RWTH Aachen

Department of Computer Science
Technical Report

Verifying Concurrent List–Manipulating Programs by LTL Model Checking

Joost–Pieter Katoen, Thomas Noll and Stefan Rieger

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2007-06

RWTH Aachen · Department of Computer Science · April 2007

Verifying Concurrent List–Manipulating Programs by LTL Model Checking

Joost–Pieter Katoen, Thomas Noll and Stefan Rieger

Software Modeling and Verification Group Lehrstuhl für Informatik II
RWTH Aachen, Germany
Email: {katoen, noll, rieger}@cs.rwth-aachen.de

Abstract. We present a novel approach to the verification of concurrent pointer–manipulating programs which operate on singly–linked lists. By abstracting from chains (i.e., non–interrupted sublists) in the heap, we obtain a finite–state representation of all possible executions of a given program. The combination of a simple pointer logic for expressing heap properties and of temporal operators then allows us to employ standard LTL model checking techniques. The usability of this approach is demonstrated by establishing correctness properties of a producer/consumer system and of a concurrent garbage collector.

1 Introduction

Techniques for the verification of elementary properties of concurrent pointer programs are indispensable. Programming with pointers is error–prone with potential pitfalls such as dereferencing null pointers and the creation of memory leaks. Pointer programming becomes even more vulnerable in a concurrent setting where data structures such as linked lists and trees are manipulated and inspected by several threads.

This paper presents a model–checking approach to the verification of concurrent programs that manipulate singly–linked lists. Existing approaches either make use of non–standard logics, advanced model–checking procedures or extended versions of Hoare logics with accompanying deduction techniques (see Sct. 6 about related work). In contrast, the approach advocated in this paper stays within the realm of traditional (linear–time) model checking. This facilitates the usage of standard (LTL) 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 pointer manipulation, cell creation and destruction, and (guarded) atomic regions that allow concurrency control constructs such as test–and–set primitives and monitors. An operational semantics is provided in terms of labeled transition systems in which states are equipped with a graph structure representing the current list configuration. List abstraction exploits a variant of summary nodes [SRW98] that represent more than M chained list cells where constant M is directly obtained from the formula to be checked. Each configuration is shown to have a canonical representation (up to isomorphism). The abstract semantics of any concurrent program in our language is finite, obtained in a fully mechanized manner, and keeps the minimal “distance” between program variables and summary nodes invariant. Over–approximation occurs in a very controlled manner; only assignments may yield nondeterminism as variables may get "too close" to summary nodes.

Properties are expressed in a first–order linear–time temporal logic (LTL) that is enriched with assertions on singly–linked lists such as reachability of cells, aliasing, and freshness of cells. Our logic is similar in spirit to NTL [DKR04,DKR06] and ETL [YRSW03]. Opposed to NTL, we avoid the use of temporal operators inside quantification. In this way, involved mechanisms to keep track of the identities of individual cells are not needed. As a result, standard LTL model checking algorithms can be employed. The differences with ETL are more of a technical nature. ETL has a three–valued interpretation, whereas our logical interpretation is a standard binary one. Moreover, ETL–formulas are translated in first–order logic with transitive closure for the evaluation on a trace, whereas in our case traces are generated by labeled transition systems and used in standard LTL model checking. The feasibility of our approach is shown by considering the verification of a simple concurrent garbage collection program. Furthermore a prototypical tool is currently under development for experimenting with real–life examples.

Please note that due to space constraints most of the proofs could not be included in this paper.

2 A List–Manipulating Programming Language

Given a universe PV of program variables, we define the set of *list–manipulating programs* (LM–programs) to be given by the following grammar (where $v_i, v \in PV$):

$$
\begin{array}{rl}
\text{LMP} & ::= \textbf{var}\ v_1, ..., v_k(\text{Stmt}_1 \| ... \| \text{Stmt}_l) \\
\text{Stmt} & ::= \textbf{skip} \mid \textbf{signal} \mid v := \text{PExp} \mid *v := \text{PExp} \mid \text{Stmt}; \text{Stmt} \\
 & \quad \mid \textbf{if}\ \text{BExp}\ \textbf{then}\ \text{Stmt}\ \textbf{else}\ \text{Stmt}\ \textbf{fi} \mid \textbf{while}\ \text{BExp}\ \textbf{do}\ \text{Stmt}\ \textbf{od} \\
 & \quad \mid \textbf{new}(\text{PExp}) \mid \textbf{del}(\text{PExp}) \mid \langle \text{BExp} : \text{Stmt} \rangle \\
\text{PExp} & ::= \mathit{nil} \mid v \mid *v \mid \&v \\
\text{BExp} & ::= \text{tt} \mid \text{ff} \mid \text{PExp} = \text{PExp} \mid \text{BExp} \wedge \text{BExp} \mid \neg \text{BExp}
\end{array}
$$

$V(\pi) := \{v_1, ..., v_k\}$ denotes the set of variables for $\pi \in \text{LMP}$.

$$
\begin{array}{l}
\textbf{var}\ x, y, z(\\
\quad \textbf{while}\ \text{tt}\ \textbf{do}\ \langle \text{tt} : \\
\quad\quad \textbf{if}\ x = \mathit{nil} \\
\quad\quad\quad \textbf{then}\ \textbf{new}(y);\ x := y \\
\quad\quad\quad \textbf{else}\ \textbf{new}(*y);\ y := *y \\
\quad\quad \textbf{fi} \\
\quad \rangle\ \textbf{od} \\
\| \quad \textbf{while}\ \text{tt}\ \textbf{do}\ \langle x \neq \mathit{nil} : \\
\quad\quad z := x;\ x := *x;\ \textbf{del}(z) \\
\quad \rangle\ \textbf{od} \\
)
\end{array}
$$

Fig. 1. Producer/Consumer

An LM–program thus consists of a declaration of global program variables and a series of statements to be executed in parallel. Each of these statements can either be a pointer assignment, a sequence of statements, a control structure, or a special statement such as **signal** which sets a global signal flag that can be tested in the logic, **new**/**del** for dynamic creation or deletion of objects at runtime (possibly leading to an unbounded number of allocated heap cells) and guarded atomic regions. If the Boolean guard g in $\langle g : s \rangle$ is true, s is executed atomically, i.e., with no interference by other processes. If g is evaluated to false, the process is blocked (until g becomes true).

Pointer expressions comprise the special constant *nil* denoting an undefined pointer value, a program variable, the dereferencing or referencing of a program variable. Note that for simplicity we do not allow arbitrary dereferencing depths; those can be emulated using a sequence of assignments within an atomic region.

Example 1. Figure 1 shows an LM–program implementing a producer inserting objects and a consumer deleting objects at the end (pointed to by y) and beginning (pointed to by x) of a queue, respectively. If the queue is empty the consumer cannot proceed due to the guard $x \neq nil$ until the producer has inserted at least one object. Insertion and deletion are executed atomically to prevent interferences.

Definition 1. *A* heap configuration *of a program* $\pi \in \mathrm{LMP}$ *is a tuple* $\gamma = (N, A, \mu, F)$ *with a set of nodes* $N \supseteq V(\pi)$*, 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* $F \subseteq \{\mathrm{err}, \mathrm{dl}, \mathrm{leak}, \mathrm{signal}, \mathrm{new}\}$*.*

Let $\mu^\star : 2^N \to 2^N$ *with* $\mu^\star(X) := \{n \in N \mid \exists k \in \mathbb{N},\ \exists n' \in X : \mu^k(n') = n\}$ *be the transitive closure of* μ*, i.e. all nodes reachable from a node in* X *(and* X *itself).*

Thus the nodes represent both the dynamic objects created and deleted 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 abstraction technique 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 new node was created, or the signal bit has been set using the **signal** command.

To ensure the finiteness of our abstraction we will automatically delete those heap nodes that are not reachable from the program variables. This is accomplished by the following *garbage collection* mapping. Whenever it removes an unreachable node, it sets the leak flag indicating a potential memory leak.

Definition 2. *For* $\gamma = (N, A, \mu, F)$ *we define* $\gamma{\downarrow} := (N', A \cap N', \mu \restriction N', F \cup \{\mathrm{leak} \mid (N \setminus N') \neq \emptyset\})$ *where* $N' = \mu^\star(PV)$*.*

$\varGamma$ *denotes the set of all* garbage–free heap configurations*, i.e.,* $\forall \gamma \in \varGamma : \gamma{\downarrow} = \gamma$*, and* $\varGamma_c \subseteq \varGamma$ *denotes the set of all concrete configurations, i.e., those with* $A_\gamma = \emptyset$*.*

From now on we will always assume garbage freeness when mentioning heap configurations. This enforces a bound on the maximal number of incoming edges for a node (essentially the number of program variables).

