Decision Procedures for the Temporal Verification of Concurrent Lists

Alejandro Sánchez[1] and César Sánchez[1,2]

[1] The IMDEA Software Institute, Madrid, Spain
[2] Spanish Council for Scientific Research (CSIC), Spain
{alejandro.sanchez,cesar.sanchez}@imdea.org

Abstract. This paper studies the problem of formally verifying temporal properties of concurrent datatypes. Concurrent datatypes are implementations of classical data abstractions, specially designed to exploit the parallelism available in multiprocessor architectures. The correctness of concurrent datatypes is essential for the overall correctness of the client software. The main difficulty to reason about concurrent datatypes is due to the simultaneous use of unstructured concurrency and dynamic memory.
The first contribution of this paper is the use of deductive temporal verification methods, in particular verification diagrams, enriched with reasoning about dynamic memory. Proofs using verification diagrams are decomposed into a finite collection of verification conditions. Our second contribution is a decision procedure mixing memory regions, pointers and lisp-like lists with locks, that allows the automatic verification of the generated verification conditions. We illustrate our techniques proving safety and liveness properties of lock-coupling concurrent lists.

1 Introduction

Concurrent data structures [5] are an efficient approach to exploit the parallelism of multiprocessor architectures. In contrast with sequential implementations, concurrent datatypes allow the simultaneous access of many threads to the memory representing the data value of the concurrent datatype. Concurrent data structures are hard to design, difficult to implement correctly and even more difficult to formally prove correct.

The main difficulty in reasoning about concurrent datatypes comes from the interaction of concurrency and heap manipulation. The most popular technique to reason about the structures in the heap is separation logic [10]. Leveraging on this success, some researchers [6, 13] have extended this logic to deal with concurrent programs. However, in separation logic disjoint regions are implicitly declared (hidden in the separation conjunction), which makes the reasoning about unstructured concurrency more cumbersome.

In this paper, we propose a complementary approach. We start from temporal deductive verification in the style of Manna-Pnueli [7], in particular using general verification diagrams [4, 11] to deal with concurrency. Then, inspired by regional logic [1], we enrich the state predicate language to reason about the different regions in the heap that a program manipulates. Finally, we build decision procedures capable of checking all generated verification conditions generated during our proofs, to aid in the automation of the verification process.

Explicit regions allow the use of a classical first-order assertion language to reason about heaps, including mutation and disjointness of memory regions. Regions correspond to finite sets of object references. Unlike separation logic, the theory of sets [14] can be easily combined with other classical theories to build more powerful decision procedures. Classical theories are also amenable of integration into SMT solvers [2]. Moreover, being a classical logic one can use classical Assume-Guarantee reasoning, for example McMillan proof rules [8], for reasoning compositionally about liveness properties. In practice, using explicit regions requires the annotation and manipulation of ghost variables of type *region*, but adding these annotations is usually straightforward.

Verification diagrams can be understood as an intuitive way to abstract the specific aspect of a program which illustrates why the program satisfies a given temporal property We propose the following verification process to show that a datatype satisfies a property expressed in linear temporal logic. First, we build the *most general client* of the datatype, parametrized by the number of threads. Then, we annotate the client and datatype with ghost fields and ghost code to support the reasoning, if necessary. Second, we build a verification diagram that serves as a witness of the proof that all possible parallel executions of the program satisfy the given temporal formula.

The proof is checked in two phases. First, we check that all executions abstracted by the diagram satisfy the property, which can be solved through a fully-automatic finite state model checking method. Second, we must check that the diagram does in fact abstract the program, which reduces to verifying a collection of verification conditions, generated from the diagram. Each concurrent datatype maintains in memory a collection of nodes and pointers with a particular layout. Based on this fact, we propose to use an assertion language whose terms include predicates in specific theories for each layout. For instance, in the case of singly linked lists, we use a decision procedure capable of reasoning about ideal lists as well as pointers representing lists in memory. In this paper, we build a decision procedure extending the theory of linked lists [9] with locks. We illustrate the whole approach to prove thread termination on a simple implementation of concurrent lists.

Most previous approaches to verifying concurrent datatypes are restricted to safety properties. In comparison, the method we propose can be used to prove *all liveness* properties, relying on the completeness of verification diagrams.

The rest of the paper is structured as follows. Section 2 presents the running example: lock-coupling concurrent lists. Section 3 briefly introduces verification diagrams and explicit regions. Section 4 describes the proposed decision procedure for concurrent lists. Finally, Section 5 shows how to apply our approach to prove termination in one case of concurrent lists. Some proofs are missing due to space limitations.

2 Concurrent Lock-Coupling Lists

The running example in this paper is the verification of lock-coupling concurrent lists [5, 13]. Lock-coupling concurrent lists are ordered lists with non-repeating elements, in which each node is protected by a lock. A thread advances through the list acquiring the lock of the node it visits. This lock is only released after the lock of the next node has been acquired. The *List* and *Node* structures, shown in Fig. 1(a) are used to maintain the data of a concurrent list.

A *List* contains one field pointing to the *Node* representing the head of the list. A *Node* consists of a value, a pointer to the next *Node* in the list and a lock. We assume that the operating system provides the operations *lock* and *unlock* to acquire and release a lock. Every list has two sentinel nodes, *Head* and *Tail*, with phantom values representing the lowest and highest possible values. For simplicity, we assume such nodes cannot be removed or modified. Concurrent Lock-Coupling Lists are used to implement sets, so they offer three operations:

– *locate*, shown in Fig. 1(d), finds an element traversing the list. This operation returns the pair consisting of the desired node and the node that precedes it in the list. If the element is not found the *Tail* node is returned as the

```
class List {
    Node list;
}

class Node {
    Value val;
    Node next;
    Lock lock;
}
```

(a) data structures

```
while true do
    $e := NondetPickElem$
    nondet
        [ call $search(e)$
          or
          call $add(e)$
          or
          call $remove(e)$ ]
end while
```

(b) *decide*

```
$prev, curr := locate(e)$
if $curr.val = e$ then
    $result := true$
else
    $result := false$
end if
$curr.unlock()$
$prev.unlock()$
return $result$
```

(c) *search*

```
$prev := Head$
$prev.lock()$
$curr := prev.next$
$curr.lock()$
while $curr.val < e$ do
    $prev.unlock()$
    $prev := curr$
    $curr := curr.next$
    $curr.lock()$
end while
return $(prev, curr)$
```

(d) *locate*

```
$prev, curr := locate(e)$
if $curr.val \neq e$ then
    $aux :=$ new $Node(e)$
    $aux.next := curr$
    $prev.next := aux$
    $result := true$
else
    $result := false$
end if
$prev.unlock()$
$curr.unlock()$
return $result$
```

(e) *add*

```
$prev, curr := locate(e)$
if $curr.val = e$ then
    $aux := curr.next$
    $prev.next := aux$
    $result := true$
else
    $result := false$
end if
$prev.unlock()$
$curr.unlock()$
return $result$
```

(f) *remove*

Fig. 1: Data structure and algorithms for concurrent lock-coupling list

current node. A search operation, shown in Fig. 1(c), that decides whether an element is in the list can be easily extended from *locate*.

– *add*, shown in Fig. 1(e), inserts a new element in the list, using *locate* to determine the position at which the element must be inserted. The operation *add* returns *true* upon success, otherwise it returns *false*.
– *remove*, in Fig. 1(f), deletes a node from the list by redirecting the next pointer of the previous node appropriately.

Fig.1(b) shows the most general client of the concurrent-list datatype: the program *decide* that repeatedly chooses non-deterministically a method and its parameters. We construct a fair transition system $\mathcal{S}[N]$ parametrized by the total number of threads N, in which all threads run *decide*. Let ψ be the temporal formula that describes that the thread which holds the last lock in the list terminates. The verification problem is then casted as $\mathcal{S}[N] \vDash \psi$, for all N.

A sketch of a verification diagram is depicted in Fig. 2. We say that a thread is the rightmost owning a lock when there is no other thread owning a lock that protects a *Node* closer to the tail. Each diagram node is labeled with a predicate. This predicate captures the set of states of the transition system that the node abstracts. Edges represent transitions between states abstracted by the nodes.

Checking the proof represented by the verification diagram requires two activities. First, show that all traces of the diagram satisfy the temporal formula ψ, which can be performed by finite state model checking. Second, prove that all computations of $\mathcal{S}[N]$ are traces of the verification diagram. This process involves the verification of formulas built from several theories. For instance, considering the execution of line 5 of program *add* we should verify that the following condition holds:

$$at\_add_5^{[k]} \wedge IsLast(k) \wedge \begin{pmatrix} r' = r \cup \langle aux^{[k]} \rangle \quad \wedge \\ prev'^{[k]}.next = aux^{[k]} \end{pmatrix} \rightarrow at'\_add_6^{[k]} \wedge IsLast'(k)$$

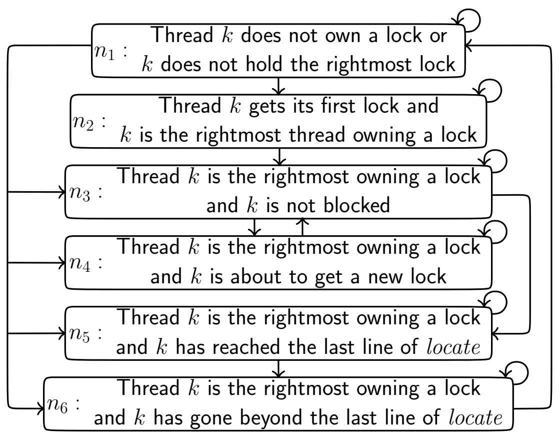
n_1 : Thread k does not own a lock or k does not hold the rightmost lock
n_2 : Thread k gets its first lock and k is the rightmost thread owning a lock
n_3 : Thread k is the rightmost owning a lock and k is not blocked
n_4 : Thread k is the rightmost owning a lock and k is about to get a new lock
n_5 : Thread k is the rightmost owning a lock and k has reached the last line of *locate*
n_6 : Thread k is the rightmost owning a lock and k has gone beyond the last line of *locate*

Fig. 2: Sketched verification diagram for $\mathcal{S}[N] \vDash \psi$

The predicate $\mathit{prev}'^{[k]}.\mathit{next} = \mathit{curr}^{[k]}$ is in the theory of pointers, while $r' = r \cup \langle \mathit{curr}^{[k]} \rangle$ is in the theory of regions. Moreover, some predicates belong to a combination of theories, like $\mathit{IsLast}(k)$, which among other things establishes that $\mathit{List}\,(h, x, r)$ holds. $\mathit{List}\,(h, x, r)$ expresses that in heap h, starting from pointer x, the pointers form a list of elements following the *next* field, and that all nodes in this list form precisely the region r.

