A Theory of Skiplists with Applications to the Verification of Concurrent Datatypes*

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 presents a theory of skiplists with a decidable satisfiability problem, and shows its applications to the verification of concurrent skiplist implementations. A skiplist is a data structure used to implement sets by maintaining several ordered singly-linked lists in memory, with a performance comparable to balanced binary trees. We define a theory capable of expressing the memory layout of a skiplist and show a decision procedure for the satisfiability problem of this theory. We illustrate the application of our decision procedure to the temporal verification of an implementation of concurrent lock-coupling skiplists. Concurrent lock-coupling skiplists are a particular version of skiplists where every node contains a lock at each possible level, reducing granularity of mutual exclusion sections.

The first contribution of this paper is the theory $\mathsf{TSL_K}$. $\mathsf{TSL_K}$ is a decidable theory capable of reasoning about list reachability, locks, ordered lists, and sublists of ordered lists. The second contribution is a proof that $\mathsf{TSL_K}$ enjoys a finite model property and thus it is decidable. Finally, we show how to reduce the satisfiability problem of quantifier-free $\mathsf{TSL_K}$ formulas to a combination of theories for which a many-sorted version of Nelson-Oppen can be applied.

1 Introduction

A skiplist [14] is a data structure that implements sets, maintaining several sorted singly-linked lists in memory. Skiplists are structured in multiple levels, where each level consists of a single linked list. The skiplist property establishes that the list at level $i+1$ is a sublist of the list at level i. Each node in a skiplist stores a value and at least the pointer corresponding to the lowest level list. Some nodes also contain pointers at higher levels, pointing to the next element present at that level. The advantage of skiplists is that they are simpler and more efficient to implement than search trees, and search is still (probabilistically) logarithmic.

* This work was funded in part by the EU project FET IST-231620 *HATS*, MICINN project TIN-2008-05624 *DOVES*, CAM project S2009TIC-1465 *PROMETIDOS*, and by the COST Action IC0901 *Rich ModelToolkit-An Infrastructure for Reliable Computer Systems.*

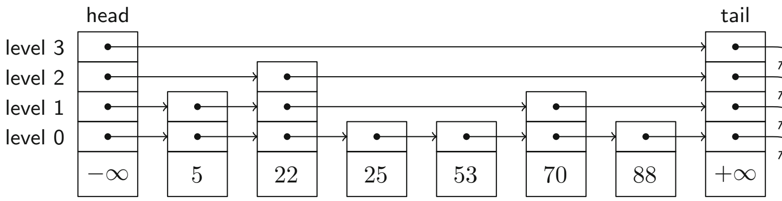

Fig. 1. A skiplist with 4 levels

Consider the skiplist shown in Fig. 1. Contrary to single-linked lists implementations, higher-level pointers allow to *skip* many elements during the search. A search is performed from left to right in a top down fashion, progressing as much as possible in a level before descending. For instance, in Fig. 1 a search for value 88 starts at level 3 of node *head*. From *head* the pointer at level 3 reaches *tail* with value $+\infty$, which is greater than 88. Hence the search algorithm moves down one level at *head* to level 2. The successor at level 2 contains value 22, which is smaller than 88, so the search continues at level 2 until a node containing a greater value is found. At that moment, the search moves down one further level again. The expected logarithmic search follows from the probability of any given node occurs at a certain level decreasing by $1/2$ as a level increases (see [14] for an analysis of the running time of skiplists).

We are interested in the formal verification of implementations of skiplists, in particular in temporal verification (liveness and safety properties) of sequential and concurrent implementations. This verification activity requires to deal with unbounded mutable data. One popular approach to verification of heap programs is Separation Logic [17]. Skiplists, however, are problematic for separation-like approaches due to the aliasing and memory sharing between nodes at different levels. Based on the success of separation logic some researchers have extended this logic to deal with concurrent programs [23, 7], but concurrent datatypes follow a programming style in which the activities of concurrent threads are not structured according to critical regions with memory footprints. In these approaches based on Separation Logic memory regions are implicitly declared (hidden in the separation conjunction), which makes the reasoning about unstructured concurrency more cumbersome.

Most of the work in formal verification of pointer programs follows program logics in the Hoare tradition, either using separation logic or with specialized logics to deal with the heap and pointer structures [9, 24, 3]. However, extending these logics to deal with concurrent programs is hard, and though some success has been accomplished it is still an open area of research, particularly for liveness.

Continuing our previous work [18] we follow a complementary approach. We start from temporal deductive verification in the style of Manna-Pnueli [11], in particular using general verification diagrams [5, 19] to deal with concurrency. This style of reasoning allows a clean separation in a proof between the temporal part (why the interleavings of actions that a set of threads can perform satisfy a certain property) with the underlying data being manipulated. A verification diagram decomposes a formal proof into a finite collection of verification conditions (VC), each of which corresponds to the effect that a small step in the program has in the data. To automatize the process of checking the proof represented by a verification diagram it is necessary to use decision procedures for the kind of data structures manipulated. This paper studies the automatic verification of VCs for the case of skiplists.

Logics like [9, 24, 3] are very powerful to describe pointer structures, but they require the use of quantifiers to reach their expressive power. Hence, these logics preclude a combination a-la Nelson-Oppen [12] or BAPA [8] with other aspects of the program state. Instead, our solution starts from a quantifier-free theory of single-linked lists [16], and extends it in a non trivial way with order and sublists of ordered lists. The logic obtained can express skiplist-like properties without using quantifiers, allowing the combination with other theories. Proofs for an unbounded number of threads are achieved by parameterizing verification diagrams, splitting cases for interesting threads and producing a single verification condition to generalize the remaining cases. However, in this paper we mainly focus in the decision procedure. Since we want to verify concurrent lock-based implementations we extend the basic theory with locks, lock ownership, and sets of locks (and in general stores of locks). The decision procedure that we present here supports the manipulation of explicit regions, as in regional logic [2] equipped with *masked regions*, which enables reasoning about disjoint portions of the same memory cell. We use masked regions to "separate" different levels of the same skiplist node.

We call our theory TSL_K, that allows to reason about skiplists of height at most K. To illustrate the use of this theory, we sketch the proof of termination of every invocation of an implementation of a lock-coupling concurrent skiplist.

The rest of the paper is structured as follows. Section 2 presents lock-coupling concurrent skiplists. Section 3 introduces TSL_K. Section 4 shows that TSL_K is decidable by proving a finite model property theorem, and describes how to construct a more efficient decision procedure using the many-sorted Nelson-Oppen combination method. Finally, Section 5 concludes the paper. Some proofs are missing due to space limitation.