Definition 3. *Let* $\gamma = (N, \emptyset, \mu, F) \in \varGamma_c$*. Then we define the semantics of pointer expressions* $[\![\cdot]\!] : \mathrm{PExp} \rightharpoonup N_{nil}$ *by*[1]*:*

$$
\begin{aligned}
[\![nil]\!] &:= nil \qquad & [\![*v]\!] &:= \mu([\![v]\!]) \\
[\![v]\!] &:= \mu(v) \qquad & [\![\&v]\!] &:= v
\end{aligned}
$$

[1] $\rightharpoonup$ denotes a partial function and $\perp$ the undefined value.

The semantics of Boolean expressions $[\![\cdot]\!] : \mathrm{BExp} \rightharpoonup \mathbb{B}$ is standard and strict[2]. Note that Def. 1 implies that $\mu(nil) = \perp$ and so $[\![\cdot]\!]$ can indeed yield undefined results for both pointer and Boolean expressions.

Definition 4. *For* $\pi = \mathrm{var}\ v_1, ..., v_k : (s_1 \| ... \| s_l) \in \mathrm{LMP}$ *the* concrete operational semantics *is given by a transition system* $T^c_\pi = (Q, q_0, lab, \rightarrow)$ *with a set of states* $Q \subseteq \Gamma_c \times \mathrm{Stmt}_\diamond(\{\|\}\mathrm{Stmt}_\diamond)^\star$ *where* $\mathrm{Stmt}_\diamond = \mathrm{Stmt} \cup \{\diamond\}\mathrm{Stmt} \cup \{\varepsilon\}$*, an initial state* $q_0 = ((N_0, \emptyset, \mu_0, \emptyset), s_1\|...\|s_l)$ *where* N_0 *and* μ_0 *represent the "input heap", a labeling* $lab : Q \rightarrow \Gamma_c$ *with* $\forall(\gamma, s) \in Q : lab((\gamma, s)) = \gamma$*, and a transition relation* $\rightarrow \subseteq Q \times Q$*.*

In the following we will use the abbreviations $\hat{F}$ *for* $F \setminus \{\mathrm{signal}, \mathrm{new}, \mathrm{leak}\}$ *and* noerr *for* $\{\mathrm{err}, \mathrm{dl}\} \cap F = \emptyset$*.* γ_{err} *and* γ_{dl} *will denote pointer error and deadlock states. Most transition rules are straightforward, thus here we will only consider some interesting examples.*

$$\frac{[\![g]\!] = 1 \qquad \gamma, s \rightarrow \gamma', s' \qquad \mathrm{noerr}}{\gamma, \langle g : s\rangle \rightarrow \gamma', \diamond s'} \tag{1}$$

$$\frac{\gamma, s \rightarrow \gamma', s' \qquad s' \neq \varepsilon \qquad \mathrm{noerr}}{\gamma, \diamond s \rightarrow \gamma', \diamond s'} \qquad \frac{\gamma, s \rightarrow \gamma', \varepsilon \qquad \mathrm{noerr}}{\gamma, \diamond s \rightarrow \gamma', \varepsilon} \tag{2}$$

$$\frac{\exists j \ s.t.\ \gamma, s_j \rightarrow \gamma', s'_j \qquad \forall i \neq j :\ \nexists s'_i\ s.t.\ s_i = \diamond s'_i \qquad \mathrm{noerr}}{\gamma, s_1\|...\|s_k \rightarrow \gamma', s_1\|...\|s'_j\|...\|s_k} \tag{3}$$

$$\frac{\nexists j \ s.t.\ \gamma, s_j \rightarrow \gamma', s'_j \qquad \exists j : s_j \neq \varepsilon \qquad \mathrm{noerr}}{\gamma, s_1\|...\|s_k \rightarrow \gamma_{\mathrm{dl}}, \varepsilon} \tag{4}$$

$$\frac{}{\gamma, \varepsilon\|...\|\varepsilon \rightarrow \gamma, \varepsilon\|...\|\varepsilon} \tag{5}$$

$$\frac{[\![\alpha]\!] \neq \perp \qquad \mathrm{noerr}}{(N, A, \mu, F), v := \alpha \rightarrow (N, A, \mu[v/[\![\alpha]\!]], \hat{F})\!\downarrow, \varepsilon} \tag{6}$$

$$\frac{\mathrm{noerr}}{(N, A, \mu, F), \mathbf{new}(v) \rightarrow (N \uplus \{n_{\mathrm{new}}\}, A, \mu[v/n_{\mathrm{new}}], \hat{F} \cup \{\mathrm{new}\})\!\downarrow, \varepsilon} \tag{7}$$

$$\frac{[\![\alpha]\!] \in N \setminus PV \qquad \mathrm{noerr}}{(N, A, \mu, F), \mathbf{del}(\alpha) \rightarrow (N \setminus \{[\![\alpha]\!]\}, A, \mu[[\![\alpha]\!]/\perp, \mu^{-1}([\![\alpha]\!])/nil], \hat{F})\!\downarrow, \varepsilon} \tag{8}$$

Some remarks on the transition rules are in order. The leak, signal, and new flags are reset after each transition; they are only activated in the state directly following the corresponding "event".

Regarding the concurrency rules we need to take care of the special semantics of the atomic regions. If a process is executing such a statement it must not be interrupted, and therefore the corresponding state is marked with $\diamond$ (rule 1). The interleaving rule 3 excludes that any other than process j is in an atomic region. If no process can proceed (all are blocked) we reach the special deadlock state (rule 4). If all processes are terminated or an error or deadlock state is reached

[2] One undefined operand yields an undefined expression.

the program loops to ensure that all paths in the transition system are infinite (rule 5).

The treatment of assignments (rule 6) and the **new** statement (rule 7) is again straightforward, we though have to keep in mind in the first case that runtime errors might occur (dereferencing of *nil* pointers) and that garbage may be generated. Rule 8 handles the deletion of nodes. Please note that the next–pointers of the predecessors of the deleted node are set to *nil* (mainly to avoid case distinctions for undefined expressions in the semantics).

We conclude that for the producer/consumer example (Fig. 1) the state space becomes infinite when applying the operational semantics as defined above.

3 State–Space Abstraction

As we have seen in the previous section the state space of LM–programs can get infinite even for simple example programs making standard verification methods inapplicable. To tackle the problem we use abstraction techniques to generate an *abstract transition system* that incorporates the behavior of the concrete one, i.e., whose runs cover all concrete ones. This approach is correct but generally incomplete: although we can conclude from the satisfaction of a property in the abstract state space its validity in the concrete case, the inverse is impossible though. But since the abstraction is parameterized via a global constant $M \in \mathbb{N}$ we can refine the abstraction depending on our needs. For a given $M > 0$ we set $\mathbb{M} := \{0, 1, ..., M, \star\}$, where $\star$ represents all values greater than M.

Chain Abstraction The main idea of our abstraction is to summarize subgraphs of a configuration into summary nodes [SRW98], which will be exactly those contained in the A–component of a heap configuration. Summary nodes (also called abstract nodes) are not allowed to represent arbitrary structures but only so–called *chains*, i.e., non–interrupted lists. Our abstraction is based on [Dis03,DKR04] with the difference that nodes are either truly abstract or concrete, thus recording node multiplicities is not necessary.

Definition 5. *Let $\gamma = (N, A, \mu, F) \in \Gamma$ be a configuration. A nonempty set of nodes $C \subseteq N$ is called a* chain *if either*

- $|C| = 1$ *and* $C \subseteq PV$ *or*
- $C \cap PV = \emptyset$ *and there exists a bijection* $\pi : \{1, ..., |C|\} \to C$ *such that* $\mu(\pi(i)) = \pi(i + 1)$ *for* $i \in \{1, ..., |C|\}$ *and* $\forall i \in \{2, ..., |C|\} : |\mu^{-1}(\pi(i))| = 1$.