The construction of a verification diagram is a manual task, but it often follows the programmer's intuitive explanation of why the property holds. The activity that we want to automate is checking that the diagram indeed proofs the property. To accomplish this automation we must build a suitable decision procedure involving many theories, which we describe in the rest of the paper.

3 Preliminaries

We describe the temporal properties of interest in linear temporal logic, using operators such as $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next) or $\mathcal{U}$ (until) in conjunction with classical logic operations. The state predicates are built from the combination of theories that we present here.

Explicit Regions We use explicit regions to represent the manipulation of memory during the execution of the system. This reasoning is handled by extending the program code with ghost variables of type **rgn**, and ghost updates of these variables. Variables of type **rgn** represent finite sets of object references stored in the heap. Regional logic [1] provides a rich set of language constructs and assertions. However, it is enough for our purposes to use only a small fragment of regional logic. The term **emp** denotes the empty region and $\langle x \rangle$ represents the singleton region whose only object is the one referenced by x. Traditional set-like operators such as $\cup$, $\cap$ and $\setminus$ are also provided and can be applied to **rgn** variables. The assertion language allows reasoning involving mutation and separation. Given two **rgn** expressions r_1 and r_2 we can assert whether they are equal ($r_1 = r_2$), one is contained into the other ($r_1 \subseteq r_2$) or they are completely disjoint ($r_1 \# r_2$).

Verification Diagrams We sketch here the important notions from [4, 11]. Verification diagrams provide an intuitive way to abstract temporal proofs over fair transition systems (FTS). A FTS Φ is a tuple $\langle \mathcal{V}, \Theta, \mathcal{T}, \mathcal{J} \rangle$ where $\mathcal{V}$ is a finite set of variables, Θ is an initial assertion, $\mathcal{T}$ is a finite set of transitions and $\mathcal{J} \subseteq \mathcal{T}$ contains the fair transitions (in this paper we will not discuss strong fairness). A *state* is an interpretation of $\mathcal{V}$. We use $\mathbf{S}$ to denote the set of all possible states. A transition $\tau \in \mathcal{T}$ is a function $\tau : \mathbf{S} \to 2^{\mathbf{S}}$, which is usually represented by a first-order logic formula $\rho_\tau(s, s')$ describing the relation between the values of the variables in a state s and in a successor state s'. Given a transition τ, the state predicate $\mathit{En}(\tau)$ denotes whether there exists a successor state s' such that $\rho_\tau(s, s')$.

A computation of Φ is an infinite sequence of states such that (a) the first state satisfies Θ; (b) any two consecutive states satisfy ρ_τ for some $\tau \in \mathcal{T}$;

(c) for each $\tau \in \mathcal{J}$, if τ is continuously enabled after some point, then τ is taken infinitely many times. We use $\mathcal{L}(\Phi)$ to denote the set of computations of the FTS Φ. Given a formula φ, $\mathcal{L}(\varphi)$ denotes the set of sequences satisfying φ. A FTS Φ satisfies a temporal formula φ if all computations of Φ satisfy φ, i.e., $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\varphi)$.

A verification diagram (VD) $\Psi : \langle N, N_0, E, \mu, \eta, \mathcal{F}, \Delta, f\rangle$ is a formula automaton with components:

- N is a finite set of nodes.
- $N_0 \subseteq N$ is the set of initial nodes.
- $E \subseteq N \times N$ is a set of edges.
- $\mu : N \to F(V)$ is a labeling function mapping nodes to assertions over V.
- $\eta : E \to 2^{\mathcal{T}}$ is a labeling function assigning sets of transitions to edges.
- $\mathcal{F} \subseteq 2^{E\times E}$ is an edge acceptance set of the form $\{(P_1, R_1), \ldots, (P_m, R_m)\}$.
- $\Delta \subseteq \{\delta | \delta : \mathbf{S} \to \mathcal{D}\}$ is a set of ranking functions from states to a well founded domain $\mathcal{D}$.
- f maps nodes into propositional formulas over atomic subformulas of φ.

If $n \in N$ then we use *next* (n) to denote the set $\{\tilde{n} \in N | (n, \tilde{n}) \in E\}$ and $\tau(n)$ for $\{\tilde{n} \in \mathit{next}(n) | \tau \in \eta(n, \tilde{n})\}$. For each $(P_j, R_j) \in \mathcal{F}$ and for each $n \in N$, Δ contains a ranking function $\delta_{j,n}$. An infinite sequence of nodes $\pi = n_0, n_1, \ldots$ is a path if $n_0 \in N_0$ and for each $i > 0$, $(n_i, n_{i+1}) \in E$. A path π is accepted if for each pair $(P_j, R_j) \in \mathcal{F}$ some edges of R_j occur infinitely often in π or all edges that occur infinitely often in π are also in P_j. An infinite path π is fair when, for any just transition τ, if τ is enabled on all nodes that appear infinitely often in π then τ is taken infinitely often.

Given a sequence of states $\sigma = s_0, s_1, \ldots$ of Φ, a path $\pi = n_0, n_1, \ldots$ is a trail of σ whenever $s_i \vDash \mu(n_i)$ for all $i \geq 0$. An infinite sequence of states σ is a computation of Ψ whenever there exists an accepting trail of σ such that is also fair. $\mathcal{L}(\Psi)$ is the set of computations of Ψ.

A verification diagram shows that $\Phi \vDash \varphi$ via the inclusions $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi)$. The map f is used to check $\mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi)$. To show $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\Psi)$ it is enough to prove the following verification conditions:

- *Initiation:* at least one initial node from N_0 satisfies the initial condition of the fair transition system Φ.
- *Consecution:* for every node $n \in N$ and transition $\tau \in \mathcal{T}$,
$$\mu(n)(s) \wedge \rho_\tau(s, s') \to \mu(\mathit{next}(n))(s').$$
- *Acceptance:* for each $(P_j, R_j) \in \mathcal{F}$, if $(n_1, n_2) \in P_j \setminus R_j$ then
$$\rho_\tau(s, s') \wedge \mu(n_1)(s) \wedge \mu(n_2)(s') \to \delta_{j,n_1}(s) \succeq \delta_{j,n_2}(s')$$
and if $(n_1, n_2) \notin P_j \cup R_j$ then
$$\rho_\tau(s, s') \wedge \mu(n_1)(s) \wedge \mu(n_2)(s') \to \delta_{j,n_1}(s) \succ \delta_{j,n_2}(s')$$
- *Fairness:* For each $e = (n_1, n_2) \in E$ and $\tau \in \eta(e)$:
  1. τ is guaranteed to be enabled in every $\mu(n_1)(s)$.
  2. Any τ-successor of a state satisfying $\mu(n_1)$ satisfies the label of some node in $\tau(n)$.

4 Building a Suitable Decision Procedure

The automatic check of the proof represented by a verification diagram requires decision procedures to verify the generated verification conditions. These decision procedures must deal with formulas containing terms belonging to different theories. In particular, for concurrent lists the decision procedure must reason about pointer data structures with a list layout, regions and locks. To obtain a suitable decision procedure, we extend the Theory of Linked Lists (TLL) [9], a decidable theory including reachability of list-like structures. However, this theory lacks the expressivity to describe locked lists of cells, a fundamental component in our proofs.

We begin with a brief description of the basic notation and concepts. A signature Σ is a triple (S, F, P) where S is a set of sorts, F a set of functions and P a set of predicates. If $\Sigma_1 = (S_1, F_1, P_1)$ and $\Sigma_2 = (S_2, F_2, P_2)$ are two signatures, we define their union $\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2, F_1 \cup F_2, P_1 \cup P_2)$. Similarly we say that $\Sigma_1 \subseteq \Sigma_2$ when $S_1 \subseteq S_2$, $F_1 \subseteq F_2$ and $P_1 \subseteq P_2$. If $t(\varphi)$ is a term (resp. formula), then we denote with $V_\sigma(t)$ (resp. $V_\sigma(\varphi)$) the set of variables of sort σ occurring in t (resp. φ).

A Σ-interpretation is a map assigning a value to each symbol in Σ. A Σ-structure is a Σ-interpretation over an empty set of variables. A Σ-formula over a set X of variables is satisfiable whenever it is true in some Σ-interpretation over X. Let Ω be an interpretation, $\mathcal{A}$ a Ω-interpretation over a set $V$of variables, $\Sigma \subseteq \Omega$ and $U \subseteq V$. $\mathcal{A}^{\Sigma,U}$ denotes the interpretation obtained from $\mathcal{A}$ restricting it to interpret only the symbols in Σ and the variables in U. We use $\mathcal{A}^{\Sigma}$ to denote $\mathcal{A}^{\Sigma,\emptyset}$. A Σ-theory is a pair $(\Sigma, \mathbf{A})$ where Σ is a signature and $\mathbf{A}$ is a class of Σ-structures. Given a theory $T = (\Sigma, \mathbf{A})$, a T-interpretation is a Σ-interpretation $\mathcal{A}$ such that $\mathcal{A}^{\Sigma} \in \mathbf{A}$. Given a Σ-theory T, a Σ-formula φ over a set of variables X is T-satisfiable if it is true on a T-interpretation over X.

Formally, the theory of linked lists is defined as $\mathsf{TLL} = (\Sigma_{\mathsf{TLL}}, \mathbf{TLL})$, where

$$\Sigma_{\mathsf{TLL}} \ := \ \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{Reachability}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{Bridge}}$$

and $\mathbf{TLL}$ is the class of Σ_{TLL}-structures satisfying the conditions shown in Fig. 4. The sorts, functions and predicates of TLL correspond to the signatures shown in Fig. 3. (Note that Figs. 4 and 3 contain an extended signature and interpretation.) Informally, Σ_{cell} models *cells*, structures containing an element (data), an addresses (pointer) and a lock owner, which represent a node in a linked list. Σ_{mem} models the memory. $\Sigma_{\mathsf{Reachability}}$ models finite sequences of non-repeating addresses, to represent paths. Σ_{set} models sets of addresses. Finally, Σ_{Bridge} is a *bridge theory* containing auxiliary functions. The sort thid contains thread identifiers. The sorts addr, elem and thid are uninterpreted, except that $\oslash : \mathsf{thid}$ is different from all others thread ids. Otherwise, $\Sigma_{\mathsf{addr}} = (\mathsf{addr}, \emptyset, \emptyset)$, $\Sigma_{\mathsf{elem}} = (\mathsf{elem}, \emptyset, \emptyset)$ and $\Sigma_{\mathsf{thid}} = (\mathsf{thid}, \emptyset, \emptyset)$.