2 Fine-Grained Concurrent Lock-Coupling Skiplists

In this section we present a simple concurrent implementation of skiplists that uses lock-coupling [6] to acquire and release locks. This implementation can be seen as an extension of concurrent lock-coupling lists [6, 23] to multiple layers of pointers. This algorithm imposes a locking discipline, consisting of acquiring locks as the search progresses, and releasing a node's lock only after the lock of the next node in the search process has been acquired. A naïve implementation of this solution would equip each node with a single lock, allowing multiple threads to access simultaneously different nodes in the list, but protecting concurrent accesses to two different fields of the same node. The performance can

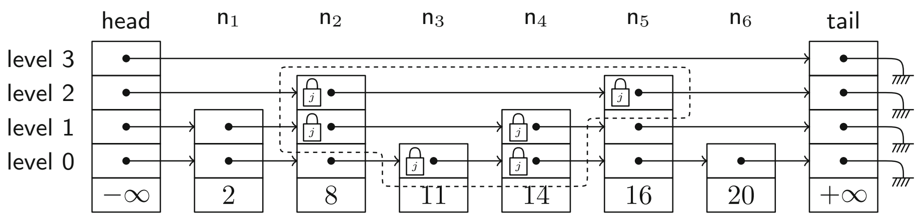

Fig. 2. A skiplist with the masked region given by the fields locked by thread j

be improved by carefully allowing multiple threads to simultaneously access the same node at different levels. We study here an implementation of this faster solution in which each node is equipped with a different lock at each level. At execution time a thread uses locks to protect the access to only some fields of a given node. A precise reasoning framework needs to capture those portions of the memory protected by a set of locks, which may include only *parts* of a node. Approaches based on strict separation (separation logic [17] or regional logic [2]) do not provide the fine grain needed to reason about individual fields of shared objects. Here, we introduce the concept of *masked regions* to describe regions and the fields within. A masked region consists of a set of pairs formed by a region (*Node* cell) and a field (a skiplist level): **mrgn** $\hat{=}$ $2^{Node \times \mathbb{N}}$ We call the field a mask, since it identifies which part of the object is relevant. For example, in Fig. 2 the region within dots represents the area of the memory that thread j is protecting. This portion of the memory is described by the masked region $\{(n_2, 2), (n_5, 2), (n_2, 1),\ (n_4, 1), (n_3, 0), (n_4, 0)\}$. As with regional logic, an empty set intersection denotes separation. In masked regions two memory nodes at different levels do not overlap. This notion is similar to data-groups [10].

Fig. 3(a) contains the pseudo-code declaration of the *Node* and *SkipList* classes. Throughout the paper we use `//@` to denote ghost code added for verification purposes. Note that the structure is parametrized by a value K, which determines the maximum possible level of any node in the modeled skiplist. The fields *val* and *key* in the class *Node* contains the value and the key of the element used to order them. Then, we can store key-value pairs, or use the skiplist as a set of arbitrary elements as long as the key can be used to compare. The *next* array stores the pointers to the next nodes at each of the possible K different levels of the skiplist. Finally, the *lock* array keeps the locks, one for each level, protecting the access to the corresponding *next* field. The *SkipList* class contains two pointer fields: *head* and *tail* plus a ghost variable field r. Field *head* points to the first node of the skiplist, and *tail* to the last one. Variable r, only used for verification purposes, keeps the (masked) region represented by all nodes in the skiplist with all their levels. In this implementation, *head* and *tail* are sentinel nodes, with $key = -\infty$ and $key = +\infty$, respectively. For simplicity, these nodes are not eliminated during the execution and their *val* field remains unchanged.

```
class $Node$ {                                  class $SkipList$ {
    $Value\ val$;                                   $Node^*\ head$;
    $Key\ key$;                                     $Node^*\ tail$;
    $Array\langle Node^* \rangle(\mathsf{K})\ next$;             //@ mrgn $r$;
    $Array\langle Node^* \rangle(\mathsf{K})\ lock$;         }
}
```

(a) data structures

```
procedure INSERT($SkipList\ sl$, $Value\ newval$)
    $Vector\langle Node^* \rangle upd[0..K-1]$                //@ mrgn $m_r := \emptyset$
    $lvl := randomLevel(K)$
    $Node^* pred := sl.head$
    $pred.locks[K-1].lock()$                       //@ $m_r := m_r \cup \{(pred, K-1)\}$
    $Node^* curr := pred.next[K-1]$
    $curr.locks[K-1].lock()$                       //@ $m_r := m_r \cup \{(curr, K-1)\}$
    for $i := K-1$ downto 0 do
        if $i < K-1$ then
            $pred.locks[i].lock()$                 //@ $m_r := m_r \cup \{(pred, i)\}$
            if $i \geq lvl$ then
                $curr.locks[i+1].unlock()$         //@ $m_r := m_r - \{(curr, i+1)\}$
                $pred.locks[i+1].unlock()$         //@ $m_r := m_r - \{(pred, i+1)\}$
            end if
            $curr := pred.next[i]$
            $curr.locks[i].lock()$                 //@ $m_r := m_r \cup \{(curr, i)\}$
        end if
        while $curr.val < newval$ do
            $pred.locks[i].unlock()$               //@ $m_r := m_r - \{(pred, i)\}$
            $pred := curr$
            $curr := pred.next[i]$
            $curr.locks[i].lock()$                 //@ $m_r := m_r \cup \{(curr, i)\}$
        end while
        $upd[i] := pred$
    end for
    $Bool\ valueWasIn := (curr.val = newval)$
    if $valueWasIn$ then
        for $i := 0$ to $lvl$ do
            $upd[i].next[i].locks[i].unlock()$     //@ $m_r := m_r - \{(upd[i].next[i], i)\}$
            $upd[i].locks[i].unlock()$             //@ $m_r := m_r - \{(upd[i], i)\}$
        end for
    else
        $x := CreateNode(lvl, newval)$
        for $i := 0$ to $lvl$ do
            $x.next[i] := upd[i].next[i]$
            $upd[i].next[i] := x$                  //@ $sl.r := sl.r \cup \{(x, i)\}$
            $x.next[i].locks[i].unlock()$          //@ $m_r := m_r - \{(x.next[i], i)\}$
            $upd[i].locks[i].unlock()$             //@ $m_r := m_r - \{(upd[i], i)\}$
        end for
    end if
    return $\neg valueWasIn$
end procedure
```

(b) insertion algorithm

Fig. 3. Data structure and insert algorithm for concurrent lock-coupling skiplist

Fig. 3(b) shows the implementation of the insertion algorithm. The algorithms for searching and removing are similar, and omitted due to space limitations. The ghost variable m_r stores a masked region containing all the nodes and fields currently locked by the running thread. The set operations $\cup$ and $-$ are used for the manipulation of the corresponding sets of pairs.