For a given chain C we will use the abbreviations $\overleftarrow{C} := \pi(1)$, and $\overrightarrow{C} := \pi(|C|)$. A chain is called maximal *if no superset $C' \supset C$ is a chain.*

Thus a chain is a sequence of pointer–connected nodes without interference of other incoming edges or a singleton set containing a program variable. It follows that the abstraction of chains preserves the graph structure. We will now introduce a type of functions, called *abstraction morphisms*, that is based on this concept.

Definition 6. *Let $\gamma_i = (N_i, A_i, \mu_i, F_i) \in \Gamma$, $i \in \{1, 2\}$ be two heap configurations. An* abstraction morphism *$h : N_1 \to N_2$ satisfies for all $v \in PV \cap N_1$ and $n_i, n'_i \in N_i$:*

1. $h(v) = v$
2. $h^{-1}(n_2)$ *is a chain in* N_1
3. $\mu_2(n_2) = n_2' \;\Rightarrow\; \mu_1(\overrightarrow{h^{-1}(n_2)}) = \overleftarrow{h^{-1}(n_2')}$
4. $\mu_1(n_1) = n_1' \Rightarrow h(n_1) = h(n_1') \vee \mu_2(h(n_1)) = h(n_1')$
5. $n_2 \in A_2 \;\Leftrightarrow\; h^{-1}(n_2) \cap A_1 \neq \emptyset \;\vee\; |h^{-1}(n_2)| > M$
6. $F_1 = F_2$

We write $h : \gamma_1 \rightarrowtail \gamma_2$ *to denote that the abstraction morphism* h *abstracts* γ_1 *to* γ_2 *and* $\gamma_2 \leq \gamma_1 \;\Leftrightarrow\; \exists h : \gamma_1 \rightarrowtail \gamma_2$.

Abstraction morphisms abstract from concrete chains with minimal length $M + 1$ (cond. 2 and 5). The preservation of the graph structure is enforced by conditions 3 and 4. Program variables, being special nodes, remain untouched (cond. 1).

Example 2. Figure 2 shows an abstraction morphism for $M = 1$. The dashed lines represent the mapping, and the black nodes denote the resulting abstract nodes. Note that for $M = 2$ the nodes 3 and 4 could not be projected onto the same abstract node (condition 5 of Def. 6). The chain $\{3, 4\}$ cannot be extended by node 5, since this node has two incoming edges which is only allowed for the first node of a chain. Although in this example the source configuration is concrete, this is of course not necessary by definition.

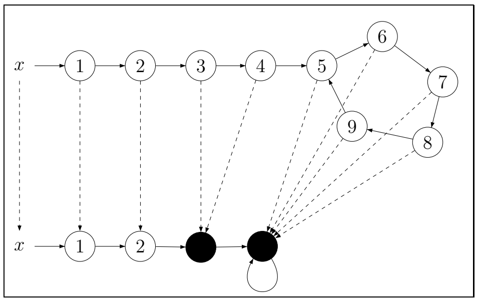

Fig. 2. An Abstraction Morphism

An important property of abstraction morphisms is their surjectivity. If, in addition a morphism is injective it becomes an isomorphism. Isomorphic configurations cannot be distinguished except for node naming, the graph structure is the same.

Canonical Configurations Previously we have defined how configurations can be abstracted. It remains the problem that there can be different abstractions of a given source configuration. For this reason we need a normal form that implies uniqueness. In the following we define this normal form, assuming $\gamma = (N, A, \mu, F) \in \Gamma$.

Definition 7.

1. *Let* $\lfloor N \rceil_j := \{n \in N \mid \nexists v \in PV : \mu^k(v) = n,\; k < j\}$ *be the set of nodes with a distance of at least* j *from the variable nodes. Analogously* $\lceil N \rfloor_j := N \setminus \lfloor N \rceil_{j+1}$.
2. *A configuration* γ *is called* canonical *if* $\lceil N \rfloor_2 \cap A = \emptyset$ *and for all maximal*[3] *chains* $C \subseteq \lfloor N \rceil_3$ *either* $|C| = 1$ *or* $|C| \leq M \;\wedge\; C \cap A = \emptyset$. *The set of all canonical configurations is denoted by* $\Gamma_\natural$.

The notion of canonical configurations is quite intuitive: maximal chains are collapsed where possible but only up to a distance of three from variable nodes. The latter condition ensures that pointer expressions always evaluate to concrete

[3] Here we refer to maximality in $\lfloor N \rceil_3$.

nodes, which will simplify the definition of the abstract LMP semantics. The abstraction morphism in Fig. 2 yields a canonical configuration, as can be easily verified.

Theorem 1 (Existence). *For every $\gamma \in \Gamma$ with $\lceil N \rceil_2 \cap A = \emptyset$ there exists a $\gamma' \in \Gamma_\natural$ such that $\gamma' \leq \gamma$.*

It is easy to construct a morphism yielding a canonical configuration. It has to collapse maximal chains that are larger than M or contain abstract nodes, if they are sufficiently distant from the variable nodes. In the following we will call this morphism $h_\natural$. The precise definition does not matter as states the following theorem.

Theorem 2 (Uniqueness). *Let $\gamma \in \Gamma$ and $\gamma_1, \gamma_2 \in \Gamma_\natural$ such that $h_1 : \gamma \rightarrowtail \gamma_1$ and $h_2 : \gamma \rightarrowtail \gamma_2$ are two abstraction morphisms. Then γ_1 and γ_2 are isomorphic.*

The proof of the uniqueness had to be omitted here. The consequence of these results is the appropriateness of canonical configurations as a normal form. The abstract semantics will operate on such configurations.

Abstract Semantics of List–Manipulating Programs As already mentioned, our goal is to guarantee the correctness of our abstraction approach. This can be achieved by ensuring that every concrete execution of a given system can be "simulated" by an abstract computation, which necessarily introduces nondeterministic behavior on the abstract side.

Regarding the expression semantics nothing needs to be modified: 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), now interpreted on canonical configurations.

Definition 8. *Given a program $\pi =$ **var** $v_1, ..., v_k : (s_1 \| ... \| s_l) \in \mathrm{LMP}$, its* abstract operational semantics *is defined by the labeled transition system $T^a_\pi = (Q, [q_0]_\cong, lab, \rightarrow)$ with state set $Q \subseteq \Gamma_\natural/_\cong \times \mathrm{Stmt}_\diamond(\{\|\}\mathrm{Stmt}_\diamond)^\star$, initial state q_0 as in Def. 4, labeling function $lab : Q \rightarrow \Gamma_\natural$ where $\forall (K, s) \in Q : lab((K, s)) = K$, and transition relation $\rightarrow$ as specified by the following rules (we focus on the assignments, since the other rules are analogous to the concrete case, but operating on isomorphism congruence classes).*

$$\frac{\alpha \notin *V(\pi) \qquad \text{noerr}}{[(N, A, \mu, F)]_\cong, v := \alpha \rightarrow [h_\natural((N, A, \mu[v/[\![\alpha]\!]], \hat{F})\!\downarrow)]_\cong, \varepsilon} \tag{1}$$

$$\frac{\gamma' \in \Gamma_\natural \ s.t. \ h_\natural((N, A, \mu[v/[\![*w]\!]], \hat{F})\!\downarrow) \leq \gamma' \qquad [\![w]\!] \neq nil \qquad \text{noerr}}{[(N, A, \mu, F)]_\cong, v := *w \rightarrow [\gamma']_\cong, \varepsilon} \tag{2}$$

$$\frac{[\![v]\!] \neq nil \qquad [\![\alpha]\!] \neq \bot \qquad \text{noerr}}{[(N, A, \mu, F)]_\cong, *v := \alpha \rightarrow [h_\natural((N, A, \mu[\mu(v)/[\![\alpha]\!]], \hat{F})\!\downarrow)]_\cong, \varepsilon} \tag{3}$$

$$\frac{[\![\alpha]\!] = \bot \ \vee [\![\alpha']\!] = \bot \qquad \text{noerr}}{[\gamma]_\cong, \alpha := \alpha' \rightarrow [\gamma_{\text{err}}]_\cong, \varepsilon} \tag{4}$$

