol{m_0}, \blacktriangleright, lab)$ *with*
$$lab(m) := \bigcup_{i=1}^{r} \{a_i \mid \exists p \in P \cap \boldsymbol{H}_\natural/_\cong : m(p) = 1 \wedge p \models \psi_i\}$$
2. *Solve* $\mathcal{T} \models^?_{\text{LTL}} \varphi[\psi_1/a_1, ..., \psi_r/a_r]$ *(admitting finite traces).*

[4] Note that $\blacktriangleright_t$ can be nondeterministic for a given transition t.

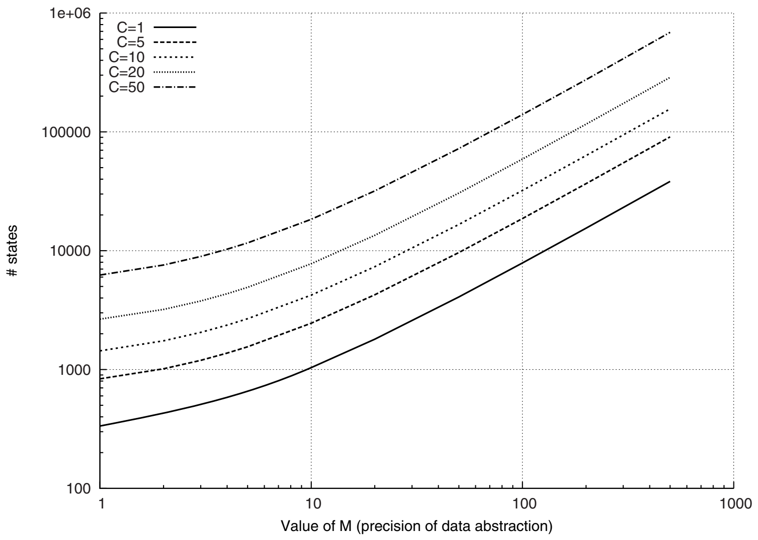

Fig. 5. Size of the state space for the server/worker example

Limitations and Refinement. Due to the over-approximation of the state space, there may exist abstract computations falsifying the property to verify and not corresponding to concrete ones. These *false negatives* can be eliminated through abstraction refinement by increasing the parameters M and C. The size of the state space is a *linear* function wrt. M (and C). This is visualized in Fig. 5 for our server/worker example employing a prototype version of our tool which is currently under development (note the logarithmic scale of both axes). Thanks to the implicit universal quantification over paths in the LTL approach, however, the successful verification of a property in the abstract case implies its correctness in the concrete case, i.e., *false positives* are excluded.

Note that our framework can be easily extended to three truth values, to eliminate false positives. The "don't know" answer would then only be given if the resulting transition system contains both positive *and* negative traces. In the other cases the answer would be an exact "yes" or "no". This would require the additional checking of a CTL formula in the case that the LTL model checker falsifies the property. If the answer is "don't know" a refinement step by increasing M and/or C is necessary.

7 Conclusions and Future Work

We have presented a framework for the verification of concurrent pointer-manipulating programs with dynamic thread creation, unbounded heap size, and destructive updates. Correctness properties are specified using temporal pointer logic (TPL) which is essentially a pointer logic for expressing heap properties enriched with temporal operators. Rather than requiring dedicated algorithms, the TPL model checking problem is reduced to an LTL model checking problem by appropriate abstractions. The trade-off is the restriction to list-like data structures and to static variables as well as the limitation in expressiveness of the logic because object identities are not tracked between configurations.

Currently we are implementing the method to analyze more interesting examples. We are planning to support the user in handling abstract computations which violate a given property, either by deriving concrete counterexamples or by suggesting refinements to eliminate false negatives. Finally we are working on an extension to arbitrary data structures.

Acknowledgments. We would like to thank Ulrich Schrempp for developing the prototype implementation of our analysis framework, which was used for computing the state spaces in the server/worker example.