Let *sl* be a pointer to a skiplist (an instance of the class described in Fig. 3(a)). The following predicate captures whether *sl* points to a well-formed skiplist of height 4 or less:

$$\begin{aligned}
& \mathit{SkipList}_4(h, sl : \mathit{SkipList}) \ \hat{=}\ \mathit{OList}(h, sl, 0)\ \wedge && (1)\\
& \quad \begin{pmatrix} h[sl].\mathit{tail}.\mathit{next}[0] = \mathit{null} \wedge h[sl].\mathit{tail}.\mathit{next}[1] = \mathit{null} \\ h[sl].\mathit{tail}.\mathit{next}[2] = \mathit{null} \wedge h[sl].\mathit{tail}.\mathit{next}[3] = \mathit{null} \end{pmatrix} \wedge && (2)\\
& \quad \begin{pmatrix} \mathit{SubList}(h, sl.\mathit{head}, sl.\mathit{tail}, 1, sl.\mathit{head}, sl.\mathit{tail}, 0) \wedge \\ \mathit{SubList}(h, sl.\mathit{head}, sl.\mathit{tail}, 2, sl.\mathit{head}, sl.\mathit{tail}, 1) \wedge \\ \mathit{SubList}(h, sl.\mathit{head}, sl.\mathit{tail}, 3, sl.\mathit{head}, sl.\mathit{tail}, 2) \end{pmatrix} && (3)
\end{aligned}$$

The predicate *OList* in (1) describes that in heap h, the pointer *sl* is an ordered linked-lists when repeatedly following the pointers at level 0 starting at *head*. The predicate (2) indicates all levels are *null* terminated, and (3) indicates that each level is in fact a sublist of its nearest lower level. Predicates of this kind also allow to express the effect of programs statements via first order transition relations. Consider the statement at line 36 in program *insert* shown in Fig. 3(b) on a skiplist of height 4, taken by thread with id t. This transition corresponds to a new node x at level i being connected to the skiplist. If the memory layout from pointer *sl* is that of a skiplist before the statement at line 36 is executed, then it is also a skiplist after the execution:

$$\mathit{SkipList}_4(h, sl) \wedge \varphi_{aux} \wedge \rho_{36}^{[t]}(V, V') \to \mathit{SkipList}_4(h', sl')$$

The effect of the statement at line 36 is represented by the first-order transition relation $\rho_{36}^{[t]}$. To ensure this property, i is required to be a valid level, and the key of the nodes that will be pointing to x must be lower than the key of node x. Moreover, the masked region of locked nodes remains unchanged. Predicate φ_{aux} contains support invariants. For simplicity, we use *prev* for $\mathit{upd}^{[t]}[i]$. Then, the full verification condition is:

$$\mathit{SkipList}_4(h, sl) \wedge \begin{pmatrix} x.\mathit{key} = \mathit{newval} & \wedge \\ \mathit{prev}.\mathit{key} < \mathit{newval} & \wedge \\ x.\mathit{next}[i].\mathit{key} > \mathit{newval} & \wedge \\ \mathit{prev}.\mathit{next}[i] = x.\mathit{next}[i] & \wedge \\ (x, i) \notin sl.r \wedge 0 \leq i \leq 3 & \end{pmatrix} \wedge \begin{pmatrix} \mathit{at}_{36}[t] & \wedge \\ \mathit{prev}'.\mathit{next}[i] = x & \wedge \\ \mathit{at}'_{37}[t] & \wedge \\ h' = h \wedge sl = sl' & \wedge \\ x' = x & \ldots \end{pmatrix} \to$$
$$\mathit{SkipList}_4(h', sl')$$

As usual, we use primed variables to describe the values of the variables after the transition is taken. Section 4 contains a full verification condition. This example illustrates that to be able to automatically prove VCs for the verification of skiplist manipulating algorithms, we require a theory that allows to reason about heaps, addresses, nodes, masked regions, ordered lists and sublists.

3 The Theory of Concurrent Skiplists of Height K: $\mathsf{TSL_K}$

We build a decision procedure to reason about skiplist of height K combining different theories, aiming to represent pointer data structures with a skiplist layout, masked regions and locks. We extend the Theory of Concurrent Linked Lists ($\mathsf{TLL3}$) [18], a decidable theory that includes reachability of concurrent list-like structures in the following way:

- each node is equipped with a *key* field, used to reason about element's order.
- the reasoning about single level lists is extended to all the K levels.
- we extend the theory of regions with masked regions.
- lists are extended to ordered lists and sub-paths of ordered lists.

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 $\Sigma_2 = (S_2, F_2, P_2)$, we define $\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2, F_1 \cup F_2, P_1 \cup P_2)$. 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 from symbols in Σ to values. 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 a signature, $\mathcal{A}$ an 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 skiplists of height K is defined as $\mathsf{TSL_K} = (\Sigma_{\mathsf{TSL_K}}, \mathbf{TSLK})$, where

$$\begin{aligned}\Sigma_{\mathsf{TSL_K}} = {} & \Sigma_{\mathsf{level_K}} \cup \Sigma_{\mathsf{ord}} \cup \Sigma_{\mathsf{thid}} \cup \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{reach}} \cup \\ & \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{setth}} \cup \Sigma_{\mathsf{mrgn}} \cup \Sigma_{\mathsf{bridge}}\end{aligned}$$

The signature of $\mathsf{TSL_K}$ is shown in Fig. 4. **TSLK** is the class of $\Sigma_{\mathsf{TSL_K}}$-structures satisfying the conditions depicted in Fig. 5. The symbols of Σ_{set} and Σ_{setth} follow their standard interpretation over sets of addresses and thread identifiers resp.

Informally, sort addr represents addresses; elem the universe of elements that can be stored in the skiplist; ord the ordered keys used to preserve a strict order in the skiplist; thid thread identifiers; $\mathsf{level_K}$ the levels of a skiplist; cell models *cells* representing a node in a skiplist; mem models the heap, mapping addresses to cells or to *null*; path describes finite sequences of non-repeating addresses to