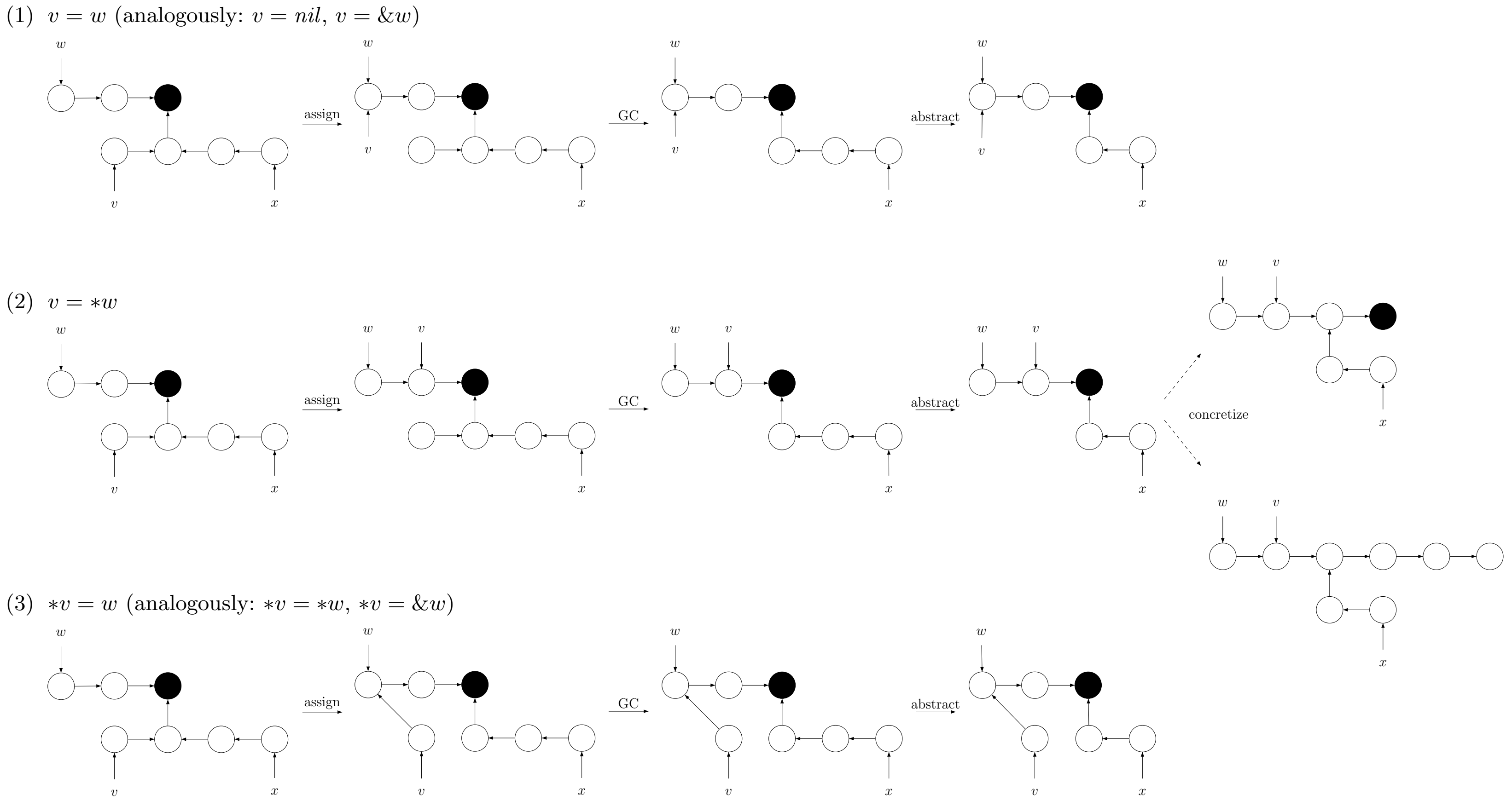

Fig. 3. Exemplary visualization of the abstract semantics ($M = 3$)

In Fig. 3 the semantic rules are visualized for an example configuration. In rule 2 there might be the necessity for both abstraction and concretion. The execution of the assignment and the following abstraction via $h_\natural$ 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 canonical configuration γ' that is at least as concrete as $\bar{\gamma}$ and related by an abstraction morphism to it. There might be more than one solution, thus this rule is nondeterministic (indicated by the dashed arrows), but remains the only source of nondeterminism.

In rules 1 and 3 the distance to an abstract node is not reduced, but the opposite case can occur: just imagine 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 execution of the assignment therefore potentially yields a non–canonical configuration and we have to re–abstract to determine the corresponding canonical configuration. According to Thm. 2 the result is unique and thus these steps are deterministic.

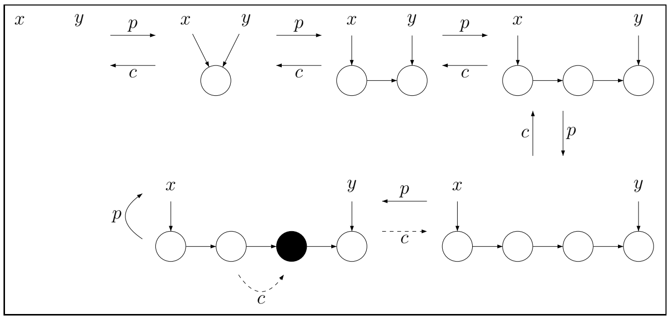

Fig. 4. Producer/Consumer: Abstract State Space ($M = 1$)

Example 3. Figure 4 shows the finite abstract state space of the producer/consumer program from Fig. 1 for $M = 1$. The p– and c–transitions each summarize several producer/consumer steps. The dashed transitions are nondeterministic steps, since the abstract node, visualized in black color, represents at least two nodes in a chain. If now the consumer deletes one node from the beginning of the queue the distance of x to the abstract node becomes two and thus we need to *concretize* the graph to obtain a canonical configuration. For this we distinguish two cases: either the abstract node represents exactly two nodes, then we reach the graph to the right, or it represents more than two, in which case we stay in the same state since the abstract node still represents more than one concrete node.

Theorem 3 (Finiteness). *For every $\pi \in$ LMP, T^a_π is finite.*

The idea of the proof is to establish a bound on the number of nodes of canonical configurations for a given number of program variables.

Theorem 4 (Correctness of the Abstraction). *Let $\pi \in$ LMP. For every transition in T^c_π there exists a corresponding abstract transition in T^a_π such that the heaps are related by abstraction morphisms.*

The proof of the correctness theorem has been omitted due to space constraints.

4 A Logic for Concurrent List–Manipulating Programs

In the previous sections we have defined our programming language for concurrent pointer manipulation and both its concrete and abstract semantics. In this section we will present a logic which will allow us to reason about heap configurations and program behavior. In the following LV denotes a set of logical variables, where we always assume that $LV \cap PV = \emptyset$.

Pointer Logic Pointer logic deals with single configurations and is employed to express graph properties as well as to inspect the special flags of heap configurations (see Def. 1).

Definition 9. *The set PL of* Pointer Logic formulas *is given by the grammar*

$$\begin{array}{ll} \text{NExp} & ::= nil \mid v\ (\in PV) \mid x\ (\in LV) \mid {*}\text{NExp} \\ \text{Atomic} & ::= \text{tt} \mid \text{ff} \mid \text{err} \mid \text{dl} \mid \text{leak} \mid \text{signal} \mid \text{new} \mid \text{NExp} = \text{NExp} \mid \text{NExp} \rightsquigarrow \text{NExp} \\ \text{PL} & ::= \text{Atomic} \mid \neg\text{PL} \mid \text{PL} \wedge \text{PL} \mid \exists x : \text{PL} \end{array}$$

Later on we will use the logical operations $\vee$, $\rightarrow$, $\leftrightarrow$, and $\forall$ (defined as usual) as abbreviations. Note that in contrast to pointer expressions in LM–programs our logic supports dereferencing operations of arbitrary depth. The special operation $\alpha \rightsquigarrow \alpha'$ expresses the reachability of heap objects.

Definition 10. *Let $\beta : LV \rightharpoonup N$ be a variable valuation and $\gamma \in \Gamma_c$ a concrete heap configuration. Then we define $[\![\cdot]\!] : \text{NExp} \rightharpoonup N_{nil}$ by:*

$$\begin{array}{ll} [\![nil]\!] := nil & [\![v]\!] := v \text{ for } v \in PV \\ [\![x]\!] := \beta(x) \text{ for } x \in LV & [\![{*}\alpha]\!] := \mu_\gamma([\![\alpha]\!]) \text{ for } \alpha \in \text{NExp} \end{array}$$

Note the semantic difference with respect to the programming language. In navigation expressions a variable v is interpreted by itself and not by the node it is referencing. This avoids the necessity of the referencing operator &.

