Automated Verification of Concurrent Linked Lists with Counters

Tuba Yavuz-Kahveci and Tevfik Bultan

Department of Computer Science, University of California,
Santa Barbara, CA 93106 USA
{tuba,bultan}@cs.ucsb.edu

Abstract. We present an automated verification technique for verification of concurrent linked lists with integer variables. We show that using our technique one can automatically verify invariants that relate (unbounded) integer variables and heap variables such as $head \neq null \rightarrow numItems > 0$. The presented technique extends our previous work on composite symbolic representations with shape analysis. The main idea is to use different data structures such as BDDs, arithmetic constraints and shape graphs as type specific symbolic representations in automated verification. We show that polyhedra based widening operation can be integrated with summarization operation in shape graphs to conservatively verify properties of concurrent linked lists.

1 Introduction

Manipulation of linked lists is tricky. So is concurrent programming. In this paper we present a technique for automated verification of concurrent linked list specifications. Since we allow unbounded variables (such as integers) verification problem in general is not decidable. Our algorithm computes conservative approximations to the the reachable states of the input system to verify or falsify invariants. When we verify an invariant it is guaranteed to hold, and when we falsify an invariant the invariant is definitely violated. However, we may not be able to verify all the invariants that actually hold, and we may not be able to falsify all the invariants that are actually violated.

We use symbolic forward fixpoint computations in which values of boolean and enumerated variables are encoded using BDDs, values of integer variables are encoded using linear arithmetic constraints, and values of heap variables and configurations of the heap are encoded using shape graphs. Our symbolic manipulator can compute the set of reachable states of an input system (where each state corresponds to valuations of all the variables in the system and the configuration of the heap) given a finite bound on the number of execution steps, i.e., it can compute a lower approximation to the reachable states of the system. This is done by iteratively applying the post-condition operation starting from

This work is supported in part by NSF grant CCR-9970976 and NSF CAREER award CCR-9984822.

the initial states. This lower approximation can be used to search for errors. More interestingly, our verification algorithm can compute an upper approximation to the set of reachable states and demonstrate the absence of errors by showing that the invariants are satisfied by all the states in the upper approximation. We compute an upper approximation to the set of reachable states using two concepts: 1) summarization nodes in shape graphs, and 2) widening operation on polyhedra. Together, these operations force the least fixpoint computation for the set of reachable states to converge to an upper approximation. We demonstrate that the approximate fixpoint computation is precise enough to prove interesting properties on some example concurrent linked list specifications. Although the technique we present for computing a lower approximation to the reachable states generalizes to heap objects with multiple selector fields, the summarization technique we present (hence, the upper approximation of reachable states) is customized for heap objects with a single selector field.

Our work in this paper combines and extends previous work on: 1) *Shape analysis:* Shape analysis algorithm we use in this paper builds on the previous work in this area [SRW98,DRS00,LAS00,LARSW00,Yah01]. However, we extend the shape graph representation by keeping an integer count for each summary node that keeps track of the number of concrete heap nodes represented by the corresponding summary node. 2) *Verification with arithmetic constraint manipulation:* Arithmetic constraints represented as polyhedra have been used in analyzing real-time and concurrent systems [AHH96,BGP99,HRP94]. The techniques we use for integer constraint manipulation are based on [CH78] and [BGP99]. 3) *Model checking with composite representations:* Our earlier work on composite representations concentrated on using BDDs and polyhedral representations together as a symbolic representation to verify infinite state systems with integer variables efficiently [YKTB01]. Here, we extend this work by adding shape graphs as a new symbolic representation.