Signt	Sort	Functions	Predicates
$\Sigma_{\mathsf{level_K}}$	$\mathsf{level_K}$	$0, 1, \ldots, \mathsf{K}-1 : \mathsf{level_K}$	$<: \mathsf{level_K} \times \mathsf{level_K}$
Σ_{ord}	ord	$-\infty, +\infty : \mathsf{ord}$	$\preceq : \mathsf{ord} \times \mathsf{ord}$
Σ_{thid}	thid	$\oslash : \mathsf{thid}$	
Σ_{cell}	cell elem ord addr thid	$\mathit{error} : \mathsf{cell}$ $\mathit{mkcell} : \mathsf{elem} \times \mathsf{ord} \times \mathsf{addr}^{\mathsf{K}} \times \mathsf{thid}^{\mathsf{K}} \rightarrow \mathsf{cell}$ $\_.\mathit{data} : \mathsf{cell} \rightarrow \mathsf{elem}$ $\_.\mathit{key} : \mathsf{cell} \rightarrow \mathsf{ord}$ $\_.\mathit{next}[\_] : \mathsf{cell} \times \mathsf{level_K} \rightarrow \mathsf{addr}$ $\_.\mathit{lockid}[\_] : \mathsf{cell} \times \mathsf{level_K} \rightarrow \mathsf{thid}$ $\_.\mathit{lock}[\_] : \mathsf{cell} \times \mathsf{level_K} \rightarrow \mathsf{thid} \rightarrow \mathsf{cell}$ $\_.\mathit{unlock}[\_] : \mathsf{cell} \times \mathsf{level_K} \rightarrow \mathsf{cell}$	
Σ_{mem}	mem addr cell	$\mathit{null} : \mathsf{addr}$ $\_[\_] : \mathsf{mem} \times \mathsf{addr} \rightarrow \mathsf{cell}$ $\mathit{upd} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{cell} \rightarrow \mathsf{mem}$	
Σ_{reach}	mem addr path	$\epsilon : \mathsf{path}$ $[\_] : \mathsf{addr} \rightarrow \mathsf{path}$	$\mathit{append} : \mathsf{path} \times \mathsf{path} \times \mathsf{path}$ $\mathit{reach}_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr}$ $\times \mathsf{level_K} \times \mathsf{path}$
Σ_{set}	addr set	$\emptyset : \mathsf{set}$ $\{\_\} : \mathsf{addr} \rightarrow \mathsf{set}$ $\cup, \cap, \setminus : \mathsf{set} \times \mathsf{set} \rightarrow \mathsf{set}$	$\in : \mathsf{addr} \times \mathsf{set}$ $\subseteq : \mathsf{set} \times \mathsf{set}$
Σ_{setth}	thid setth	$\emptyset_T : \mathsf{setth}$ $\{\_\}_T : \mathsf{thid} \rightarrow \mathsf{setth}$ $\cup_T, \cap_T, \setminus_T : \mathsf{setth} \times \mathsf{setth} \rightarrow \mathsf{setth}$	$\in_T : \mathsf{thid} \times \mathsf{setth}$ $\subseteq_T : \mathsf{setth} \times \mathsf{setth}$
Σ_{mrgn}	mrgn addr $\mathsf{level_K}$	$\mathbf{emp}_{\mathsf{mr}} : \mathsf{mrgn}$ $\langle \_, \_ \rangle_{\mathsf{mr}} : \mathsf{addr} \times \mathsf{level_K} \rightarrow \mathsf{mrgn}$ $\cup_{\mathsf{mr}}, \cap_{\mathsf{mr}}, -_{\mathsf{mr}} : \mathsf{mrgn} \times \mathsf{mrgn} \rightarrow \mathsf{mrgn}$	$\in_{\mathsf{mr}} : \mathsf{addr} \times \mathsf{level_K} \times \mathsf{mrgn}$ $\subseteq_{\mathsf{mr}} : \mathsf{mrgn} \times \mathsf{mrgn}$ $\#_{\mathsf{mr}} : \mathsf{mrgn} \times \mathsf{mrgn}$
Σ_{bridge}	mem addr set path	$\mathit{path2set} : \mathsf{path} \rightarrow \mathsf{set}$ $\mathit{addr2set}_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{level_K} \rightarrow \mathsf{set}$ $\mathit{getp}_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level_K} \rightarrow \mathsf{path}$ $\mathit{fstlock}_{\mathsf{K}} : \mathsf{mem} \times \mathsf{path} \times \mathsf{level_K} \rightarrow \mathsf{addr}$	$\mathit{ordList} : \mathsf{mem} \times \mathsf{path}$

Fig. 4. The signature of the $\mathsf{TSL_K}$ theory

model non-cyclic list paths; set models sets of addresses – also known as regions –, while setth models sets of thread identifiers and mrgn masked regions.

$\Sigma_{\mathsf{level_K}}$ contains symbols for level identifiers 0, 1, ..., $\mathsf{K}-1$ and their conventional order. Σ_{ord} contains two special elements $-\infty$ and ∞ for the lowest and highest values in the order $\preceq$. Σ_{thid} only contains, besides $=$ and $\neq$ as for all the other theories, a special constant $\oslash$ to represent the absence of a thread identifier. Σ_{cell} contains the constructors and selectors for building and inspecting

Interpret. of sorts: addr, elem, thid, $\mathsf{level}_\mathsf{K}$, ord, cell, mem, path, set, setth and mrgn

Each sort σ in $\Sigma_{\mathsf{TSL_K}}$ is mapped to a non-empty set $\mathcal{A}_\sigma$ such that:	
(a) $\mathcal{A}_\mathsf{addr}$ and $\mathcal{A}_\mathsf{elem}$ are discrete sets	(b) $\mathcal{A}_\mathsf{thid}$ is a discrete set containing $\oslash$
(c) $\mathcal{A}_{\mathsf{level_K}}$ is the finite collection 0,. . . ,K-1	(d) $\mathcal{A}_\mathsf{ord}$ is a total ordered set
(e) $\mathcal{A}_\mathsf{cell} = \mathcal{A}_\mathsf{elem} \times \mathcal{A}_\mathsf{ord} \times \mathcal{A}_\mathsf{addr}^\mathsf{K} \times \mathcal{A}_\mathsf{thid}^\mathsf{K}$	(f) $\mathcal{A}_\mathsf{mem} = \mathcal{A}_\mathsf{cell}^{\mathcal{A}_\mathsf{addr}}$
(g) $\mathcal{A}_\mathsf{path}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_\mathsf{addr}$	(h) $\mathcal{A}_\mathsf{set}$ is the power-set of $\mathcal{A}_\mathsf{addr}$ (i) $\mathcal{A}_\mathsf{setth}$ is the power-set of $\mathcal{A}_\mathsf{thid}$
(j) $\mathcal{A}_\mathsf{mrgn}$ is the power-set of $\mathcal{A}_\mathsf{addr} \times \mathcal{A}_{\mathsf{level_K}}$	