Definition 11. *The (concrete) satisfaction relation $\models$ for PL–formulas is given as follows*[4] *(for $\gamma = (N, \emptyset, \mu, F)$):*

$$\begin{array}{ll} \gamma, \beta \models f & \text{iff } f \in F, \text{ where } f \in \{\text{err}, \text{dl}, \text{leak}, \text{signal}, \text{new}\} \\ \gamma, \beta \models \alpha_1 = \alpha_2 & \text{iff } [\![\alpha_1]\!] = [\![\alpha_2]\!] \neq \bot \\ \gamma, \beta \models \alpha_1 \rightsquigarrow \alpha_2 & \text{iff } [\![\alpha_i]\!] \neq \bot \ \wedge\ [\![\alpha_2]\!] \in \mu^\star([\![\alpha_1]\!]) \\ \gamma, \beta \models \exists x : \varphi & \text{iff } \exists n \in N : \gamma, \beta[x/n] \models \varphi \end{array}$$

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 will now extend this logic by temporal operators.

Definition 12. *The set TPL of* Temporal Pointer Logic formulas *is given as follows:*

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

For $\varphi \in \text{TPL}$ we use the abbreviations $\boldsymbol{F}\varphi := \text{tt}\,\boldsymbol{U}\varphi$ and $\boldsymbol{G}\varphi := \neg\boldsymbol{F}\neg\varphi$. Moreover $V(\varphi) \subseteq LV$ denotes the set of (bound or free) logical variables occurring in φ.

[4] For $\wedge, \neg$, tt and ff the semantics is standard and therefore omitted.

Note that it is *not* possible to nest 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. To the best of our knowledge the only approach to support this idea is the one in [Dis03,DKR04,DKR06]; other works in the area such as [SRW02] consider only *shapes* of the heap. This results in a loss of expressivity, e.g., a property like $\forall x : \mathrm{new}(x) \rightarrow \mathbf{F}\ \mathrm{del}(x)$ which states that every produced object will eventually be consumed cannot be formulated. Nonetheless we can specify many interesting properties.

Example 4. For our producer/consumer system from Fig. 1 it holds true:

1. $\neg\mathbf{F}(\mathrm{dl} \vee \mathrm{err})$ (never deadlock or pointer errors)
2. $\mathbf{GF}$ new (new objects are created infinitely often)
3. $\mathbf{G}((*x \neq nil \vee *y \neq nil) \rightarrow (x \rightsquigarrow *y \wedge \forall v : (v \neq y \rightarrow x \rightsquigarrow v)))$
   (whenever the queue is not empty, the object y points to is reachable from x and between x and this object lies a chain)

More general correctness properties are:

4. $\mathbf{F}\ *x = *y$ (x and y will eventually become aliases)
5. $\mathbf{G}\neg(\exists z : (x \rightsquigarrow z \wedge y \rightsquigarrow z))$ (x and y always point to disjoint parts of the heap)
6. $\mathbf{G}(\forall y : (x \rightsquigarrow y \rightarrow (\neg\exists z : (y \rightsquigarrow z \wedge *z \rightsquigarrow y))))$
   (x always points to a non–cyclic list)
7. $\mathbf{FG}(\neg\mathrm{leak})$ (only finitely often a memory leak can occur)
8. $\mathbf{G}(\forall y : (x \rightsquigarrow y \rightarrow (\forall z : (z \rightsquigarrow y \rightarrow x \rightsquigarrow z))))$ (x always points to a chain)

As mentioned before, TPL specifies computation paths. The set of possible paths is represented by a transition system.

Definition 13. *Let $T = (Q, q_0, lab, \rightarrow)$ be a (concrete) transition system with $lab : Q \rightarrow \Gamma_c$. A* path *in T is an infinite sequence of states $\rho = \rho_0\rho_1\rho_2... \in Q^\omega$ such that $\rho_i \rightarrow \rho_{i+1}$ for all $i \in \mathbb{N}$. Then for $\varphi \in PL$ we have*

$$\rho \models \varphi\ (\in \mathrm{PL})\ \textit{iff}\ \exists\beta : LV \rightharpoonup N_{lab(\rho_0)}\ \textit{s.t.}\ lab(\rho_0), \beta \models_{\mathrm{PL}} \varphi$$

For the temporal operators the semantics is identical to the one of LTL. We write $T \models \varphi$ iff $\rho \models \varphi$ for all paths $\rho \in \{q_0\}Q^\omega$ in T.

Reasoning about Abstract Computations As expected the concrete semantics is straightforward. When we switch to abstract configurations, however, 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 *which* concrete node, represented by the summary node, it is bound to. This could lead to undefinedness of Pointer Logic formulas. This problem occurs mainly in direct comparisons of the form $\alpha = \alpha'$. To tackle this problem we choose the global precision constant M in dependence of the formula as follows. If $\varphi \in$ TPL is the formula to check, then we assume from now on that

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

Due to the presence of abstract nodes it is not sufficient anymore to evaluate logical variables by simple variable–to–node mappings. Additionally we must record the offset of a variable referring to an abstract node and the distance between variables pointing to the same abstract node. This leads to the concept of *abstract valuations*.

Given $\gamma \in \Gamma_\natural$ and $\varphi \in \mathrm{TPL}$, an *abstract valuation* is of the form $\eta = (\beta, o, \delta)$, where $\beta : V(\varphi) \to N_\gamma$ maps logical variables to (abstract) nodes, $o : V(\varphi) \to \mathbb{M}$ denotes the offset for an abstract node, and $\delta : V(\varphi) \to V(\varphi) \rightharpoonup \mathbb{M}$ is a "distance matrix" for the logical variables with potentially undefined entries. δ is only defined if both arguments are mapped onto the same entity, and o is only different from 1 if the corresponding variable is mapped onto an abstract node. The set of all such valuations will be denoted by $\mathrm{Val}_{\gamma,\varphi}$.

Using this concept one can define a function $d_{\gamma,\eta} : \mathrm{NExp} \times \mathrm{NExp} \to \{0, 1, \infty\}$ measuring the "distance" of pointer expressions, where distance here means either 0 if the expressions are mapped onto the same (concrete) entity, 1 if the the first case does not hold but the second argument is reachable from the first or ∞ if neither is the case.

The presence of abstract nodes plays a vital role in the abstract semantics. Without the global constraint for M we would not be able to resolve all possible cases of abstract valuations, a third truth value would thus become necessary. The distance function δ is required for the case that both variables are mapped onto an abstract node with offset $\star$. With the help of the distance function the abstract semantics of PL and TPL is straightforward.

Definition 14. *Let $\gamma = (N, A, \mu, F) \in \Gamma_\natural$ and $\eta = (\beta, o, \delta) \in \mathrm{Val}_{\gamma,\varphi}$. The satisfaction relation $\models$ for PL–formulas on canonical configurations is then given as follows (omitting the trivial cases):*

$$\begin{array}{ll}
\gamma, \eta \models f & \textit{iff } f \in F, \textit{ where } f \in \{\mathrm{err}, \mathrm{dl}, \mathrm{leak}, \mathrm{signal}, \mathrm{new}\} \\
\gamma, \eta \models \alpha_1 = \alpha_2 & \textit{iff } d_{\gamma,\eta}(\alpha_1, \alpha_2) = 0 \\
\gamma, \eta \models \alpha_1 \rightsquigarrow \alpha_2 & \textit{iff } d_{\gamma,\eta}(\alpha_1, \alpha_2) \in \{0, 1\} \\
\gamma, \eta \models \exists x : \varphi & \textit{iff } \exists n \in N,\ \textit{off} \in \mathbb{M},\ \textit{dist} : V(\varphi) \rightharpoonup \mathbb{M} \textit{ s.t.} \\
& \quad \gamma, (\beta_\eta[x/n], o_\eta[x/\textit{off}], \delta_\eta[x/\textit{dist}]) \models \varphi
\end{array}$$

Let $T = (Q, q_0, lab, \to)$ be an abstract transition system with $lab : Q \to \Gamma_\natural/_{\cong}$ and $\rho \in Q^\omega$ a path in it. Then $\rho \models \varphi \in \mathrm{PL}$ iff for $\gamma \in lab(\rho_0)$ there exists an $\eta \in \mathrm{Val}_{\gamma,\varphi}$ s.t. $\gamma, \eta \models_{\mathrm{PL}} \varphi$. Temporal operators and Boolean connectives are treated in the standard way. We write $T \models \varphi$ iff $\rho \models \varphi$ for all paths $\rho \in \{q_0\}Q^\omega$ in T.