We extend TLL into the theory of concurrent single linked lists $\mathsf{TLL3} := (\Sigma_{\mathsf{TLL3}}, \mathbf{TLL3})$, where $\Sigma_{\mathsf{TLL3}} = \Sigma_{\mathsf{TLL}} \cup \Sigma_{\mathsf{setth}} \cup \{\mathit{lockid}, \mathit{lock}, \mathit{unlock}, \mathit{firstlocked}\}$. The sorts, functions and predicates of Σ_{TLL3} are described in Fig. 3. $\mathbf{TLL3}$ is the class of Σ_{TLL3}-structures satisfying the conditions listed in Fig. 4.

Signature	Sorts	Functions	Predicates
Σ_{cell}	cell elem addr thid	$error$: cell $mkcell$: elem × addr × thid → cell $\_.data$: cell → elem $\_.next$: cell → addr $\_.lockid$: cell → thid $\_.lock$: cell → thid → cell $\_.unlock$: cell → cell	
Σ_{mem}	mem addr cell	$null$: addr $\_[\_]$: mem × addr → cell upd : mem × addr × cell → mem	
$\Sigma_{\mathsf{Reachability}}$	mem addr path	ϵ : path $[\_]$: addr → path	$append$: path × path × path $reach$: mem × addr × addr × path
Σ_{set}	addr set	$\emptyset$: set $\{\_\}$: addr → set $\cup, \cap, \setminus$: set × set → set	$\in$: addr × set $\subseteq$: set × set
Σ_{setth}	thid setth	$\emptyset_T$: setth $\{\_\}_T$: thid → setth $\cup_T, \cap_T, \setminus_T$: setth × setth → setth	$\in_T$: thid × setth $\subseteq_T$: setth × setth
Σ_{Bridge}	mem addr set path	$path2set$: path → set $addr2set$: mem × addr → set $getp$: mem × addr × addr → path $firstlocked$: mem × path → addr	

Fig. 3: The signature of the TLL3 theory

Definition 1 (Finite Model Property). *Let Σ be a signature, $S_0 \subseteq S$ be a set of sorts, and T be a Σ-theory. T has the finite model property with respect to S_0 if for every T-satisfiable quantifier-free Σ-formula φ there exists a T-interpretation $\mathcal{A}$ satisfying φ such that for each sort $\sigma \in S_0$, $\mathcal{A}_\sigma$ is finite.*

TLL [9] enjoys the finite model property. We now show that TLL3 also has the finite model property with respect to domains elem, addr and thid. Hence, TLL3 is decidable because one can enumerate Σ_{TLL3}-structures up to a certain cardinality. To prove this result, we first extend the set of normalized TLL-literals.

Definition 2 (TLL3-normalized literals). *A TLL3-literal is normalized if it is a flat literal of the form:*

$e_1 \neq e_2$	$a_1 \neq a_2$	
$a = null$	$c = error$	
$c = mkcell(e, a)$	$c = rd(m, a)$	$m_2 = upd(m_1, a, c)$
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$p_1 \neq p_2$	$p = [a]$	$p_1 = rev(p_2)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$
$s = addr2set(m, a)$	$p = getp(m, a_1, a_2)$	
$k_1 \neq k_2$	$c = mkcell\,(e, a, k)$	$a = firstlocked\,(m, p)$

where e, e_1 and e_2 are elem*-variables, a, a_1 and a_2 are* addr*-variables, c is a* cell*-variable, m, m_1 and m_2 are* mem*-variables, p, p_1, p_2 and p_3 are* path*-variables, and k, k_1 and k_2 are* thid*-variables.*

Interpretation of sort symbols: cell, mem, path, set and setth	
Each sort σ in Σ_{TLL3} is mapped to a non-empty set $\mathcal{A}_\sigma$ such that:	
(a) $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{thid}}$	(b) $\mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$
(c) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$	(d) $\mathcal{A}_{\mathsf{set}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$ (e) $\mathcal{A}_{\mathsf{setth}}$ is the power-set of $\mathcal{A}_{\mathsf{thid}}$

Signature	Interpretation
Σ_{cell}	– $mkcell(e,a,k) = \langle e,a,k\rangle$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $k \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e,a,t\rangle.data^{\mathcal{A}} = e$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e,a,t\rangle.next^{\mathcal{A}} = a$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e,a,t\rangle.lockid^{\mathcal{A}} = t$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e,a,t\rangle.lock^{\mathcal{A}}(t') = \langle e,a,t'\rangle$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t, t' \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e,a,t\rangle.unlock^{\mathcal{A}} = \langle e,a,\oslash\rangle$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t \in \mathcal{A}_{\mathsf{thid}}$ – $error^{\mathcal{A}}.next^{\mathcal{A}} = null^{\mathcal{A}}$
Σ_{mem}	– $m[a]^{\mathcal{A}} = m(a)$ for each $m \in \mathcal{A}_{\mathsf{mem}}$ and $a \in \mathcal{A}_{\mathsf{addr}}$ – $upd^{\mathcal{A}}(m,a,c) = m_{a \mapsto c}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}}$ – $m^{\mathcal{A}}(null^{\mathcal{A}}) = error^{\mathcal{A}}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$
$\Sigma_{\mathsf{Reachability}}$	– $\epsilon^{\mathcal{A}}$ is the empty sequence – $[i]^{\mathcal{A}}$ is the sequence containing $i \in \mathcal{A}_{\mathsf{addr}}$ as the only element – $([i_1,\ldots,i_n], [j_1,\ldots,j_m], [i_1,\ldots,i_n,j_1,\ldots,j_m]) \in append^{\mathcal{A}}$ iff i_k and j_l are all distinct – $(m,i,j,p) \in reach^{\mathcal{A}}$ iff $i = j$ and $p = \epsilon$, or there exist addresses $i_1,\ldots,i_n \in \mathcal{A}_{\mathsf{addr}}$ such that: (a) $p = [i_1,\ldots,i_n]$ (c) $m(i_r).next^{\mathcal{A}} = i_{r+1}$, for $1 \le r < n$ (b) $i_1 = i$ (d) $m(i_n).next^{\mathcal{A}} = j$
Σ_{set}	The symbols $\emptyset$, $\{\_\}$, $\cup$, $\cap$, $\setminus$, $\in$ and $\subseteq$ are interpreted according to their standard interpretation over sets of addresses.
Σ_{setth}	The symbols $\emptyset_T$, $\{\_\}_T$, $\cup_T$, $\cap_T$, $\setminus_T$, $\in_T$ and $\subseteq_T$ are interpreted according to their standard interpretation over sets of thread identifiers.
Σ_{Bridge}	– $addr2set^{\mathcal{A}}(m,i) = \left\{ j \in \mathcal{A}_{\mathsf{addr}} \mid \exists p \in \mathcal{A}_{\mathsf{path}} \text{ s.t. } (m,i,j,p) \in reach \right\}$ – $path2set^{\mathcal{A}}(p) = \{i_1,\ldots,i_n\}$ for $p = [i_1,\ldots,i_n] \in \mathcal{A}_{\mathsf{path}}$ – $getp^{\mathcal{A}}(m,i,j) = \begin{cases} p & \text{if } (m,i,j,p) \in reach^{\mathcal{A}} \\ \epsilon & \text{otherwise} \end{cases}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$, $p \in \mathcal{A}_{\mathsf{path}}$ and $i, j \in \mathcal{A}_{\mathsf{addr}}$ – $firstlocked^{\mathcal{A}}(m, [a_1,\ldots,a_n]) = \begin{cases} a_k & \text{if there is } 1 \le k \le n \text{ such that} \\ & \text{for all } 1 \le j < k, m[a_j].lockid = \oslash \\ & \text{and } m[a_k].lockid \neq \oslash \\ null & \text{otherwise} \end{cases}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$ and $a_1, \ldots a_n \in \mathcal{A}_{\mathsf{addr}}$

Fig. 4: Characterization of a TLL3-interpretation $\mathcal{A}$

Lemma 1. *Deciding the TLL3-satisfiability of a quantifier-free TLL3-formula is equivalent to verifying the TLL3-satisfiability of the normalized TLL3-literals.*

Proof. By cases on the shape of all possible TLL3-literals. $\square$

Consider an arbitrary TLL3-interpretation $\mathcal{A}$ satisfying a conjunction of normalized TLL3-literals Γ. We show that if there are sets $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{addr}}$ and $\mathcal{A}_{\mathsf{thid}}$

then there are finite sets $\mathcal{A}'_{\mathsf{elem}}$, $\mathcal{A}'_{\mathsf{addr}}$ and $\mathcal{A}'_{\mathsf{thid}}$ with bounded cardinalities (the bound depending on Γ). $\mathcal{A}'_{\mathsf{elem}}$, $\mathcal{A}'_{\mathsf{addr}}$ and $\mathcal{A}'_{\mathsf{thid}}$ can in turn be used to obtain a finite interpretation $\mathcal{A}'$ satisfying Γ.

Lemma 2 (Finite Model Property). *Let Γ be a conjunction of normalized* TLL3*-literals. Let $\bar{e} = |V_{\mathsf{elem}}(\Gamma)|$, $\bar{a} = |V_{\mathsf{addr}}(\Gamma)|$, $\bar{m} = |V_{\mathsf{mem}}(\Gamma)|$, $\bar{p} = |V_{\mathsf{path}}(\Gamma)|$ and $\bar{k} = |V_{\mathsf{thid}}(\Gamma)|$. Then the following are equivalent:*
1. *Γ is* TLL3*-satisfiable;*
2. *Γ is true in a* TLL3 *interpretation $\mathcal{A}$ such that*

$$
\begin{aligned}
|\mathcal{A}_{\mathsf{elem}}| &\le \bar{e} + \bar{m}\ |\mathcal{A}_{\mathsf{addr}}| \\
|\mathcal{A}_{\mathsf{addr}}| &\le \bar{a} + 1 + \bar{m}\ \bar{a} + \bar{p}^2 + \bar{p}^3 + \bar{m}\bar{p} \\
|\mathcal{A}_{\mathsf{thid}}| &\le \bar{k} + \bar{m}\,|\mathcal{A}_{\mathsf{addr}}| + 1
\end{aligned}
$$

Proof. $(2 \to 1)$ is immediate. $(1 \to 2)$, by case analysis on normalized TLL3 literals. □

Lemma 3 justifies a brute force method to automatically check TLL3 satisfiability of normalized TLL3-literals. However, such a method is not efficient in practice. To find a more efficient decision procedure we decompose TLL3 into a combination of theories, and apply a many-sorted variant of the Nelson-Oppen combination method [12]. This method requires the theories to fulfill two conditions. First, each theory must have a decision procedure. Second, all involved theories must be stable infinite and share sorts only.