Signature	Interpretation
Σ_ord	$x \preceq^\mathcal{A} y \wedge y \preceq^\mathcal{A} x \rightarrow x = y$ $x \preceq^\mathcal{A} y \vee y \preceq^\mathcal{A} x$ for any $x, y, z \in \mathcal{A}_\mathsf{ord}$ $x \preceq^\mathcal{A} y \wedge y \preceq^\mathcal{A} z \rightarrow x \preceq^\mathcal{A} z$ $-\infty^\mathcal{A} \preceq^\mathcal{A} x \wedge x \preceq^\mathcal{A} +\infty^\mathcal{A}$
Σ_cell	– $mkcell^\mathcal{A}(e, k, \overrightarrow{a}, \overrightarrow{t}) = \langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle$ – $error^\mathcal{A}.next^\mathcal{A} = null^\mathcal{A}$ – $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.data^\mathcal{A} = e$ – $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.key^\mathcal{A} = k$ – $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.next^\mathcal{A}[j] = a_j$ – $\langle e, k, \overrightarrow{a}, \overrightarrow{t} \rangle.lockid^\mathcal{A}[j] = t_j$ – $\langle e, k, \overrightarrow{a}, ...t_{j-1}, t_j, t_{j+1}... \rangle.lock^\mathcal{A}[j](t') = \langle e, k, \overrightarrow{a}, ...t_{j-1}, t', t_{j+1}... \rangle$ – $\langle e, k, \overrightarrow{a}, ...t_{j-1}, t_j, t_{j+1}... \rangle.unlock^\mathcal{A}[j] = \langle e, k, \overrightarrow{a}, ...t_{j-1}, \oslash, t_{j+1}... \rangle$ for each $e \in \mathcal{A}_\mathsf{elem}$, $k \in \mathcal{A}_\mathsf{ord}$, $t_0, \ldots, t_j, t_{j+1}, t_{j-1}, t' \in \mathcal{A}_\mathsf{thid}$, $\overrightarrow{a} \in \mathcal{A}_\mathsf{addr}^\mathsf{K}$, $\overrightarrow{t} \in \mathcal{A}_\mathsf{thid}^\mathsf{K}$ and $j \in \mathcal{A}_{\mathsf{level_K}}$
Σ_mem	$m[a]^\mathcal{A} = m(a)$ $upd^\mathcal{A}(m, a, c) = m_{a \mapsto c}$ $m^\mathcal{A}(null^\mathcal{A}) = error^\mathcal{A}$ for each $m \in \mathcal{A}_\mathsf{mem}$, $a \in \mathcal{A}_\mathsf{addr}$ and $c \in \mathcal{A}_\mathsf{cell}$
Σ_reach	– $\epsilon^\mathcal{A}$ is the empty sequence – $[i]^\mathcal{A}$ is the sequence containing $i \in \mathcal{A}_\mathsf{addr}$ as the only element – $([i_1 \mathinner{..} i_n], [j_1 \mathinner{..} j_m], [i_1 \mathinner{..} i_n, j_1 \mathinner{..} j_m]) \in append^\mathcal{A}$ iff $i_k \neq j_l$. – $(m, a_{init}, a_{end}, l, p) \in reach_\mathsf{K}^\mathcal{A}$ iff $a_{init} = a_{end}$ and $p = \epsilon$, or there exist addresses $a_1, \ldots, a_n \in \mathcal{A}_\mathsf{addr}$ such that: (a) $p = [a_1 \mathinner{..} a_n]$ (c) $m(a_r).next^\mathcal{A}[l] = a_{r+1}$, for $r < n$ (b) $a_1 = a_{init}$ (d) $m(a_n).next^\mathcal{A}[l] = a_{end}$
Σ_mrgn	– $\mathbf{emp}_\mathsf{mr}^\mathcal{A} = \emptyset$ – $r \cup_\mathsf{mr}^\mathcal{A} s = r \cup s$ – $(a, j) \in_\mathsf{mr}^\mathcal{A} r \leftrightarrow (a, j) \in r$ – $\langle a, j \rangle_\mathsf{mr}^\mathcal{A} = \{(a, j)\}$ – $r \cap_\mathsf{mr}^\mathcal{A} s = r \cap s$ – $r \subseteq_\mathsf{mr}^\mathcal{A} s \leftrightarrow r \subseteq s$ – $r -_\mathsf{mr}^\mathcal{A} s = r \setminus s$ – $r \#_\mathsf{mr}^\mathcal{A} s \leftrightarrow r \cap_\mathsf{mr}^\mathcal{A} s = \mathbf{emp}_\mathsf{mr}^\mathcal{A}$ for each $a \in \mathcal{A}_\mathsf{addr}$, $j \in \mathcal{A}_{\mathsf{level_K}}$ and $r, s \in \mathcal{A}_\mathsf{mrgn}$
Σ_bridge	– $path2set^\mathcal{A}(p) = \{a_1, \ldots, a_n\}$ for $p = [a_1, \ldots, a_n] \in \mathcal{A}_\mathsf{path}$ – $addr2set_\mathsf{K}{}^\mathcal{A}(m, a, l) = \{a' \mid \exists p \in \mathcal{A}_\mathsf{path} \;.\; (m, a, a', l, p) \in reach_\mathsf{K}\}$ – $getp_\mathsf{K}{}^\mathcal{A}(m, a_{init}, a_{end}, l) = \begin{cases} p & \text{if } (m, a_{init}, a_{end}, l, p) \in reach_\mathsf{K}{}^\mathcal{A} \\ \epsilon & \text{otherwise} \end{cases}$ for each $m \in \mathcal{A}_\mathsf{mem}$, $p \in \mathcal{A}_\mathsf{path}$, $l \in \mathcal{A}_{\mathsf{level_K}}$ and $a_{init}, a_{end} \in \mathcal{A}_\mathsf{addr}$ – $fstlock^\mathcal{A}(m, [a_1 \mathinner{..} a_n], l) = \begin{cases} a_k & \text{if there is } k \leq n \text{ such that} \\ & \text{for all } j < k, m[a_j].lockid[l] = \oslash \\ & \text{and } m[a_k].lockid[l] \neq \oslash \\ null & \text{otherwise} \end{cases}$ – $ordList^\mathcal{A}(m, p)$ iff $p = \epsilon$ or $p = [a]$ or $p = [a_1 \mathinner{..} a_n]$ with $n \geq 2$ and $m(a_i).key^\mathcal{A} \preceq m(a_{i+1}).key^\mathcal{A}$ for all $1 \leq i < n$, for any $m \in \mathcal{A}_\mathsf{mem}$

Fig. 5. Characterization of a $\mathsf{TSL_K}$-interpretation $\mathcal{A}$

cells, including *error* for incorrect dereferences. Σ_{mem} is the signature for heaps, with the usual memory access and single memory mutation functions. Σ_{set} and Σ_{setth} are theories of sets of addresses and thread ids resp. Σ_{mrgn} is the theory of masked regions. The signature Σ_{reach} contains predicates to check reachability of address using paths at different levels, while Σ_{bridge} contains auxiliary functions and predicates to manipulate and inspect paths and locks.

4 Decidability of $\mathsf{TSL_K}$

We show that $\mathsf{TSL_K}$ is decidable by proving that it enjoys the finite model property with respect to its sorts, and exhibiting upper bounds for the sizes of the domains of a small interpretation of a satisfiable formula.

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.*

The fact that $\mathsf{TSL_K}$ has the finite model property with respect to domains elem, addr, ord, $\mathsf{level_K}$ and thid, implies that $\mathsf{TSL_K}$ is decidable by enumerating all possible $\Sigma_{\mathsf{TSL_K}}$-structures up to a certain cardinality. We now define the set of normalized $\mathsf{TSL_K}$-literals.

Definition 2 ($\mathsf{TSL_K}$-normalized literals). *A $\mathsf{TSL_K}$-literal is normalized if it is a flat literal of the form:*

$$\begin{array}{lll}
e_1 \neq e_2 & a_1 \neq a_2 & l_1 \neq l_2 \\
a = null & c = error & c = rd(m,a) \\
k_1 \neq k_2 & k_1 \preceq k_2 & m_2 = upd(m_1,a,c) \\
\multicolumn{3}{l}{c = mkcell(e,k,a_0,\ldots,a_{K-1},t_0,\ldots,t_{K-1})} \\
s = \{a\} & s_1 = s_2 \cup s_3 & s_1 = s_2 \setminus s_3 \\
g = \{t\}_T & g_1 = g_2 \cup_T g_3 & g_1 = g_2 \setminus_T g_3 \\
r = \langle a, l\rangle_{\mathsf{mr}} & r_1 = r_2 \cup_{\mathsf{mr}} r_3 & r_1 = r_2 -_{\mathsf{mr}} r_3 \\
p_1 \neq p_2 & p = [a] & p_1 = rev(p_2) \\
s = path2set(p) & append(p_1,p_2,p_3) & \neg append(p_1,p_2,p_3) \\
s = addr2set_K(m,a,l) & p = getp_K(m,a_1,a_2,l) & \\
t_1 \neq t_2 & a = fstlock\,(m,p,l) & ordList(m,p)
\end{array}$$

where e, e_1 and e_2 are elem-variables; a, a_0, a_1, $a_2,\ldots,a_{K-1}$ 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; s, s_1, s_2 and s_3 are set-variables; g, g_1, g_2 and g_3 are setth-variables; r, r_1, r_2 and r_3 are mrgn-variables; k, k_1 and k_2 are ord-variables; l, l_1 and l_2 are $\mathsf{level_K}$-variables and t, t_0, t_1, $t_2,\ldots,t_{K-1}$ are thid-variables.