The following theorem states that the abstract semantics of TPL and of the programming language is correct, i.e., that the validity of a formula under the abstract interpretation implies the validity under the concrete one. The converse though does not hold.

Theorem 5. *Let $\pi \in \mathrm{LMP}$ and $\varphi \in \mathrm{TPL}$. If $T^a_\pi \models \varphi$ then $T^c_\pi \models \varphi$.*

Proof. It suffices to show for all $\varphi \in \mathrm{PL}$ and $\gamma \in \Gamma_c$ the proposition:

$$\exists \beta : LV \rightharpoonup N_\gamma \text{ s.t. } \gamma, \beta \models \varphi \ \Leftrightarrow \ \exists \eta \in \mathrm{Val}_{\gamma,\varphi} \text{ s.t. } h_\natural(\gamma), \eta \models \varphi \qquad (\star)$$

Note that the $\Leftarrow$–direction is sufficient for correctness, the $\Rightarrow$–direction though is trivial. In the proof the choice of the global constant M (depending on the formula) plays a central role. Imagine for example a property "the heap contains at least five objects different from program variables". To formulate this property we need at least five different logical variables and the constraint on M implies that $M \geq 5$. For smaller M it can happen that a formula that is satisfied in the abstract case, does not hold in all concrete configurations associated with the abstract one. E.g. for $M = 1$ and a graph with one abstract node our example property would be satisfied; in the corresponding concrete graph where the abstract node is represented by two concrete nodes not necessarily.

With ($\star$) we can infer from Thm. 4 the validity of the claim, since TPL does not allow path quantifiers. By construction of the abstract PL–semantics it is intuitively clear that ($\star$) holds.

Model Checking Temporal Pointer Logic Because of the two–stage approach in defining the logic, we can reduce the TPL model checking problem to an LTL model checking problem, which can efficiently be verified by existing model checkers.

Algorithm 6. *Let $T = (Q, q_0, lab, \rightarrow)$ be the abstract transition system generated by a program $\pi \in$ LMP and $\varphi \in$ TPL the formula to verify. Let $\Psi := \{\psi \in$ PL $\mid \psi$ maximal subformula of $\varphi\} = \{\psi_1, ..., \psi_r\}$.*

Define a "traditional" transition system $T' = (Q, q_0, lab', \rightarrow)$ where $lab' : Q \rightarrow 2^{AP}$ with $AP = \{p_i \mid i \in \{1, ..., r\}\}$ such that $p_i \in lab'(q) \;\Leftrightarrow\; lab(q) \models \psi_i$.

Now solve the LTL model checking problem $T' \models^?_{\mathrm{LTL}} \varphi[\psi_1/p_1, ..., \psi_r/p_r]$.

The idea is thus to replace all (maximal) PL–subformulas by atomic propositions to obtain an LTL–formula. To do so we first have to evaluate the PL–formulas on the transition system and to change its labeling from configurations to atomic propositions, where each atomic proposition represents the truth value of the corresponding PL–subformula on the given configuration. The correctness of this approach is clear.

Limitations Due to the nondeterminism in the abstract semantics caused by the presence of abstract nodes we may obtain *false negatives*. This means that in the abstract transition system there may exist computations which do not correspond to concrete ones and on which the property to verify does not hold.

Consider a program creating a list (pointed to by v) with $M+3$ elements and then deleting again $M+3$ elements. The property to verify is $\mathbf{XF}(*v = nil)$, i.e. that the list becomes empty. It is obvious that due to the presence of an abstract node after the construction of the list in the abstract semantics there is a path that retains that abstract node and thus the list never becomes empty (see Def. 8, rule 2). In the concrete case however the formula is satisfied.

Due to the overapproximation and the LTL approach false *positives* though cannot occur. This means that the successful verification of a property in the abstract case implies the correctness in the concrete case. False negatives can only occur in cases where information on the precise number of objects is necessary.

5 Application: Concurrent Garbage Collection

In this section we will show we will employ our approach to find counterexamples of a concurrent garbage collection algorithm. More concretely we will consider a so–called *mark–and–sweep* collector, which maintains a bit for each object in the heap to record its reachability status. Here we model this information as an additional heap component, a (partial) function $r : N \rightharpoonup \mathbb{B}$ which indicates whether the collector considers a node to be reachable (1) or not (0). This component is made accessible to the garbage collector program using the additional constructs

- **reset** $\in$ Stmt, which resets the reachability value of every node to 0,
- **mark**$(\alpha) \in$ Stmt where $\alpha \in$ PExp, which sets the reachability information of the node $[\![\alpha]\!]$ to 1, and
- $r(\alpha) \in$ BExp where $\alpha \in$ PExp tests whether the reachability bit of $[\![\alpha]\!]$ is set.

We refrain from giving the formal details of the extended syntax and semantics of LM–programs; these are straightforward to formalize. The only modification we would like to mention explicitly is an adaptation of the automatic garbage collection procedure (cf. Def. 2), which is activated after the execution of every LM–statement which potentially causes nodes to become unreachable (we refer to the derivation rules in Def. 4). To ensure the finiteness of our abstraction, we still have to use it. However, we will adapt the handling of the leak flag such that it will be set only if the garbage collector considers an unreachable node n to be reachable, i.e., if $r(n) = 1$. Formally this means that for an extended configuration $\hat{\gamma} = (N, A, \mu, F, r)$ we define $\hat{\gamma}{\downarrow} := (N', A \cap N', \mu \restriction N', F \cup \{\text{leak} \mid \exists n \in (N \setminus N') : r(n) = 1\}, r \restriction N')$ with $N' = \mu^\star(PV)$.

Using these concepts we can now proceed by describing how a concurrent garbage collector can be added to a given LM–program, called a *mutator*. For $\pi = \textbf{var}\ v_1, \ldots, v_k : (s_1 \| \ldots \| s_l) \in$ LMP, we define $\pi' := \textbf{var}\ v_1, \ldots, v_k, t : (s_1 \| \ldots \| s_l \| c)$ with garbage collector c as in Fig. 5.

Thus the garbage collector is running concurrently with the mutator. It executes an infinite loop, starting by resetting the reachability bit of every node in the heap. Using the auxiliary variable t, it then marks every reachable node, beginning with the roots of the heap which are accessible by the program variables. Here the statement **with** $v \in PV$ **do** s **od** is a meta construct which is expanded to $s[v/v_1]; s[v/v_2]; ...; s[v/v_k]$ for $PV = \{v_1, \ldots, v_k\}$. Whenever it encounters a node which has already been marked (**if** statement), it continues with the next program variable to avoid redundant assignments. Finally it employs the signaling mechanism of our programming language to indicate that now the actual collection phase would start, i.e., that all nodes whose reachability bit is 0 would be removed.

```
while tt do
  reset;
  with v ∈ PV do
    t := v;
    while t ≠ nil do
      if r(t) then t := nil
            else  mark(t);
                  t := *t
      fi
    od
  od;
  signal
od
```

Fig. 5. A naive garbage collector

Note, however, that we are still using our automatic garbage collection procedure such that we can guarantee that in every configuration of the system, all nodes are reachable. In other words, whenever the signal occurs there should not exist any unmarked node in the heap. This observation is the key idea for specifying the *soundness* of the garbage collector c as a safety property in TPL. Here we assume that the underlying Pointer Logic (cf. Def. 9) is extended by atomic propositions of the form $r(\alpha)$ which allow us test the reachability information of the node to which the navigation expression α refers:

$$\mathbf{G}(\text{signal} \rightarrow \forall x : r(x))$$

Another important issue is the *completeness* of the garbage collector, which means that every node which has become unreachable in the course of the computation, will eventually be removed. This, however, cannot be directly expressed for two reasons. First, verifying this property would require to keep track of the identity of objects between different configurations, which in turn involves the nesting of quantifiers and temporal operators. This is not supported by our logic. Second, our automatic garbage collection procedure immediately removes nodes that have become unreachable.