Definition 3 (stable-infiniteness). *A Σ-theory T is stably infinite if for every T-satisfiable quantifier-free Σ-formula φ there exists a T-interpretation $\mathcal{A}$ satisfying φ whose domain is infinite.*

All theories involved in TLL [9] are stably-infinite, so the only missing theory is the one defining *firstlocked*. We define the theory T_{Base3} as follows:

$$T_{\mathsf{Base3}} = T_{\mathsf{addr}} \oplus T_{\mathsf{elem}} \oplus T_{\mathsf{cell}} \oplus T_{\mathsf{mem}} \oplus T_{\mathsf{path}} \oplus T_{\mathsf{set}} \oplus T_{\mathsf{setth}} \oplus T_{\mathsf{thid}}$$

where T_{path} extends the theory of finite sequences of addresses with the auxiliary functions and predicates shown in Fig. 5.

The theory of finite sequences of addresses is defined by $T_{\mathsf{fseq}} = (\Sigma_{\mathsf{fseq}}, \mathsf{TGen})$, where $\Sigma_{\mathsf{fseq}} = \big(\{\mathsf{addr}, \mathsf{fseq}\}, \{nil : \mathsf{fseq}, cons : \mathsf{addr} \times \mathsf{fseq} \to \mathsf{fseq}, hd : \mathsf{fseq} \to \mathsf{addr}, tl : \mathsf{fseq} \to \mathsf{fseq}\}, \emptyset\big)$ and TGen as the class of multi-sorted term-generated structures that satisfy the axioms of T_{fseq}. These axioms are the standard for a theory of lists, such as distinctness, uniqueness and generation of sequences using the constructors *cons* and *nil*, as well as acyclicity of sequences (see, for example [3]). Let *PATH* be the set of axioms of T_{fseq} including all in Fig. 5. Then, we can formally define $T_{\mathsf{path}} = (\Sigma_{\mathsf{path}}, \mathsf{ETGen})$ where ETGen is $\big\{\mathcal{A}^{\Sigma_{\mathsf{path}}} | \mathcal{A}^{\Sigma_{\mathsf{path}}} \models PATH$ and $\mathcal{A}^{\Sigma_{\mathsf{fseq}}} \in \mathsf{TGen}\big\}$. Next, we extend T_{Base3} defining the missing functions and predicates from $T_{\mathsf{Reachability}}$ and Σ_{Bridge}. For example:

$$ispath\,(p) \land firstmarked\,(m, p, i) \leftrightarrow firstlocked\,(m, p) = i$$

$app : \mathsf{fseq} \times \mathsf{fseq} \rightarrow \mathsf{fseq}$
$app(nil, l) = l$ $app(cons(a, l), l') = cons(a, app(l, l'))$

$fseq2set : \mathsf{fseq} \rightarrow \mathsf{set}$
$fseq2set(nil) = \emptyset$ $fseq2set(cons(a, l)) = \{a\} \cup fseq2set(l)$

$ispath : \mathsf{fseq}$
$ispath(nil)$ $ispath(cons(a, nil))$ $\{a\} \nsubseteq fseq2set(l) \wedge ispath(l) \rightarrow ispath(cons(a, l))$

$last : \mathsf{fseq} \rightarrow \mathsf{addr}$
$last(cons(a, nil)) = a$ $l \neq nil \rightarrow last(cons(a, l)) = last(l)$

$isreachable : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr}$
$isreachable(m, a, a)$ $m[a].next = a' \wedge isreachable(m, a', b) \rightarrow isreachable(m, a, b)$

$isreachablep : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{fseq}$
$isreachablep(m, a, a, nil)$ $m[a].next = a' \wedge isreachablep(m, a', b, p) \rightarrow isreachablep(m, a, b, cons(a, p))$

$firstmarked : \mathsf{mem} \times \mathsf{fseq} \times \mathsf{addr}$
$firstmarked(m, nil, null)$ $p \neq nil \wedge p = cons(j, q) \wedge m[j].lockid \neq \oslash \rightarrow firstmarked(m, p, j)$ $p \neq nil \wedge p = cons(j, q) \wedge m[j].lockid = \oslash \wedge firstmarked(m, q, i) \rightarrow firstmarked(m, p, i)$

Fig. 5: Functions, predicates and axioms of T_{path}

Let GAP be the set of axioms that define ϵ, $[\_]$, $append$, $reach$, $path2set$, $addr2set$ and $getp$. We define $\widehat{\mathsf{TLL3}} = (\Sigma_{\widehat{\mathsf{TLL3}}}, \widehat{\mathsf{ETGen}})$ where $\Sigma_{\widehat{\mathsf{TLL3}}}$ is $\Sigma_{\mathsf{TLL}} \cup \{$ $getp$, $append$, $path2set$, $firstlocked$ $\}$ and $\widehat{\mathsf{ETGen}} := \{\mathcal{A}^{\Sigma_{\widehat{\mathsf{TLL3}}}} | \mathcal{A}^{\Sigma_{\widehat{\mathsf{TLL3}}}} \models GAP$ and $\mathcal{A}^{\Sigma_{\mathsf{path}}} \in \mathsf{ETGen}\}$.

Using the definitions of GAP it is easy to prove that if Γ is a set of normalized $\mathsf{TLL3}$-literals, then Γ is $\mathsf{TLL3}$-satisfiable iff Γ is $\widehat{\mathsf{TLL3}}$-satisfiable. Therefore, $\widehat{\mathsf{TLL3}}$ can be used in place of $\mathsf{TLL3}$ for satisfiability checking. We reduce $\widehat{\mathsf{TLL3}}$ into T_{Base3} in two steps. First we do the unfolding of the definition of auxiliary functions defined in $PATH$ and GAP, getting rid of the extra functions, and obtaining a formula in $\widehat{\mathsf{TLL3}}$ and T_{Base}. Then, we use the known reduction [9] from $\widehat{\mathsf{TLL}}$ into T_{Base}. All theories involved in T_{Base3} share only sorts symbols, are stably-infinite and for all of them there is a decision procedure. Hence, the multi-sorted Nelson-Oppen combination method can be applied, obtaining a decision procedure for $\mathsf{TLL3}$.

We now define some auxiliary functions and predicates using $\mathsf{TLL3}$, that aid in the reasoning about concurrent linked-lists (see Fig. 6). For example, predicate $List(h, a, r)$ expresses that in heap h, starting from address a there is sequence of cells all of which form region r. Function $LastMarked(h, p)$, on the other hand,

$\mathit{List} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{set}$
$\mathit{List}(h, a, r) \leftrightarrow \mathit{null} \in \mathit{addr2set}(h, a) \wedge r = \mathit{path2set}(\mathit{getp}(h, a, \mathit{null}))$

$f_a : \mathsf{mem} \times \mathsf{addr} \to \mathsf{path}$
$f_a(h, n) = \begin{cases} \epsilon & \text{if } n = \mathit{null} \\ \mathit{getp}(h, h[n].\mathit{next}, \mathit{null}) & \text{if } n \neq \mathit{null} \end{cases}$

$\mathit{LastMarked} : \mathsf{mem} \times \mathsf{path} \to \mathsf{addr}$
$\mathit{LastMarked}(m, p) = \mathit{firstlocked}(m, \mathit{rev}(p))$

$\mathit{NoMarks} : \mathsf{mem} \times \mathsf{path}$
$\mathit{NoMarks}(m, p) \leftrightarrow \mathit{firstlocked}(m, p) = \mathit{null}$

$\mathit{SomeMark} : \mathsf{mem} \times \mathsf{path}$
$\mathit{SomeMark}(m, p) \leftrightarrow \mathit{firstlocked}(m, p) \neq \mathit{null}$

Fig. 6: Auxiliary functions to reason about concurrent lists

returns the address of the last locked node in path p on memory h. All these functions can be used in verification conditions. Then, using the equivalences in Fig. 6 the predicates are removed, generating a pure $\widehat{\mathsf{TLL3}}$ formula whose satisfiability can be checked with the procedure described above.

5 Termination of Concurrent Lock-Coupling Lists

In this section we show the proof of a simple liveness property of concurrent lock-coupling lists: termination of the leading thread.

To aid in the verification of this property we annotate the code in Fig. 1 with ghost fields and ghost updates, as shown in Fig. 7, where the boxes represent the annotations introduced. The predicate $c.\mathit{lockid} = \oslash$ denotes that the lock of list node c is not taken. The predicate $c.\mathit{lockid} = k$ establishes that the lock at list node c is owned by thread k. We enrich *List* objects with a ghost field r of type region that keeps track of all the nodes in the list. The code for *add* and *remove* is extended with ghost updates to maintain r.