Lemma 1. *Every $\mathsf{TSL_K}$-formula is equivalent to a collection of conjunctions of normalized $\mathsf{TSL_K}$-literals.*

Proof (sketch). First, transform a formula in disjunctive normal form. Then each conjunct can be normalized introducing auxiliary fresh variables when necessary.

The phase of normalizing a formula is commonly known [15] as the "variable abstraction phase". Note that normalized literals belong to just one theory.

Consider an arbitrary $\mathsf{TSL_K}$-interpretation $\mathcal{A}$ satisfying a conjunction of normalized $\mathsf{TSL_K}$-literals Γ. We show that if $\mathcal{A}$ consists of domains $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{addr}}$, $\mathcal{A}_{\mathsf{thid}}$, $\mathcal{A}_{\mathsf{level_K}}$ and $\mathcal{A}_{\mathsf{ord}}$ then there are finite sets $\mathcal{B}_{\mathsf{elem}}$, $\mathcal{B}_{\mathsf{addr}}$, $\mathcal{B}_{\mathsf{thid}}$, $\mathcal{B}_{\mathsf{level_K}}$ and $\mathcal{B}_{\mathsf{ord}}$ with bounded cardinalities, where the finite bound on the sizes can be computed from Γ. Such sets can in turn be used to obtain a finite interpretation $\mathcal{B}$ satisfying Γ, since all the other sorts are bounded by the sizes of these sets.

Lemma 2 (Finite Model Property). *Let Γ be a conjunction of normalized $\mathsf{TSL_K}$-literals. Let $\overline{e} = |V_{\mathsf{elem}}(\Gamma)|$, $\overline{a} = |V_{\mathsf{addr}}(\Gamma)|$, $\overline{m} = |V_{\mathsf{mem}}(\Gamma)|$, $\overline{p} = |V_{\mathsf{path}}(\Gamma)|$, $\overline{t} = |V_{\mathsf{thid}}(\Gamma)|$ and $\overline{o} = |V_{\mathsf{ord}}(\Gamma)|$. Then the following are equivalent:*

1. *Γ is $\mathsf{TSL_K}$-satisfiable;*
2. *Γ is true in a $\mathsf{TSL_K}$ interpretation $\mathcal{B}$ such that*

$$|\mathcal{B}_{\mathsf{addr}}| \leq \overline{a} + 1 + \overline{m}\,\overline{a}\,\mathsf{K} + \overline{p}^2 + \overline{p}^3 + (\mathsf{K}+2)\overline{m}\,\overline{p} \qquad |\mathcal{B}_{\mathsf{elem}}| \leq \overline{e} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}|$$
$$|\mathcal{B}_{\mathsf{thid}}| \leq \overline{t} + \mathsf{K}\,\overline{m}\,|\mathcal{B}_{\mathsf{addr}}| + 1 \qquad |\mathcal{B}_{\mathsf{ord}}| \leq \overline{o} + \overline{m}\ |\mathcal{B}_{\mathsf{addr}}|$$
$$|\mathcal{B}_{\mathsf{level_K}}| \leq \mathsf{K}$$

Proof. ($2 \to 1$) is immediate. ($1 \to 2$) is proved on a case analysis over the set of normalized literals of $\mathsf{TSL_K}$. □

4.1 A Combination-Based Decision Procedure for $\mathsf{TSL_K}$

Lemma 2 enables a brute force method to automatically check whether a set of normalized $\mathsf{TSL_K}$-literals is satisfiable. However, such a method is not efficient in practice. We describe now how to obtain a more efficient decision procedure for $\mathsf{TSL_K}$ applying a many-sorted variant [22] of the Nelson-Oppen combination method [12], by combining the decision procedures for the underlying theories. This combination method requires that the theories fulfill some conditions. First, each theory must have a decision procedure. Second, two theories can only share sorts (but not functions or predicates). Third, when two theories are combined, either both theories are stable infinite or one of them is polite with respect to the underlying sorts that it shares with the other. The stable infinite condition for a theory establishes that if a formula has a model then it has a model with infinite cardinality. In our case, some theories are not stable infinite. For example, $T_{\mathsf{level_K}}$ is not stably infinite, T_{ord}, and T_{thid} need not be stable infinite in same instances. The observation that the condition of stable infinity may be cumbersome in the combination of theories for data structures was already made in [16] where they suggest the condition of *politeness*:

Definition 3 (Politeness). *T is polite with respect to sorts $S : \{\sigma_1 \dots \sigma_n\}$ whenever:*

(1) Let φ be a satisfiable formula in theory T, $\mathcal{A}$ be one model of φ and let $|\mathcal{A}_{\sigma_1}|, \dots, |\mathcal{A}_{\sigma_n}|$ be the cardinalities of the domains of $\mathcal{A}$ for sorts in S. For every tuple of larger cardinalities $k_1 \geq |\mathcal{A}_{\sigma_1}|, \dots, k_n \geq |\mathcal{A}_{\sigma_n}|$, there is a model $\mathcal{B}$ of φ with $|\mathcal{B}_{\sigma_i}| = k_i$.

(2) There is a computable function that for every formula φ returns an equivalent formula $(\exists\overline{v})\psi$ (where $\overline{v} = V_\psi \setminus V_\varphi$) such that, if ψ is satisfiable, then there is an interpretation $\mathcal{A}$ with $\mathcal{A}_\sigma = [V_\sigma(\psi)]^{\mathcal{A}}$ for each sort σ.

Condition (*1*) is called *smoothness*, and guarantees that interpretations can be enlarged as needed. Condition (*2*) is called *finite witnessability*, and gives a procedure to produce a model in which every element is represented by a variable. The Finite Model Property, Lemma 2 above, guarantees that every sub-theory of $\mathsf{TSL_K}$ is finite witnessable since one can add as many fresh variables as the bound for the corresponding sort in the lemma. The smoothness property can be shown for:

$$T_{\mathsf{cell}} \oplus T_{\mathsf{mem}} \oplus T_{\mathsf{path}} \oplus T_{\mathsf{set}} \oplus T_{\mathsf{setth}} \oplus T_{\mathsf{mrgn}}$$

with respect to sorts addr, $\mathsf{level_K}$, elem, ord and thid. Moreover, these theories can be combined because all of them are stably infinite. The following can also be combined: $T_{\mathsf{level_K}} \oplus T_{\mathsf{ord}} \oplus T_{\mathsf{thid}}$ because they do not share any sorts, so combination is trivial. The many-sorted Nelson-Oppen method allows to combine the first collection of theories with the second. Regarding the decision procedures for each individual theory, $T_{\mathsf{level_K}}$ is trivial since it is just a finite set of naturals with order. For T_{ord} we can adapt a decision procedure for dense orders as the reals [21], or other appropriate theory. For T_{cell} we can use a decision procedure for recursive data structures [13]. T_{mem} is the theory of arrays [1]. T_{set}, T_{setth} and T_{mrgn} are theories of (finite) sets for which there are many decision procedures [25, 8]. The remaining theories are T_{reach} and T_{bridge}. Following the approaches in [16, 18] we extend a decision procedure for the theory T_{path} of finite sequences of (non-repeated) addresses with the auxiliary functions and predicates shown in Fig. 6, and combine this theory to obtain:

$$T_{\mathsf{SLKBase}} = T_{\mathsf{addr}} \oplus T_{\mathsf{ord}} \oplus T_{\mathsf{thid}} \oplus T_{\mathsf{level_K}} \oplus T_{\mathsf{cell}} \oplus T_{\mathsf{mem}} \oplus T_{\mathsf{path}} \oplus T_{\mathsf{set}} \oplus T_{\mathsf{setth}} \oplus T_{\mathsf{mrgn}}$$