What we can formulate instead, however, is a safety property which comes very close to the actual completeness. It expresses that a node which has become unreachable will never be marked by the garbage collector. Employing the modified handling of the leak flag, this property can simply be formulated as

$$\mathbf{G}\ \neg\text{leak}$$

Note that this formalization is only justified since the garbage collector is monotonic in the following sense: once a node has been marked, its reachability information will not be reset before the collection signal occurs. Moreover completeness can only be expected (just as the above soundness property) if it is guaranteed that the mutator does not modify the reachability bits.

The example computation in Fig. 6 shows that the above garbage collector violates both of these requirements. Here the mutator program is assumed to be of the form $y := *x;\ *x := *y;\ *y := nil;\ y := nil$; it simply discards the second node of the list whose head is referenced by x (assuming that this node exists). Here C and M stand for operations of the collector and the mutator, respectively, which are either concretely given or summarized by a "+" sign. The bits labeling the nodes indicate the reachability information as set by the collector.

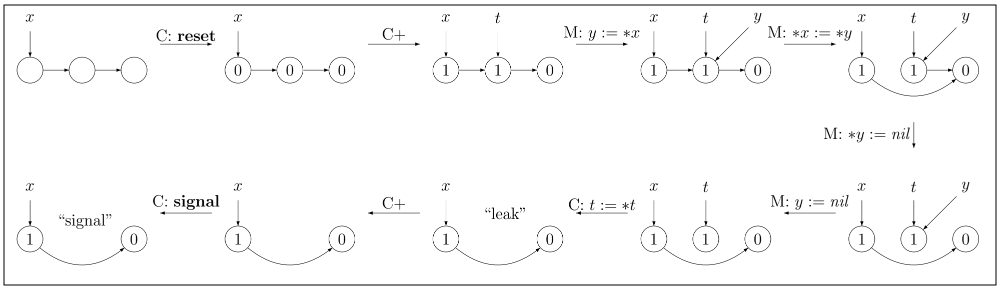

Fig. 6. Possible erroneous run of garbage collector and mutator

The computation shows that the collector is neither correct nor complete. In the final step involving the signal flag, the reachability value 0 of the list's tail node means that it would be removed by the collector although it is reachable. Two steps earlier, the leak flag indicates that garbage has automatically been deleted which has been marked as reachable by the collector. Both of these problems are caused by the uncontrolled interaction between the mutator and the collector; they can be avoided by placing the body of the collector loop in an atomic region.

6 Related Work

Related work on the topic of analyzing pointer–manipulating programs can be classified into the following (often overlapping) categories.

Predicate abstraction abstracts the state space of the program by evaluating it under a number of given predicates. This yields a Boolean program which conservatively simulates all potential executions [GS97]. Successful software model checkers such as BLAST [HJMS03] and SLAM [BR02] are based on this approach. There are several papers that use classical predicate abstraction for pointer analysis [BPZ05,DN03]. In particular, [DD01,DDP99] study concurrent garbage collection using predicate abstraction.

Shape analysis is a static analysis framework that represents recursive data structures of unbounded size by finite structures, called "shape graphs". The idea is to apply to the heap the same abstraction that is applied to the program's states in predicate abstraction: it is defined in terms of equivalence classes of heap objects that are induced by a finite set of predicates on those objects. The usual approach is to formalize shape graphs by three–valued logical structures [SRW02]. This approach has been implemented TVLA [LAS00] and in BLAST [BHT06] which makes use of TVLA.

Recent developments comprise adaptive methods which automatically adjust to the data structures that occur in the given program [LYY05,LRS05,YR04], demand–driven techniques [BHT06,HT01], efficiency improvements [LAIS06], and interprocedural shape analysis [HR05,JLRS04,RBR$^+$05,RSY05].

It is often argued that the application of predicate abstraction to pointer structures does not work well because it is difficult to find predicates which abstract heap structures in an appropriate and compact way [BHT06]. This claim is substantiated by the results in [MYRS05] which investigates the application of both predicate abstraction and shape analysis to programs operating on singly–linked lists, employing a similar abstraction as ours: elements on unshared list segments are summarized. It is shown that standard predicate abstraction requires an exponential number of predicates in comparison to the number of predicates in shape analysis. Also [PW05] considers both techniques, but in a very restricted programming–language setting which only supports single assignments.

Regular model checking is a framework for unified verification of infinite–state systems based on automata theory. It represents states using words (trees) over a finite alphabet and sets of states using finite (tree) automata [BHRV06]. Like in our approach, singly–linked lists are also considered in [BBH$^+$06,BHMV05], but only safety and termination properties are verified.

Dataflow analysis is a technique for gathering information about certain aspects of a program using its control flow graph. This approach is generally efficient but restricted to rather shallow properties of programs such as aliasing relations [Deu94,NKmWH04], points–to information [WL02,ZC04], or pointer range analysis [YH04].

Hoare–style approaches: first–order reasoning typically breaks down when it comes to prove properties of pointer–manipulating programs. The main reason is that it is impossible to express an invariant of all members of a data structure in first–order logic. The latter has to be extended therefore to support the definition of a reachability predicate [ABB06,BIL04,FGM96,LAIR$^+$05,MS01,Nel83]. However such deductive techniques usually involve user interaction, or otherwise only restricted properties such as dereferencing of nil pointers or aliasing effects can be analyzed.

Separation logic has been proposed as an extension to Hoare logic that permits local reasoning about linked structures, supporting features to support modular correctness proofs for pointer–manipulating programs [OYR04,Rey02]. It has been employed for termination proofs of heap–manipulating programs [BCDO06], for interprocedural shape analysis [GBC06], for handling abstract data types [BCOP05], and for verifying garbage collection algorithms [BTSR04]. However most of the work on separation logic focuses on verifying programs manually.

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, the introduction of abstract entities which represent a potentially unbounded number of heap cells (called "summary nodes" in [CWZ90]); see e.g. [BPZ05,BHMV05,MYRS05], and the observation that, in this setting, the number of sharing points in heap structures is bounded by the number of program variables [BHMV05,MYRS05].

However none of these combines the strengths of our approach which supports concurrent programs with dynamic memory allocation and destructive updates such that arbitrary (cyclic) linked lists can be constructed, integrates both abstraction and model checking in a fully automated way, supports a linear–time logic in which both safety and liveness properties can be expressed, and which allows to use standard LTL model checkers.

In comparison, many of the existing approaches suffer from the poor programming environment, the exclusion of cyclic data structures, the requirement of user interaction, or the restriction to safety properties. Notable exceptions are [BPZ05], which also offers liveness properties but requires user–defined ranking functions, [DKR06], which employs extended tableau–based techniques for model checking, and [YRSW03], which has a non–standard interpretation.

7 Conclusions and Future Work

We have presented a framework for the verification of concurrent pointer–manipulating programs with unbounded heap size and destructive updates. The correctness properties are specified using temporal pointer logic which is essentially pointer logic for expressing heap properties enriched with temporal operators. Instead of requiring dedicated algorithms, the TPL model checking problem is reduced to an LTL model checking problem that can be verified effectively with a broad variety of existing model checkers. The tradeoff is the restriction to list–like data structures as well as the limitation in expressiveness of the logic because object identities are not tracked between configurations.

Currently we are implementing our method to verify more realistic examples in the future. In particular we will extend the analysis of concurrent garbage collectors by defining a "hardest mutator", i.e., a general mutator program which is capable of simulating the behavior of any other mutator. This will enable us to establish the correctness of garbage collectors independent of the concrete mutator.

Furthermore due to the extensive use of concurrency, state space reduction and optimization techniques such as partial order reduction [FG05,God96] will have to be employed and integrated in the implementation. We also plan to extend our framework with dynamic (unbounded) creation of threads. Another interesting aspect could be the combination of existing finite–state modeling languages like Promela [Hol03] and pointer manipulation. Finally in the long run we have plans to increase the expressivity of the logic as well as to generalize our approach to richer data structures, for which new abstractions will be necessary. Moreover an automata–theoretic approach to defining a storeless semantics, as it is studied in [BIL03] for a (concrete) semantics for pointer programs seems promising.

References

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object–oriented programs. In *POPL '06*, pages 91–102. ACM Press, 2006.

[BBH+06] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In *CAV '06*, volume 4144 of *LNCS*, pages 517–531. Springer–Verlag, 2006.