T_k denotes thread k. We want to prove that if a thread has acquired a lock at node n and no other thread holds a lock ahead of n, then thread k eventually terminates. The predicate $\mathit{at\_add}^{[k]}_{n}$ means that thread k is executing line n of program *add*. Similarly, $\mathit{at\_add}^{[k]}_{n_1,\ldots,n_m}$ is a short for thread k is running some of the lines $n_1, \ldots, n_m$ of program *add*. To reduce notation, $\tau^{[k]}_{a_n}$, $\tau^{[k]}_{r_n}$ and $\tau^{[k]}_{l_n}$ denote $\tau^{[k]}_{\mathit{add}_n}$, $\tau^{[k]}_{\mathit{remove}_n}$ and $\tau^{[k]}_{\mathit{locate}_n}$ respectively. The instance of a local variable v in thread k is represented by $v^{[k]}$. We define *DisjList* as an extension of *List* enriching it with the property that new nodes created during insertion are all disjoint one from each other, including all nodes that are already part of the list:

$$\begin{aligned} \mathit{DisjList}(h, a, r) \mathrel{\hat{=}} \mathit{List}(h, a, r) \wedge \forall j : T_{ID}.\mathit{at\_a}^{[j]}_{4,5} \to \langle \mathit{aux}^{[j]} \rangle \# r \wedge \\ \forall i, j : T_{ID}.i \neq j \wedge \mathit{at\_a}^{[i]}_{4,5} \wedge \mathit{at\_a}^{[j]}_{4,5} \to \langle \mathit{aux}^{[i]} \rangle \# \langle \mathit{aux}^{[j]} \rangle \# r \end{aligned}$$

We now define the following auxiliary predicate:

$$\begin{aligned}\mathit{IsLast}(k) \mathrel{\hat{=}} \mathit{DisjList}(h, l.\mathit{list}, l.r) \wedge \mathit{SomeMark}\big(h, \mathit{getp}(h, l.\mathit{list}, \mathit{null})\big) \\ \wedge\ \mathit{LastMarked}\big(h, \mathit{getp}(h, l.\mathit{list}, \mathit{null})\big) = a \quad \wedge \quad h[a].\mathit{lockid} = k\end{aligned}$$

The formula $\mathit{IsLast}(k)$ identifies whether T_k is the thread owning the last lock in the list (i.e., the closest node towards the end of the list). Using these predicates we define the parametrized temporal formula we want to verify as:

$$\psi(k) \mathrel{\hat{=}} \square \left(\mathit{at\_locate}^{[k]}_{3..10} \wedge \mathit{IsLast}(k) \rightarrow \mathit{IsLast}(k)\, \mathcal{U}\, \mathit{at\_locate}^{[k]}_{11} \right)$$

This temporal formula states that if thread k is running *locate* and it owns the last locked node in the list, then thread T_k will still own the last locked node until T_k reaches the last line of *locate*. Reachability of the last line of *locate* implies termination of the invocation to the concurrent datatype because *locate* is the only program containing potentially blocking operations.

We proceed with the construction of a verification diagram that proves the parallel execution of all threads guarantee the satisfaction of formula $\psi(k)$. Given N, we build the transitions system $\mathcal{S}[N]$, in which threads $T_1, \ldots, T_N$ run in parallel the program *decide* and show that $\mathcal{S}[N] \vDash \psi(k)$. The verification diagram

```
class List {
    Node list;
    [ rgn r; ]
}

class Node {
    Value val;
    Node next;
    Lock lock;
}
```

(a) data structure

```
 1: $prev := Head$
 2: $prev.lock()$
 3: $curr := prev.next$
 4: $curr.lock()$
 5: while $curr.val < e$ do
 6:     $prev.unlock()$
 7:     $prev := curr$
 8:     $curr := curr.next$
 9:     $curr.lock()$
10: end while
11: return $(prev, curr)$
```

(b) *locate*

```
 1: $prev, curr := locate(e)$
 2: if $curr.val \neq e$ then
 3:     $aux :=$ new $Node(e)$
 4:     $aux.next := curr$
 5:     $prev.next := aux$
          [ $l.r := l.r \cup \langle aux \rangle$ ]
 6:     $result := true$
 7: else
 8:     $result := false$
 9: end if
10: $prev.unlock()$
11: $curr.unlock()$
12: return $result$
```

(c) *add*

```
 1: $prev, curr := locate(e)$
 2: if $curr.val = e$ then
 3:     $aux := curr.next$
 4:     $prev.next := aux$
          [ $l.r := l.r - \langle curr \rangle$ ]
 5:     $result := true$
 6: else
 7:     $result := false$
 8: end if
 9: $prev.unlock()$
10: $curr.unlock()$
11: return $result$
```

(d) *remove*

Fig. 7: Concurrent lock-coupling list extended with ghost fields

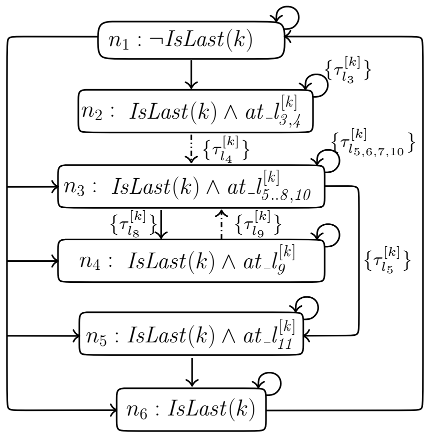
$n_1 : \neg IsLast(k)$
$n_2 : IsLast(k) \wedge at\_l^{[k]}_{3,4}$
$\{\tau^{[k]}_{l_3}\}$
$\{\tau^{[k]}_{l_4}\}$
$\{\tau^{[k]}_{l_{5,6,7,10}}\}$
$n_3 : IsLast(k) \wedge at\_l^{[k]}_{5..8,10}$
$\{\tau^{[k]}_{l_8}\}$
$\{\tau^{[k]}_{l_9}\}$
$n_4 : IsLast(k) \wedge at\_l^{[k]}_{9}$
$\{\tau^{[k]}_{l_5}\}$
$n_5 : IsLast(k) \wedge at\_l^{[k]}_{11}$
$n_6 : IsLast(k)$

Fig. 8: Verification diagram Ψ for $\|_{j<N}\ T_j \vDash \psi(k)$

is depicted in Fig. 8. Dashed arrows in the diagram denote transitions that strictly decrement the ranking function δ. Formally, the verification diagram is:

- $N_0 = \{n_1\}$
- $\mathcal{F} = \{(P, R)\}$ where
  $P = \{(n_3, n_4), (n_3, n_5), (n_5, n_6), (n_6, n_1)\} \cup$
  $\{(n_1, n_j) | j \in 2..6\} \cup \{(n_j, n_j) | j \in 1..6\}$
  $R = \emptyset$
- $\delta(n, s) = \begin{cases} \{a \mid a \in dom(h)\} & n = n_1, n_2 \\ path2set\left(f_a(h, LastMarked(h, getp(h, prev^{[k]}, null)))\right) & \text{otherwise} \end{cases}$
- $f(n) = \begin{cases} \emptyset & \text{if } n = n_1, n_6 & at\_locate^{[k]}_{3,4} & \text{if } n = n_2 \\ at\_locate^{[k]}_{5..8,10} & \text{if } n = n_3 & at\_locate^{[k]}_{9} & \text{if } n = n_4 \\ at\_locate^{[k]}_{11} & \text{if } n = n_5 \end{cases}$

We can now describe the verification conditions:

initialization Trivial, since in the initial state *l.list* forms an empty list, and consequently $\neg IsLast(k)$.

consecution We will show, for illustration purposes, transition $\tau^{[j]}_{l_9}$ on node n_2 with $j \neq k$. The verification condition is:

$$\left(\begin{array}{c} \overbrace{IsLast(k)}^{\mathsf{TLL3}} \wedge \overbrace{j \neq k}^{T_{\mathsf{thid}}} \wedge \\ at\_l^{[k]}_{3,4} \wedge at\_l^{[j]}_{9} \wedge \\ \underbrace{curr^{[j]}.lockid = \oslash}_{\mathsf{TLL3}} \end{array}\right) \wedge \overbrace{curr^{[j]}.lock(j)}^{\mathsf{TLL3}} \rightarrow \left(\begin{array}{c} \overbrace{IsLast(k')}^{\mathsf{TLL3}} \wedge at'\_l^{[k']}_{3,4} \wedge \\ at'\_l^{[j']}_{10} \wedge pres(\mathcal{V} - curr^{[j]}) \wedge \\ \underbrace{curr'^{[j']}.lockid = j'}_{\mathsf{TLL3}} \end{array}\right)$$

where *pres* is the predicate denoting variable preservation. Note that all fragments of such verification condition belong to theories for which we have already defined a decision procedure, including propositional logic for the (finite) locations of the program counters.

acceptance The ranking function δ maps, at a given state, the set of list nodes accessible from the last node with an owned lock. This set remains identical for all transitions except $\tau_{l_4}^{[k]}$ and $\tau_{l_9}^{[k]}$, for which the set decrements (in the inclusion order on sets). The decision procedure presented in Section 4 proves this automatically (using $\subset$ operation and equality over sets of addresses).

fairness Only two conditions must be verified. First, all transitions labeling an edge are enabled since the only potentially blocking operation is $\tau_{l_9}^{[k]}$ and $\mathit{IsLast}(k)$ implies that $\tau_{l_9}^{[k]}$ is enabled. Second, for all nodes and labelled edges, starting from a state that satisfies the predicate of the incoming node satisfies the predicate of the outgoing node via taking the transition. Sequential progress of thread k is guaranteed by fairness, since all idling transitions for thread k are in fact a diagram idiom to represent the expansion of such nodes to a sequence of nodes with a single program position on each node.

satisfaction $\mathcal{L}(\Psi) \subseteq \mathcal{L}(\psi(k))$ is automatically checkable via a finite LTL model-checking problem.

6 Conclusion

We have presented a method for the verification of temporal properties (safety and liveness) of an imperative implementation of concurrent lists. The verification is performed using verification diagrams – a complete method to prove temporal properties of reactive systems – and explicit reasoning of memory regions. The verification process usually requires the aid of ghost variables. Checking a proof is reduced to proving a finite number of verification conditions, which requires decision procedures in the appropriate theories, including regions, pointers, locks and specific theories for memory layouts, in this case single linked-lists. This paper also presents a decision procedure built as a combination of theories.

There are some key differences with other approaches in the literature. Building on the success of separation logic in proving sequential programs, the most popular approach has been extending separation logic to concurrent programs. These extensions require adapting techniques like rely-guarantee that cannot be directly used with separation logic. Our decision to use explicit regions (finite sets of addresses) allows the direct use of classical techniques like assume-guarantee and the combination of decision procedures. Furthermore, in concurrent separation logic, it is critical to describe memory footprints of sections of code. This description becomes very cumbersome when the code is not organized in mutual exclusion regions, as in fine-grain synchronization algorithms. Moreover, the integration into SMT solvers is quite straightforward with classical logics, but it is still an open question with separation logic.