Using T_{path} all symbols in T_{reach} can be easily defined. The theory of finite sequences of addresses is defined by $T_{\mathsf{fseq}} = (\Sigma_{\mathsf{fseq}}, \mathsf{TGen})$, where $\Sigma_{\mathsf{fseq}} = \big(\{\mathsf{addr}, \mathsf{fseq}\}, \{nil : \mathsf{fseq}, cons : \mathsf{addr} \times \mathsf{fseq} \rightarrow \mathsf{fseq}, hd : \mathsf{fseq} \rightarrow \mathsf{addr}, tl : \mathsf{fseq} \rightarrow \mathsf{fseq}\}, \emptyset\big)$ and TGen as the class of term-generated structures that satisfy the axioms of distinctness, uniqueness and generation of sequences using constructors, as well as acyclicity (see, for example [4]). Let Σ_{path} be Σ_{fseq} extended with the symbols of Fig. 6 and let *PATH* be the set of axioms of T_{fseq} including the ones in Fig. 6. Then, we can formally define $T_{\mathsf{path}} = (\Sigma_{\mathsf{path}}, \mathsf{ETGen})$ where ETGen is $\big\{\mathcal{A}^{\Sigma_{\mathsf{path}}} | \mathcal{A}^{\Sigma_{\mathsf{path}}} \models PATH \text{ and } \mathcal{A}^{\Sigma_{\mathsf{fseq}}} \in \mathsf{TGen}\big\}$. Next, we extend T_{SLKBase} with definitions for translating all missing functions and predicates from Σ_{reach} and Σ_{bridge} appearing in normalized $\mathsf{TSL_K}$-literals by definitions from T_{SLKBase}. Let *GAP* be the set of axioms that define ϵ, $[\_]$, *append*, *reach*$_{\mathsf{K}}$, *path2set*, *getp*$_{\mathsf{K}}$, *fstlock* and *ordList*. For instance: $ispath\,(p) \wedge ordPath\,(m,p) \;\leftrightarrow\; ordList\,(m,p)$ We now define $\widehat{\mathsf{TSL_K}} = (\Sigma_{\widehat{\mathsf{TSL_K}}}, \widehat{\mathsf{ETGen}})$ where $\Sigma_{\widehat{\mathsf{TSL_K}}}$ is $\Sigma_{T_{\mathsf{SLKBase}}} \cup \{$ *append*, *reach*$_{\mathsf{K}}$, *path2set*, *getp*$_{\mathsf{K}}$, *fstlock*, *ordList* $\}$ and $\widehat{\mathsf{ETGen}} := \big\{\mathcal{A}^{\Sigma_{\widehat{\mathsf{TSL_K}}}} | \mathcal{A}^{\Sigma_{\widehat{\mathsf{TSL_K}}}} \models GAP \text{ and } \mathcal{A}^{\Sigma_{T_{\mathsf{SLKBase}}}} \in \mathsf{ETGen}\big\}$.

$app : \mathsf{fseq} \times \mathsf{fseq} \rightarrow \mathsf{fseq}$
$app(nil, l) = l \qquad app(cons(a, l), l') = cons(a, app(l, l'))$
$fseq2set : \mathsf{fseq} \rightarrow \mathsf{set}$
$fseq2set(nil) = \emptyset \qquad fseq2set(cons(a, l)) = \{a\} \cup fseq2set(l)$
$ispath : \mathsf{fseq}$
$ispath(nil) \quad ispath(cons(a, nil)) \quad \{a\} \not\subseteq fseq2set(l) \wedge ispath(l) \rightarrow ispath(cons(a, l))$
$last : \mathsf{fseq} \rightarrow \mathsf{addr}$
$last(cons(a, nil)) = a \qquad l \neq nil \rightarrow last(cons(a, l)) = last(l)$
$isreach_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level}_{\mathsf{K}}$
$isreach_{\mathsf{K}}(m, a, a, l) \qquad m[a].next[l] = a' \wedge isreach_{\mathsf{K}}(m, a', b, l) \rightarrow isreach_{\mathsf{K}}(m, a, b, l)$
$isreachp_{\mathsf{K}} : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{level}_{\mathsf{K}} \times \mathsf{fseq}$
$isreachp_{\mathsf{K}}(m, a, a, l, nil)$ $m[a].next[l] = a' \wedge isreachp(m, a', b, l, p) \rightarrow isreachp(m, a, b, l, cons(a, p))$
$fstmark : \mathsf{mem} \times \mathsf{fseq} \times \mathsf{level}_{\mathsf{K}} \times \mathsf{addr}$
$fstmark(m, nil, l, null)$ $p \neq nil \wedge p = cons(a, q) \wedge m[a].lockid[l] \neq \oslash \rightarrow fstmark(m, p, l, a)$ $p \neq nil \wedge p = cons(a, q) \wedge m[a].lockid[l] = \oslash \wedge fstmark(m, q, l, b) \rightarrow fstmark(m, p, l, b)$
$ordPath : \mathsf{mem} \times \mathsf{fseq}$
$ordPath(h, nil)$ $\begin{pmatrix} h[a].next[0] = a' \wedge h[a].key \preceq h[a'].key \wedge \\ p = cons(a, q) \wedge ordPath(h, q) \end{pmatrix} \rightarrow ordPath(h, p)$

Fig. 6. Functions, predicates and axioms of T_{path}

Using the definitions of GAP it is easy to prove that if Γ is a set of normalized $\mathsf{TSL_K}$-literals, then Γ is $\mathsf{TSL_K}$-satisfiable iff Γ is $\widehat{\mathsf{TSL_K}}$-satisfiable. Therefore, $\widehat{\mathsf{TSL_K}}$ can be used in place of $\mathsf{TSL_K}$ for satisfiability checking. The reduction from $\widehat{\mathsf{TSL_K}}$ into T_{SLKBase} is performed in two steps. First, by the finite model theorem (Lemma 2), it is always possible to calculate an upper bound in the number of elements of sort addr, elem, thid, ord and level in a model (if there is one model), based on the input formula. Therefore, one can introduce one variable per element of each of these sorts and unfold all definitions in $PATH$ and GAP, by symbolic expansion, leading to terms in Σ_{fseq}, and thus, in T_{SLKBase}. This way, it is always possible to reduce a $\widehat{\mathsf{TSL_K}}$-satisfiability problem of normalized literals into a T_{SLKBase}-satisfiability problem. Hence, using a decision procedure for T_{SLKBase} we obtain a decision procedure for $\widehat{\mathsf{TSL_K}}$, and thus, for $\mathsf{TSL_K}$. Notice, for instance, that the predicate $subPath : \mathsf{path} \times \mathsf{path}$ for ordered lists can be defined using only $path2set$ as: $subPath(p_1, p_2) \mathrel{\hat{=}} path2set(p_1) \subseteq path2set(p_2)$.