References

1. Balaban, I., Pnueli, A., Zuck, L.D.: Shape Analysis by Predicate Abstraction. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 164–180. Springer, Heidelberg (2005)
2. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
3. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
4. Bouajjani, A., Habermehl, P., Moro, P., Vojnar, T.: Verifying programs with dynamic 1-selector-linked list structures in regular model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 13–29. Springer, Heidelberg (2005)
5. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
6. Bozga, M., Iosif, R., Lakhnech, Y.: On logics of aliasing. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 344–360. Springer, Heidelberg (2004)
7. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI 1990, pp. 296–310. ACM Press, New York (1990)
8. Dams, D., Namjoshi, K.S.: Shape Analysis through Predicate Abstraction and Model Checking. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 310–323. Springer, Heidelberg (2002)
9. Das, S., Dill, D.L.: Successive approximation of abstract transition relations. In: LICS 2001, pp. 51–58. IEEE Computer Society Press, Los Alamitos (2001)
10. Das, S., Dill, D.L., Park, S.: Experience with Predicate Abstraction. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 160–171. Springer, Heidelberg (1999)
11. Distefano, D., Katoen, J.-P., Rensink, A.: Safety and liveness in concurrent pointer programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 280–312. Springer, Heidelberg (2006)
12. Esparza, J.: On the decidability of model checking for several μ-calculi and Petri nets. In: Tison, S. (ed.) CAAP 1994. LNCS, vol. 787, pp. 115–129. Springer, Heidelberg (1994)
13. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI 2007, pp. 266–277. ACM Press, New York (2007)

14. Havelund, K., Rosu, G.: Testing linear temporal logic formulae on finite execution traces. Technical Report TR 01-08, RIACS (2001)
15. Katoen, J.-P., Noll, T., Rieger, S.: Verifying concurrent list-manipulating programs by LTL model checking. Technical Report 2007-06, RWTH Aachen University, Dept. of Computer Science, Germany (April 2007)
16. Lahiri, S.K., Qadeer, S.: Verifying properties of well-founded linked lists. In: POPL 2006, pp. 115–126. ACM Press, New York (2006)
17. Lambert, J.L.: A structure to decide reachability in Petri nets. Theor. Comput. Sci. 99(1), 79–104 (1992)
18. Lev-Ami, T., Immerman, N., Reps, T.W., Sagiv, S., Srivastava, S., Yorsh, G.: Simulating Reachability Using First-Order Logic with Applications to Verification of Linked Data Structures. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 99–115. Springer, Heidelberg (2005)
19. Manevich, R., Berdine, J., Cook, B., Ramalingam, G., Sagiv, M.: Shape analysis by graph decomposition. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 3–18. Springer, Heidelberg (2007)
20. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005)
21. Nystrom, E.M., Kim, H.-S., Hwu, W.W.: Bottom-up and top-down context-sensitive summary-based pointer analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 165–180. Springer, Heidelberg (2004)
22. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: POPL 2004, pp. 268–280. ACM Press, New York (2004)
23. Podelski, A., Wies, T.: Boolean Heaps. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 268–283. Springer, Heidelberg (2005)
24. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE Computer Society Press, Los Alamitos (2002)
25. Rugina, R., Rinard, M.: Pointer analysis for multithreaded programs. SIGPLAN Not. 34(5), 77–90 (1999)
26. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM Trans. Program. Lang. Syst. 20(1), 1–50 (1998)
27. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
28. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: PPoPP 2001, pp. 12–23. ACM Press, New York (2001)
29. Vechev, M.T., Yahav, E., Bacon, D.F.: Correctness-preserving derivation of concurrent garbage collection algorithms. In: PLDI 2006, pp. 341–353. ACM Press, New York (2006)
30. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. ACM SIGPLAN Notices 36(3), 27–40 (2001)
31. Yahav, E., Reps, T., Sagiv, M., Wilhelm, R.: Verifying Temporal Heap Properties Specified via Evolution Logic. In: Degano, P. (ed.) ESOP 2003 and ETAPS 2003. LNCS, vol. 2618, pp. 204–222. Springer, Heidelberg (2003)
32. Yen, H.-C.: A unified approach for deciding the existence of certain Petri net paths. Inf. Comput. 96(1), 119–137 (1992)
33. Yong, S.H., Horwitz, S.: Pointer-range analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 133–148. Springer, Heidelberg (2004)
34. Zhu, J., Calman, S.: Symbolic pointer analysis revisited. In: PLDI 2004, pp. 145–157. ACM Press, New York (2004)