The technique we propose can be seen as a method to separate the reasoning about concurrency (with verification diagrams) from the reasoning about the memory (with decision procedures). The former is independent of the data structure under consideration. We are currently extending our approach to the verification of other pointer-based concurrent data structures like skip-lists or concurrent hash maps. Again, the sharing of these data structures makes it very hard to reason using separation logic. For our approach, these extensions will require the design of suitable decision procedures. Future work also includes building a generic VCgen for verification diagrams, implementing an ad-hoc version of the decision procedure described here, and later integrating this decision procedure into state-of-the-art SMT solvers.

Acknowledgment

We are grateful to the anonymous reviewers for their detailed comments and suggestions.

References

1. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Proc. of Europ. Conf. Object-Oriented Programming (ECOOP'08). LNCS, vol. 5142, pp. 387–411. Springer (2008)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Handbook of Satifiability, chap. Satisfiability Modulo Theories. IOS Press (2008)
3. Bradley, A.R., Manna, Z.: The Calculus of Computation. Springer (2007)
4. Browne, A., Manna, Z., Sipma, H.B.: Generalized verification diagrams. In: Proc. of 15th Conf. on the Foundations of Software Theory an Theoretical Computer Science (FSTTCS'95). LNCS, vol. 1206, pp. 484–498. Springer (1995)
5. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan-Kaufmann (2008)
6. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Proc. of European Symposium on Programming (ESOP'08). LNCS, vol. 4960, pp. 353–367. Springer (2008)
7. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer (1995)
8. McMillan, K.L.: Circular compositional reasoning about liveness. In: Proc. of CHARME'99. LNCS, vol. 1703, pp. 342–345. Springer (1999)
9. Ranise, S., Zarba, C.G.: A theory of singly-linked lists and its extensible decision procedure. In: Proc. of Software Engineering and Formal Methods (SEFM'06). IEEE Computer Society Press (2006)
10. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. Logic in Computer Science (LICS'02). pp. 55–74. IEEE Computer Society Press (2002)
11. Sipma, H.B.: Diagram-Based Verification of Discrete, Real-Time and Hybrid Systems. Ph.D. thesis, Stanford University (1999)
12. Tinelli, C., Zarba, C.G.: Combining decision procedures for sorted theories. In: Proc. Logic in Artificial Intelligence (JELIA'04). LNCS, vol. 3229, pp. 641–653. Springer (2004)
13. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: Principles & Practice of Parallel Programming (PPOPP'06). pp. 129–136. ACM (2006)
14. Wies, T., Piskac, R., Kuncak, V.: Combining theories with shared set operations. In: Proc. of Frontiers of Combining Systems (FroCoS'09). LNCS, vol. 5749, pp. 366–382. Springer (2009)

A Missing Proofs

Lemma 1. *Deciding the* TLL3*-satisfiability of a quantifier-free* TLL3*-formula is equivalent to verifying the* TLL3*-satisfiability of the normalized* TLL3*-literals.*

Proof. By cases on the shape of all possible TLL3-literals. □

We define the *compress* function which, given a path p and a set X of addresses, returns the path obtained from p by removing all the addresses that do not belong to X.

$$compress([i_1,\ldots,i_n],X) = \begin{cases} \epsilon & \text{if } n=0 \\ [i_1] \circ compress\left([i_2,\ldots,i_n],X\right) & \text{if } n>0 \text{ and } i_1 \in X \\ compress\left([i_2,\ldots,i_n],X\right) & \text{otherwise} \end{cases}$$

Lemma 3 (Finite Model Property). *Let Γ be a conjunction of normalized* TLL3*-literals. Let $\overline{e} = |V_{\mathsf{elem}}(\Gamma)|$, $\overline{a} = |V_{\mathsf{addr}}(\Gamma)|$, $\overline{m} = |V_{\mathsf{mem}}(\Gamma)|$, $\overline{p} = |V_{\mathsf{path}}(\Gamma)|$ and $\overline{k} = |V_{\mathsf{thid}}(\Gamma)|$. Then the following are equivalent:*
1. Γ is TLL3*-satisfiable;*
2. Γ is true in a TLL3 *interpretation $\mathcal{A}$ such that*

$$\begin{aligned} |\mathcal{A}_{\mathsf{elem}}| &\le \overline{e} + \overline{m}\ |\mathcal{A}_{\mathsf{addr}}| \\ |\mathcal{A}_{\mathsf{addr}}| &\le \overline{a} + 1 + \overline{m}\,\overline{a} + \overline{p}^2 + \overline{p}^3 + \overline{mp} \\ |\mathcal{A}_{\mathsf{thid}}| &\le \overline{k} + \overline{m}\,|\mathcal{A}_{\mathsf{addr}}| + 1 \end{aligned}$$

Proof. $(2 \to 1)$. Immediate.

$(1 \to 2)$. We will prove this implication only for the new TLL3-literals. Before doing so, we define some auxiliary functions. We start by defining the function *first*. Let $X \subseteq \tilde{X}$, $m : \tilde{X} \to Z \times \tilde{X} \times Y$ and $a \in X$. The function $first(m,a,X)$ is defined by

$$first(m,a,X) = \begin{cases} null & \text{if } (\forall r \ge 1)\left[m^r(a).next \notin X\right] \\ m^s(a).next & \text{if } (\exists s \ge 1)\Big[m^s(a).next \in X \wedge \\ & \quad (\forall r \ge 1)\left(r < s \to m^r(a).next \notin X\right)\Big] \end{cases}$$

where $m^1(a).next$ stands for $m(a).next$ and $m^{n+1}(a).next$ for $m(m^n(a).next).next$ when $n > 1$. We also define the *compress* function which, given a path p and a set X of addresses, returns the path obtained from p by removing all the addresses that do not belong to X.

$$compress([i_1,\ldots,i_n],X) = \begin{cases} \epsilon & \text{if } n=0 \\ [i_1] \circ compress\left([i_2,\ldots,i_n],X\right) & \text{if } n>0 \text{ and } i_1 \in X \\ compress\left([i_2,\ldots,i_n],X\right) & \text{otherwise} \end{cases}$$

We now define the κ : $\mathsf{mem} \times \mathsf{path} \to \mathsf{set}$ function. This function, given m of sort mem and p of sort path analyzes the cells stored at each address in p mapped at memory m and returns a set with the address at which the first locked node has been found.

$$\kappa\left(m,[i_1,\ldots,i_n]\right)=\begin{cases}\emptyset & \text{if } n=0\\ \{i_1\} & \text{if } m\left[i_1\right].\mathit{lockid}\neq\oslash\\ \kappa\left(m,[i_2,\ldots,i_n]\right) & \text{if } m\left[i_1\right].\mathit{lockid}=\oslash\end{cases}$$

We conclude by defining the function δ [9] that outputs a set of addresses accountable for disequality of two given paths:

$$\delta([i_1,\ldots,i_n],[j_1,\ldots j_m])=\begin{cases}\emptyset & \text{if } n=m=0\\ \{i_1\} & \text{if } n>0 \text{ and } m=0\\ \{j_1\} & \text{if } n=0 \text{ and } m>0\\ \{i_1,j_1\} & \text{if } n,m>0 \text{ and } i_1\neq j_1\\ \delta([i_2,\ldots,i_m],[j_2,\ldots,j_m]) & \text{otherwise}\end{cases}$$

and function σ [9] that outputs an element common to two paths (an element that witnesses that $\mathit{path2set}(p)\cap\mathit{path2set}(q)\neq\emptyset$):

$$\sigma([i_1,\ldots,i_n],p)=\begin{cases}\emptyset & \text{if } n=0\\ \{i_1\} & \text{if } n>0 \text{ and } i_1\in\mathit{path2set}(p)\\ \sigma([i_2,\ldots,i_n],p) & \text{otherwise}\end{cases}$$

Bearing in mind all auxiliary functions defined above, let now $\mathcal{B}$ be a TLL3-interpretation satisfying Γ. We will use $\mathcal{B}$ to construct a TLL3-interpretation $\mathcal{A}$ satisfying Γ. We define the sets X, Y and Z as

$$\begin{aligned}X=\ & V^{\mathcal{B}}_{\mathsf{addr}}\cup\left\{\mathit{null}^{\mathcal{B}}\right\}\ \cup\\ & \left\{m^{\mathcal{B}}(v^{\mathcal{B}}).\mathit{next}^{\mathcal{B}}\mid m\in V_{\mathsf{mem}}\text{ and } v\in V_{\mathsf{addr}}\right\}\cup\\ & \left\{v\in\delta(p^{\mathcal{B}},q^{\mathcal{B}})\mid\ \text{the literal } p\neq q \text{ is in } \Gamma\right\}\ \cup\\ & \left\{v\in\sigma({p_1}^{\mathcal{B}},{p_2}^{\mathcal{B}})\mid\ \text{the literal } \neg\mathit{append}(p_1,p_2,p_3)\text{ is in }\Gamma\text{ and}\right.\\ & \qquad\qquad\left.\mathit{path2set}^{\mathcal{B}}({p_1}^{\mathcal{B}})\cap\mathit{path2set}^{\mathcal{B}}({p_2}^{\mathcal{B}})\neq\emptyset\right\}\ \cup\\ & \left\{v\in\sigma({p_1}^{\mathcal{B}}\circ{p_2}^{\mathcal{B}},{p_3}^{\mathcal{B}})\mid\ \text{the literal } \neg\mathit{append}(p_1,p_2,p_3)\text{ is in }\Gamma\text{ and}\right.\\ & \qquad\qquad\left.\mathit{path2set}^{\mathcal{B}}({p_1}^{\mathcal{B}})\cap\mathit{path2set}^{\mathcal{B}}({p_2}^{\mathcal{B}})=\emptyset\right\}\ \cup\\ & \left\{v\in\kappa(m,p)\mid\mathit{firstlocked}(m,p)\text{ is in }\Gamma\right\}\end{aligned}$$

$$Y=V^{\mathcal{B}}_{\mathsf{thid}}\cup\left\{\ \oslash\ \right\}\cup\left\{m^{\mathcal{B}}(v).\mathit{lockid}^{\mathcal{B}}\mid m\in V_{\mathsf{mem}}\text{ and } v\in X\right\}$$

$$Z=V^{\mathcal{B}}_{\mathsf{elem}}\cup\left\{m^{\mathcal{B}}(v).\mathit{data}^{\mathcal{B}}\mid m\in V_{\mathsf{mem}}\text{ and } v\in X\right\}$$