For space reasons, we do not provide complete specification and proofs of the temporal properties. However, in [18] is detailed an example of a termination proof over concurrent lists, which easily carries over to skiplists. For illustration purposes, we now show the full verification condition for the verification of the safety property $\square\big(SkipList_4(h, sl)\big)$ when executing transition 36 of program *insert* by a thread with id t, from Section 2. For clarity, we again use *prev* as a short for $upd^{[t]}[i^{[t]}]$, and we use the auxiliary predicate $setnext(c, d, i, x)$ that makes the cell d identical to c except that $c.next[i] = x$.

$$setnext(c, d, i, x) \hat{=} \begin{pmatrix} d.data = c.data \wedge d.key = c.key \wedge d.lock[j] = c.lock[j] \wedge \\ (i \neq j) \rightarrow d.next[j] = c.next[j] \wedge d.next[i] = x \end{pmatrix}$$

The VC is $(SkipList_4(h, sl) \wedge \varphi \rightarrow SkipList_4(h', sl'))$ where φ is:

$$\begin{pmatrix} x^{[t]}.key = newval & \wedge \\ prev.key < newval & \wedge \\ x^{[t]}.next[i^{[t]}].key > newval & \wedge \\ prev.next[i^{[t]}] = x^{[t]}.next[i^{[t]}] & \wedge \\ (x^{[t]}, i^{[t]}) \notin sl.r \wedge 0 \leq i^{[t]} \leq 3 & \end{pmatrix} \wedge \begin{pmatrix} at_{36}[t] \wedge at'_{37}[t] & \wedge \\ prev'.next[i^{[t]}] = x^{[t]} & \wedge \\ setnext(h[prev], newcell, i^{[t]}, x^{[t]}) & \wedge \\ h' = upd(h, prev, newcell) & \wedge \\ sl = sl' \wedge x'^{[t]} = x^{[t]} & \wedge \end{pmatrix}$$

5 Conclusion and Future Work

In this paper we have presented $\mathsf{TSL_K}$, a theory of skiplists of height at most K, useful for automatically prove the VCs generated during the verification of concurrent skiplist implementations. $\mathsf{TSL_K}$ is capable of reasoning about memory, cells, pointers, masked regions and reachability, enabling ordered lists and sublists, allowing the description of the skiplist property, and the representation of memory modifications introduced by the execution of program statements.

We showed that $\mathsf{TSL_K}$ is decidable by proving its finite model property, and exhibiting the minimal cardinality of a model if one such model exists. Moreover, we showed how to reduce the satisfiability problem of quantifier-free $\mathsf{TSL_K}$ formulas to a combination of theories using the many-sorted version of Nelson-Oppen, allowing the use of well studied decision procedures. The complexity of the decision problem for $\mathsf{TSL_K}$ is easily shown to be NP-complete since it properly extends TLL [16].

Current work includes the translation of formulas from T_{ord}, $T_{\mathsf{level_K}}$, T_{set}, T_{setth} and T_{mrgn} into BAPA [8]. In BAPA, arithmetic, sets and cardinality aids in the definition of skiplists properties. Paths can be represented as finite sequences of addresses. We are studying how to replace the recursive functions from T_{reach} and Σ_{bridge} by canonical set and list abstractions [20], which would lead to a more efficient decision procedure, essentially encoding full $\mathsf{TSL_K}$ formulas into BAPA. The family of theories presented in the paper is limited to skiplists of a fixed maximum height. Typical skiplist implementations fix a maximum number of levels and this can be handled with $\mathsf{TSL_K}$. Inserting more than than 2^{levels} elements into a skiplist may slow-down the search of a skiplist implementation but this issue affects performance and not correctness, which is the goal pursued in this paper. We are studying techniques to describe skiplists of arbitrary many levels. A promising approach consists of equipping the theory with a primitive predicate denoting that the skiplist property holds above and below a given level. Then the reasoning is restricted to the single level being modified. This approach, however, is still work in progress.

Furthermore, we are working on a direct implementation of our decision procedure, as well as its integration into existing solvers. Future work also includes the temporal verification of sequential and concurrent skiplists implementations, including one at the `java.concurrent` standard library. This can be accomplished by the design of verification diagrams that use the decision procedure presented in this paper.

References

1. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Information and Computation 183(2), 140–164 (2003)
2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
3. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: A logic-based framework for reasoning about composite data structures. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 178–195. Springer, Heidelberg (2009)
4. Bradley, A.R., Manna, Z.: The Calculus of Computation. Springer, Heidelberg (2007)
5. Browne, A., Manna, Z., Sipma, H.B.: Generalized verification diagrams. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 484–498. Springer, Heidelberg (1995)
6. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgran-Kaufmann, San Francisco (2008)
7. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)
8. Kuncak, V., Nguyen, H.H., Rinard, M.C.: An algorithm for deciding BAPA: Boolean algebra with presburger arithmetic. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 260–277. Springer, Heidelberg (2005)
9. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: Proc. of POPL 2008, pp. 171–182. ACM, New York (2008)
10. Leino, K.R.M.: Data groups: Specifying the modication of extended state. In: OOPSLA 1998, pp. 144–153. ACM, New York (1998)
11. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer, Heidelberg (1995)
12. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
13. Oppen, D.C.: Reasoning about recursively defined data structures. J. ACM 27(3), 403–411 (1980)
14. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33(6), 668–676 (1990)
15. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 48–64. Springer, Heidelberg (2005)

16. Ranise, S., Zarba, C.G.: A theory of singly-linked lists and its extensible decision procedure. In: Proc. of SEFM 2006. IEEE CS Press, Los Alamitos (2006)
17. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of LICS 2002, pp. 55–74. IEEE CS Press, Los Alamitos (2002)
18. Sánchez, A., Sánchez, C.: Decision procedures for the temporal verification of concurrent lists. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 74–89. Springer, Heidelberg (2010)
19. Sipma, H.B.: Diagram-Based Verification of Discrete, Real-Time and Hybrid Systems. Ph.D. thesis, Stanford University (1999)
20. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: Proc. of POPL 2010, pp. 199–210. ACM, New York (2010)
21. Tarski, A.: A decision method for elementary algebra and geometry. University of California Press, Berkeley (1951)
22. Tinelli, C., Zarba, C.G.: Combining decision procedures for sorted theories. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 641–653. Springer, Heidelberg (2004)
23. Vafeiadis, V.: Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge (2007)
24. Yorsh, G., Rabinovich, A.M., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 94–110. Springer, Heidelberg (2006)
25. Zarba, C.G.: Combining sets with elements. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 762–782. Springer, Heidelberg (2004)