[BCDO06] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape–shifting heaps. In *CAV '06*, volume 4144 of *LNCS*, pages 386–400. Springer–Verlag, 2006.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL '05*, pages 259–270. ACM Press, 2005.

[BHMV05] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1–selector-linked list structures in regular model checking. In *TACAS '05*, volume 3440 of *LNCS 3440*, pages 13–29. Springer–Verlag, 2005.

[BHRV06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *SAS '06*, volume 4134 of *LNCS*, pages 52–70. Springer–Verlag, 2006.

[BHT06] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy shape analysis. In *CAV '06*, volume 4144 of *LNCS*, pages 532–546. Springer–Verlag, 2006.

[BIL03] Marius Bozga, Radu Iosif, and Yassine Lakhnech. Storeless semantics and alias logic. *ACM SIGPLAN Not.*, 38(10):55–65, 2003.

[BIL04] Marius Bozga, Radu Iosif, and Yassine Lakhnech. On logics of aliasing. In *SAS '04*, volume 3148 of *LNCS*, pages 344–360. Springer–Verlag, 2004.

[BPZ05] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape analysis by predicate abstraction. In *VMCAI '05*, volume 3385 of *LNCS*, pages 164–180. Springer–Verlag, 2005.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02*, pages 1–3. ACM Press, 2002.

[BTSR04] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In *POPL '04*, pages 220–231. ACM Press, 2004.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90*, pages 296–310. ACM Press, 1990.

[DD01] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *LICS '01*, pages 51–58. IEEE, 2001.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *CAV '99*, volume 1633 of *LNCS*, pages 160–171. Springer–Verlag, 1999.

[Deu94] Alain Deutsch. Interprocedural may–alias analysis for pointers: beyond k–limiting. In *PLDI '94*, pages 230–241. ACM Press, 1994.

[Dis03] Dino Distefano. *On Model Checking the Dynamics of Object–Based Software: a Foundational Approach.* PhD thesis, Univ. of Twente, 2003.

[DKR04] D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? – on the automated verification of linked list structures. In *FSTTCS '04*, volume 3328 of *LNCS*, pages 250–262. Springer–Verlag, 2004.

[DKR06] D. Distefano, J.-P. Katoen, and A. Rensink. Safety and liveness in concurrent pointer programs. In *FMCO '06*, volume 4111 of *LNCS*, pages 280–312. Springer–Verlag, 2006.

[DN03] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI '03*, volume 2575 of *LNCS*, pages 310–323. Springer–Verlag, 2003.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial–order reduction for model checking software. In *POPL '05*, pages 110–121. ACM Press, 2005.

[FGM96] P. Fradet, R. Gaugne, and D. Le Métayer. Static detection of pointer errors: an axiomatisation and a checking algorithm. In *ESOP '96*, volume 1058 of *LNCS*, pages 125–140. Springer–Verlag, 1996.

[GBC06] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS '06*, volume 4134 of *LNCS*, pages 240–260. Springer–Verlag, 2006.

[God96] Patrice Godefroid. *Partial–Order Methods for the Verification of Concurrent Systems: An Approach to the State–Explosion Problem*, volume 1032 of *LNCS*. Springer–Verlag, 1996.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *CAV '97*, volume 1254 of *LNCS*, pages 72–83. Springer–Verlag, 1997.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *SPIN '03*, volume 2648 of *LNCS*, pages 235–239. Springer–Verlag, 2003.

[Hol03] G.J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison–Wesley, 2003.

[HR05] Brian Hackett and Radu Rugina. Region–based shape analysis with tracked locations. In *POPL '05*, pages 310–323. ACM Press, 2005.

[HT01] Nevin Heintze and Olivier Tardieu. Demand–driven pointer analysis. *ACM SIGPLAN Not.*, 36(5):24–34, 2001.

[JLRS04] Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. A relational approach to interprocedural shape analysis. In *SAS '04*, volume 3148 of *LNCS*, pages 246–264. Springer–Verlag, 2004.

[LAIR+05] Tal Lev-Ami, Neil Immerman, Thomas W. Reps, Shmuel Sagiv, S. Srivastava, and Greta Yorsh. Simulating reachability using first–order logic with applications to verification of linked data structures. In *CADE '05*, volume 3632 of *LNCS*, pages 99–115. Springer–Verlag, 2005.

[LAIS06] Tal Lev-Ami, Neil Immerman, and Shmuel Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV '06*, volume 4144 of *LNCS*, pages 547–561. Springer–Verlag, 2006.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *SAS '00*, volume 1824 of *LNCS*, pages 280–302. Springer–Verlag, 2000.

[LRS05] Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. Abstraction refinement via inductive learning. In *CAV '05*, volume 3576 of *LNCS*, pages 519–533. Springer–Verlag, 2005.

[LYY05] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP '05*, volume 3444 of *LNCS*, pages 124–140. Springer–Verlag, 2005.

[MS01] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01*, pages 221–231. ACM Press, 2001.

[MYRS05] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly–linked lists. In *VMCAI '05*, volume 3385 of *LNCS*, pages 181–198. Springer–Verlag, 2005.

[Nel83] Greg Nelson. Verifying reachability invariants of linked structures. In *POPL '83*, pages 38–47. ACM Press, 1983.

[NKmWH04] Erik M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Bottom–up and top–down context–sensitive summary–based pointer analysis. In *SAS '04*, volume 3148 of *LNCS*, pages 165–180. Springer–Verlag, 2004.

[OYR04] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *POPL '04*, pages 268–280. ACM Press, 2004.

[PW05] Andreas Podelski and Thomas Wies. Boolean heaps. In *SAS '05*, volume 3672 of *LNCS*, pages 268–283. Springer–Verlag, 2005.

[RBR+05] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL '05*, pages 296–309. ACM Press, 2005.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02*, pages 55–74. IEEE Computer Society, 2002.

[RSY05] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint–free programs. In *SAS '05*, volume 3672 of *LNCS*, pages 284–302. Springer–Verlag, 2005.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape–analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, 1998.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[WL02] John Whaley and Monica S. Lam. An efficient inclusion–based points–to analysis for strictly–typed languages. In *SAS '02*, volume 2477 of *LNCS*, pages 180–195. Springer–Verlag, 2002.

[YH04] Suan Hsi Yong and Susan Horwitz. Pointer–range analysis. In *SAS '04*, volume 3148 of *LNCS*, pages 133–148. Springer–Verlag, 2004.

[YR04] Eran Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *PLDI '04*, pages 25–34. ACM Press, 2004.

[YRSW03] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *ESOP '03*, volume 2618 of *LNCS*, pages 204–222. Springer–Verlag, 2003.

[ZC04] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *PLDI '04*, pages 145–157. ACM Press, 2004.

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from http://aib.informatik.rwth-aachen.de/. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2001-01 *	Jahresbericht 2000
2001-02	Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03	Thierry Cachat: The power of one-letter rational languages
2001-04	Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05	Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages
2001-06	Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07	Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem
2001-08	Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09	Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11	Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 *	Jahresbericht 2001
2002-02	Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03	Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages
2002-04	Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05	Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06	Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07	Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08	Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09	Markus Mohnen: Interfaces with Default Implementations in Java
2002-10	Martin Leucker: Logics for Mazurkiewicz traces
2002-11	Jürgen Giesl, Hans Zantema: Liveness in Rewriting
2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations

2003-04 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs

2003-05 Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard

2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates

2003-07 Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung

2003-08 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs

2004-01 * Fachgruppe Informatik: Jahresbericht 2003

2004-02 Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting

2004-04 Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming

2004-05 Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming

2004-06 Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming

2004-07 Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination

2004-08 Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

2004-09 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10 Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: “Aachen Summer School Applied IT Security”

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-01 * Fachgruppe Informatik: Jahresbericht 2005

2006-02 Michael Weber: Parallel Algorithms for Verification of Large Systems

2006-03 Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation

2006-05 Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09 Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking

2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed

2006-11 Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers

2006-12 Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning

2006-13 Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities

2006-14 Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"

2006-15 Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices

2006-16 Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness

2006-17 Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines

2007-01 * Fachgruppe Informatik: Jahresbericht 2006

2007-02 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations

2007-03 Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase

2007-04 Jan Buchholz, Eric Lee, Jonathan Klein, Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation

2007-05 Uwe Naumann: On Optimal DAG Reversal

* These reports are only available as a printed version.

Please contact biblio@informatik.rwth-aachen.de to obtain copies.