We now let $\mathcal{A}$ be the $\mathsf{TLL3}$-interpretation defined by

$$\mathcal{A}_{\mathsf{addr}} = X, \quad \mathcal{A}_{\mathsf{thid}} = Y \quad \text{and} \quad \mathcal{A}_{\mathsf{elem}} = Z$$

and let

$$\begin{aligned}
\mathit{error}^{\mathcal{A}} &= \mathit{error}^{\mathcal{B}} && \\
\mathit{null}^{\mathcal{A}} &= \mathit{null}^{\mathcal{B}} && \\
e^{\mathcal{A}} &= e^{\mathcal{B}} && \text{for each } e \in V_{\mathsf{elem}} \\
a^{\mathcal{A}} &= a^{\mathcal{B}} && \text{for each } a \in V_{\mathsf{addr}} \\
c^{\mathcal{A}} &= c^{\mathcal{B}} && \text{for each } c \in V_{\mathsf{cell}} \\
k^{\mathcal{A}} &= k^{\mathcal{B}} && \text{for each } k \in V_{\mathsf{thid}} \\
m^{\mathcal{A}}(v) &= \big(m^{\mathcal{B}}(v).\mathit{data}^{\mathcal{B}}, \mathit{first}(m^{\mathcal{B}}, v, \mathcal{B}_{\mathsf{addr}}), m^{\mathcal{B}}(v).\mathit{lockid}^{\mathcal{B}}\big) && \text{for each } m \in V_{\mathsf{mem}} \\
& && \quad \text{and } v \in \mathcal{B}_{\mathsf{addr}} \\
s^{\mathcal{A}} &= s^{\mathcal{B}} \cap \mathcal{B}_{\mathsf{addr}} && \text{for each } s \in V_{\mathsf{set}} \\
g^{\mathcal{A}} &= g^{\mathcal{B}} \cap \mathcal{B}_{\mathsf{thid}} && \text{for each } g \in V_{\mathsf{setth}} \\
p^{\mathcal{A}} &= \mathit{compress}(p^{\mathcal{B}}, \mathcal{B}_{\mathsf{addr}}) && \text{for each } p \in V_{\mathsf{path}}
\end{aligned}$$

Clearly, by construction $\mathcal{A}_{\mathsf{addr}}$, $\mathcal{A}_{\mathsf{thid}}$ and $\mathcal{A}_{\mathsf{elem}}$ satisfy the given cardinality constraints. The proof that $\mathcal{A}$ satisfies all TLL-literals in Γ is not shown here. For $\mathsf{TLL3}$-literals we must consider the following cases:

Literals of the form $k_1 \neq k_2$ Immediate
Literals of the form $c = \mathit{mkcell}(e, a, k)$ We know that

$$c^{\mathcal{A}} = c^{\mathcal{B}} = \big(e^{\mathcal{B}}, a^{\mathcal{B}}, k^{\mathcal{B}}\big) = \big(e^{\mathcal{A}}, a^{\mathcal{A}}, k^{\mathcal{A}}\big)$$

Literals of the form $c = \mathit{rd}(m, a)$ In this case we have that

$$\begin{aligned}
\big[\mathit{rd}(m, a)\big]^{\mathcal{A}} &= m^{\mathcal{A}}(a^{\mathcal{A}}) \\
&= m^{\mathcal{A}}(a^{\mathcal{B}}) \\
&= \Big(m^{\mathcal{B}}(a^{\mathcal{B}}).\mathit{data}^{\mathcal{B}}, \mathit{first}(m^{\mathcal{B}}, a^{\mathcal{B}}, X), m^{\mathcal{B}}(a^{\mathcal{B}}).\mathit{lockid}^{\mathcal{B}}\Big) \\
&= \Big(m^{\mathcal{B}}(a^{\mathcal{B}}).\mathit{data}^{\mathcal{B}}, m^{\mathcal{B}}(a^{\mathcal{B}}).\mathit{next}^{\mathcal{B}}, m^{\mathcal{B}}(a^{\mathcal{B}}).\mathit{lockid}^{\mathcal{B}}\Big) \quad \text{(Lemma 21.a [9])} \\
&= m^{\mathcal{B}}(a^{\mathcal{B}}) \\
&= c^{\mathcal{B}} \\
&= c^{\mathcal{A}}
\end{aligned}$$

Literals of the form $m = \mathit{upd}(\tilde{m}, a, c)$ In this particular case we want to prove that $m^{\mathcal{A}} = \tilde{m}^{\mathcal{A}}_{a^{\mathcal{A}} \mapsto c^{\mathcal{A}}}$. However, since $m^{\mathcal{B}}(a^{\mathcal{B}}) = c^{\mathcal{B}}$, then we have that $c^{\mathcal{A}} = m^{\mathcal{A}}(a^{\mathcal{A}})$. Let now $v \neq a^{\mathcal{A}}$. We have that

$$\begin{aligned}
m^{\mathcal{A}}(v) &= \Big(m^{b}(v).\mathit{data}^{\mathcal{B}}, \mathit{first}(m^{\mathcal{B}}, v, X), m^{\mathcal{B}}(v).\mathit{lockid}^{\mathcal{B}}\Big) \\
&= \Big(\tilde{m}^{b}(v).\mathit{data}^{\mathcal{B}}, \mathit{first}(\tilde{m}^{\mathcal{B}}, v, X), \tilde{m}^{\mathcal{B}}(v).\mathit{lockid}^{\mathcal{B}}\Big) \text{ (Lemma 21b [9])} \\
&= \tilde{m}^{\mathcal{B}}(v)
\end{aligned}$$

Literals of the form $a = \mathit{firstlocked}(m, p)$. If we consider the case at which $p = \epsilon$, then we know that $\mathit{firstlocked}^{\mathcal{B}}(m^{\mathcal{B}}, \epsilon^{\mathcal{B}}) = \mathit{null}^{\mathcal{B}}$. At the same time, we know that $\epsilon^{\mathcal{A}} = \mathit{compress}(\epsilon^{\mathcal{B}}, X)$ and so $\mathit{firstlocked}^{\mathcal{A}}(m^{\mathcal{A}}, \epsilon^{\mathcal{A}}) = \mathit{null}^{\mathcal{A}}$. Let's now consider the case at which $p = [a_1, \ldots, a_n]$. There are two possible scenarios to consider.

- If for all $1 \le k \le n$, $m^{\mathcal{B}}(a_k^{\mathcal{B}}).\mathit{lockid}^{\mathcal{B}} = \oslash$, then we have that

$$\mathit{firstlocked}^{\mathcal{B}}(m^{\mathcal{B}}, p^{\mathcal{B}}) = \mathit{null}^{\mathcal{B}}$$

  Notice that function *compress* returns a subset of the path it receives with the property that all addresses in the returned path belong to the received path. Then, if $[\tilde{a}_1, \ldots, \tilde{a}_m] = p^{\mathcal{A}} = \mathit{compress}(p^{\mathcal{B}}, X)$, we know that $\{\tilde{a}_1, \ldots, \tilde{a}_m\} \subseteq X$ and hence for all $1 \le j \le m$, $m^{\mathcal{A}}(\tilde{a}_j).\mathit{lockid}^{\mathcal{A}} = \oslash$. Then, we can conclude that $\mathit{firstlocked}^{\mathcal{A}}(m^{\mathcal{A}}, p^{\mathcal{A}}) = \mathit{null}^{\mathcal{A}}$.
- If exists a $1 \le k \le n$ such that for all $1 \le j < k$, $m^{\mathcal{B}}(a_j^{\mathcal{B}}).\mathit{lockid}^{\mathcal{B}} = \oslash$ and $m^{\mathcal{B}}(a_k^{\mathcal{B}}).\mathit{lockid}^{\mathcal{B}} \ne \oslash$ then since by the construction of model $\mathcal{A}$, we have that $a^{\mathcal{A}} = a^{\mathcal{B}}$, we can say that $a^{\mathcal{A}} = a^{\mathcal{B}} = x \in X$. It then remains to verify whether

$$x = \mathit{firstlocked}^{\mathcal{B}}(m^{\mathcal{B}}, p^{\mathcal{B}}) \;\rightarrow\; x = \mathit{firstlocked}^{\mathcal{A}}(m^{\mathcal{A}}, \mathit{compress}(p^{\mathcal{B}}, X))$$

  By definition of *firstlocked* we have that $x = a_k^{\mathcal{B}}$ and by function κ and the construction of set X, we know that $a_k^{\mathcal{B}} \in X$. Let $[\tilde{a}_1, \ldots, \tilde{a}_i, \ldots, \tilde{a}_m] = \mathit{compress}(p^{\mathcal{B}}, X)$ such that $\tilde{a}_i = a_k^{\mathcal{B}}$. It is clear that $\tilde{a}_j \in X$ for all $1 \le j \le m$. Then, as *compress* preserves the order and for all $1 \le j < k$, $m^{\mathcal{B}}(a_j^{\mathcal{B}}).\mathit{lockid}^{\mathcal{B}} = \oslash$, we have that for all $1 \le j < i$, $m^{\mathcal{A}}(\tilde{a}_j).\mathit{lockid}^{\mathcal{A}} = \oslash$. Besides $m^{\mathcal{A}}(\tilde{a}_i).\mathit{lockid}^{\mathcal{A}} \ne \oslash$. Then:

$$\begin{aligned} \mathit{firstlocked}^{\mathcal{A}}(m^{\mathcal{A}}, \mathit{compress}(p^{\mathcal{B}}, X) &= \mathit{firstlocked}^{\mathcal{A}}(m^{\mathcal{A}}, [\tilde{a}_1, \ldots, \tilde{a}_m]) \\ &= \tilde{a}_i \\ &= a^{\mathcal{B}} \\ &= x \end{aligned}$$

□

Instead of proving that all accepting paths in the diagram are contained into the set of sequences satisfying formula $\psi(k)$, we show that the intersection with the negation of the formula is empty. We say that $\langle v_1, v_2, v_3\rangle$ interprets $\langle at\_l_{3..10}^{[k]}, at\_l_{11}^{[k]}, \mathit{IsLast}\rangle$. In fact *IsLast* should be decomposed into all its atomic subformulas. However, for the sake of simplicity we assume that an assignment to *IsLast* represents all necessarily assignments to its atomic subformulas in order to make *IsLast* predicate true. Then, since:

$$\neg\psi(k) = \Diamond\left(at\_l_{3..10}^{[k]} \wedge \mathit{IsLast} \wedge \left(\neg at\_l_{11}^{[k]}\ \mathcal{W}\ \neg\mathit{IsLast}\right)\right)$$

we have that:

$$\mathcal{L}^{P}\left(\neg\psi\left(k\right)\right) = \langle -,-,-\rangle^{*}\langle \mathrm{T},\mathrm{F},\mathrm{T}\rangle\left(\langle -,\mathrm{F},-\rangle^{\omega} \cup \langle -,\mathrm{F},-\rangle^{*}\langle -,-,\mathrm{F}\rangle\langle -,-,-\rangle^{\omega}\right)$$

while for our verification diagram we have

$$\left(\langle -, -, \text{F}\rangle^{+}\langle \text{T}, \text{F}, \text{T}\rangle^{*}\langle \text{F}, \text{T}, \text{T}\rangle^{+}\langle -, -, \text{T}\rangle^{+}\right)^{\omega}$$

Imagine we consider the sequence $\langle -, -, -\rangle^{*}\langle \text{T}, \text{F}, \text{T}\rangle\langle -, \text{F}, -\rangle^{\omega}$. Then, $\langle \text{T}, \text{F}, \text{T}\rangle$ should correspond to $\langle \text{T}, \text{F}, \text{T}\rangle^{*}$. Then, it is impossible to match $\langle -, \text{F}, -\rangle^{\omega}$ with any possible pattern in the accepting paths of the diagram. On the other hand, if we consider the sequence $\langle -, \text{F}, -\rangle^{*}\langle -, -, \text{F}\rangle\langle -, -, -\rangle^{\omega}$, notice that there is no possible way to match $\langle -, -, \text{F}\rangle$ with the accepting paths of the diagram. This way we have shown that both languages are disjoint.

B No Thread Overtake

In this section we proof that it is not possible in our implementation that once a thread has acquired a lock, then it is impossible for it to overtake another one. To carry out the proof, we first extend the code for concurrent lock-coupling singly-linked lists with two ghost variables. A global ghost variable *ticket* and a local ghost variable *myTicket*. The modifications need to be done on the *Node* data structure and on the *locate* function. An extended version of these two functions is depicted in Figure 9

The idea is that every time a thread gets the first lock in the list, it also gets a ticket number. The tickets are delivered by the data structure and they are strictly increasingly. This policy guarantees that no thread gets a duplicated ticket and ticket's number order threads according to the order in which they acquired their first lock. We want to assure that there is no overtake in the

```
class List {
    Node list;
    rgn r;
}

class Node {
    Value val;
    Node next;
    Lock lock;
}
```

(a) data structures

```
 1: prev := Head
 2: prev.lock()
          myTicket := ticket
          ticket := ticket + 1
 3: curr := prev.next
 4: curr.lock()
 5: while $curr.val < e$ do
 6:     prev.unlock()
 7:     prev := curr
 8:     curr := curr.next
 9:     curr.lock()
10: end while
11: return (prev, curr)
```

(b) *locate*

Fig. 9: Data structure and *locate* for preventing overtake

system. This means that, threads remains ordered as they got their ticket. For describing the formula that describes this condition we first define the *ahead* predicate defined over elements of T_{ID} and a memory M:

$$\begin{aligned} \mathit{ahead}(t_1, t_2) \hat{=} & \mathit{firstlocked}(M, \mathit{prev}^{[t_1]}) \neq \mathit{null} \wedge \\ & \mathit{firstlocked}(M, \mathit{prev}^{[t_2]}) \neq \mathit{null} \wedge \\ & \mathit{firstlocked}(M, \mathit{prev}^{[t_2]}) \in \mathit{addr2set}(M, \mathit{curr}^{[t_1]}) \end{aligned}$$

Roughly speaking, predicate *ahead* holds when both threads have at least one locked node and the first node locked (that is, the nearest to the head of the list) by t_2 is between the position of t_1 and the tail of the list.

We can now use *ahead* predicate to define the condition we want to verify. We define the formula φ by:

$$\varphi \hat{=} \mathit{myTicket}^{[t_1]} > \mathit{myTicket}^{[t_2]} \rightarrow \Box\left(\mathit{ahead}(t_1, t_2) \rightarrow \bigcirc\left(\mathit{ahead}(t_1, t_2)\right)\right)$$

Notice that φ is a system invariant. Then, we just need to verify that:

– it is satisfied by the system's initial condition, and
– all transitions preserve it.

It is easy to see that the initial condition satisfies φ. At such moment, no ticket number has been assigned nor any thread has acquired any lock. Then, formula φ is trivially satisfied.

Here we do not analyze all transitions, but we limit ourself to possible offending ones. These transitions are:

– When a threads acquires its first lock. For an arbitrary thread t, this transition corresponds to $\mathit{at\_locate}_{2}^{[t]}$. This transition clearly satisfies φ' because of, on one hand, value of *ticket* is strictly increasing. This guarantees that for all other thread s with locks in the list, $\mathit{myTicket}^{[t]} > \mathit{myTicket}^{[s]}$ and $\mathit{ahead}(t, s)$ hold, since $\mathit{firstlocked}(M, s) \in \mathit{addr2set}(M, \mathit{prev}(t))$
– When a thread gets a new lock. This modification is accomplished by the transitions $\mathit{at\_locate}_{4}^{[t]}$ and $\mathit{at\_locate}_{9}^{[t]}$.
– When a thread releases its last lock. This happens when transitions $\mathit{at\_add}_{11}^{[t]}$, $\mathit{at\_remove}_{10}^{[t]}$ or $\mathit{at\_search}_{8}^{[t]}$ are taken.
– When the position of a thread advances through the nodes in the lists. This progress corresponds to transitions $\mathit{at\_locate}_{3}^{[t]}$ and $\mathit{at\_locate}_{8}^{[t]}$.

C Supporting Invariants

To prove list preservation, we require to verify that the invariant $\psi = \Box(\mathit{List}(h, l.\mathit{list}.r))$ holds. However, to verify it, we require some extra supporting invariants. In this section, we provide a rigorous detail of such invariants.

We want to verify that $\forall \mathrm{pc}.\forall j.\psi\{\tau_{\mathrm{pc}}^{[j]}\}\psi'$. This means to verify whether:

$$\begin{array}{cc}
 & \theta \rightarrow \psi \\
\forall j & \psi\{\tau^{[j]}_{locate_1}\}\psi' \\
 & \vdots \\
\forall j & \psi\{\tau^{[j]}_{locate_{11}}\}\psi' \\
\forall j & \psi\{\tau^{[j]}_{add_1}\}\psi' \\
 & \vdots \\
\forall j & \psi\{\tau^{[j]}_{add_{12}}\}\psi' \\
\forall j & \psi\{\tau^{[j]}_{remove_1}\}\psi' \\
 & \vdots \\
\forall j & \psi\{\tau^{[j]}_{remove_{11}}\}\psi' \\
\forall j & \psi\{\tau^{[j]}_{search_1}\}\psi' \\
 & \vdots \\
\forall j & \psi\{\tau^{[j]}_{search_9}\}\psi' \\
\forall j & \psi\{\tau^{[j]}_{decide_1}\}\psi' \\
 & \vdots \\
\forall j & \psi\{\tau^{[j]}_{decide_5}\}\psi' \\
\hline
 & \Box(\psi)
\end{array}$$

Notice that it is not necessary to bear in mind all possible transitions. We require only to verify those on which an element involved in the *List* predicate is modified. More precisely, h, l and r. Notice that the only transitions that introduce some modification in these elements are τ_{add_5} and τ_{remove_4}. Moreover, we do not carry out the proof for all possible threads. Instead, under the assumption of a symmetric system, we do the proof for a single thread, generalizing the behavior of all of them.

Let's first consider transition τ_{remove_4} for an arbitrary thread j. The verification condition for such transition is:

$$\psi \wedge \begin{pmatrix} h'[a] = h[a] \wedge a \neq prev^{[j]} & \wedge \\ h'[prev^{[j]}].val = h[prev^{[j]}].val & \wedge \\ h'[prev^{[j]}].lock = h[prev^{[j]}].lock & \wedge \\ h'[prev^{[j]}].next = aux & \wedge \\ pres(\mathcal{V} - h[prev^{[j]}]) & \end{pmatrix} \rightarrow \psi'$$

However, in this verification condition we are not asserting that aux is in fact the node that follows $curr$. Therefore, we require to add some supporting invariants.

Let β be the following formula:

$$\beta(i) \hat{=} \begin{pmatrix} curr^{[i]} = prev^{[i]}.next & \wedge \\ prev^{[i]}.lock = i \wedge curr^{[i]}.lock = i & \wedge \\ at\_remove^{[i]}_{4} \rightarrow curr^{[i]}.next = aux^{[i]} & \end{pmatrix}$$

then, we define the supporting invariant ψ_1 parametrized by thread i by:

$$\psi_1(i) \hat{=} \Box \left(at\_remove^{[i]}_{1..4} \rightarrow \beta(i) \right)$$

Invariant ψ_1 establishes that before proceeding to remove a node from the list, *curr* pointer points to the node next to *prev*. Besides, it sets that *prev* and *curr* must be locked and once position at_remove_4 is reached, *aux* points to the node next to *curr*. This prevents us from constructing circular lists. Now, using ψ_1 it is possible to prove ψ as invariant, since the predicate *List* holds.

Let's now consider transition τ_{add_5} taken by an arbitrary thread j. The first supporting invariant we need is one similar to ψ_1, describing the position of *prev*, *curr* and *aux* before performing the operation of transition τ_{add_5}. We name such invariant ψ_2 and we define it by:

$$\psi_2(i) \hat{=} \Box \left(at\_add^{[i]}_{1..5} \rightarrow \beta(i) \right)$$

Besides, we require that cells returned by an invocation to **new** are kept in new addresses which must not be part of the list. This property is guaranteed by the semantic of the **new** operator and let us define the supporting invariant $\psi_3(i)$ that says:

$$\psi_3(i) \hat{=} \Box \left(at\_add^{[i]}_{4,5} \rightarrow \langle curr^{[i]} \rangle \# r \right)$$

Moreover, we also require that calls to **new** , performed by different threads, do also return disjoint addresses. Invariant ψ_4 describes this property:

$$\psi_4(i,j) \hat{=} \Box \left(i \neq j \wedge at\_add^{[i]}_{4,5} \wedge at\_add^{[i]}_{4,5} \rightarrow \langle aux^{[i]} \rangle \# \langle aux^{[j]} \rangle \right)$$