Our main contribution in this paper is to show a technique that merges shape analysis and verification based on arithmetic constraints under the composite symbolic representation framework. It is not possible to verify the types of systems we are analyzing using the techniques presented in earlier work. The approach used in [LAS00,LARSW00,Yah01] can be extended to handle some predicates on integers but this will require addition of instrumentation predicates and formulas to show how integer predicates change based on single-step execution. Our approach is completely automatic, the key invariants that relate the number of nodes in a linked list to the integer variables in the system are automatically discovered by our verification algorithm. Our tool discovers these invariants using integer counters which keep track of the number of concrete heap nodes represented by summary nodes and using polyhedral widening operation. These invariants in turn imply the relationships between the values of the heap variables and the values of the integer variables. We integrated the shape analysis algorithm presented in this paper to our tools Composite Symbolic Library [YKTB01] and Action Language Verifier [BYK01] using a BDD encoding

```
module main()
  heap {next} head, tail, add, get, newHead;
  boolean mutex; integer numItems;
  initial: head=null and tail=null and numItems=0 and mutex;
  module put()
    enumerated pc {create, setNull, checknull, updateTail};
    initial: pc=create and add=null;
    put1: pc=create and mutex and !mutex' and add'=new and pc'=setNull;
    put2: pc=setNull and add'.next=null and pc'=checknull;
    put3: pc=checknull and numItems=0 and tail'=add and
          head'=add and numItems'=1 and mutex' and pc'=create;
    put4: pc=checknull and numItems>0 and tail'.next=add and pc'=updateTail;
    put5: pc=updateTail and tail'=add and
          numItems'=numItems+1 and mutex' and pc'=create;
    put: put1 | put2 | put3 | put4 | put5;
  endmodule
  module take()
    enumerated pc {copyHeadNext, setNextnull, updateTail};
    initial: pc=copyHeadNext and get=null;
    take1: pc=copyHeadNext and mutex and numItems>0 and !mutex' and
           newHead'=head.next and get'=head and pc'=setNextnull;
    take2: pc=setNextnull and get'.next=null and head'=newHead and
           numItems'=numItems-1 and pc'=updateTail;
    take3: pc=updateTail and newHead=null and tail'=null
           and mutex' and pc'=copyHeadNext;
    take4: pc=updateTail and newHead!=null and mutex' and pc'=copyHeadNext;
    take:  take1 | take2 | take3 | take4;
  endmodule
  main: put() | put() | take() | take() ;
  spec: invariant((head=null and tail=null) <=> numItems=0)
endmodule
```

Fig. 1. Action Language specification of a concurrent linked list

of the shape graphs. We verified several properties of three concurrent linked list specifications.

In Section 2, we present a concurrent linked list example in our input specification language. We discuss the composite symbolic representation in Section 3. In Section 4, we discuss the integration of shape graphs to the composite symbolic representation and in Section 5, we discuss summarization and widening operations. We report our experimental results in Section 6 and give our conclusions in Section 7.

2 A Concurrent Linked List Example in Action Language

As an example system consider the concurrent linked list specification given in Action Language in Figure 1. Action Language is a specification language for modular specification of reactive software systems which allows both asynchronous and synchronous compositions (in this paper we are focusing on verification of asynchronous systems) [BYK01]. Action Language can be thought of as an intermediate language for verification. One can automatically translate a concurrent linked list implementation to an Action Language specification and verify it using our verification tools. Alternatively, one can specify a concurrent linked list in Action Language, verify it, and then automatically generate an implementation from the Action Language specification (which is the approach we took in [YKB02]). In this paper we will concentrate on verification procedures for concurrent linked lists specified in Action Language. These techniques can be extended to verification of concurrent linked lists in programming languages such as Java using one of the approaches above.

An Action Language specification consists of a set of hierarchical module definitions and a set of properties to be verified about the specification. Semantically, each module corresponds to a transition system with a set of states, a set of initial states and a transition relation. In Figure 1, module `main` has two submodules `put` and `take`. Submodule `put` models a process that adds a new item to the linked list and submodule `take` models a process that removes an item from the linked list. Each submodule has a local variable `pc` denoting the program counter. Each module instantiation creates a new instantiation of the local variables of that module. Variables of the `main` module correspond to global variables. The variables `head`, `tail`, `add`, `get`, and `newHead` are heap variables with a selector (i.e., field) called `next`. We are assuming that all selectors of heap objects are pointers to heap (i.e., we are ignoring data members such as integers or booleans). The `main` module has an integer variable `numItems` which keeps track of the number of items in the linked list, and a boolean variable `mutex` which is used as a semaphore.

An initial expression defines the set of initial states of a module. The initial expressions are written as state formulas. We call an expression a *state formula* if it contains no primed variables. In Figure 1, initially, `head` and `tail` are equal to `null`, `numItems` is equal to 0, and the `mutex` is `true`. Initial expressions of submodules `take` and `put` define the initial values of their program counters.

Behavior of a module (i.e., its transition relation) is defined using a module expression. A module expression (which starts with the name of the module) is written by combining its actions and submodules using asynchronous (denoted by `|`) and/or synchronous (denoted by `&`) composition operators. For instance, module `put` is defined in terms of asynchronous composition of its actions `put1`, `put2`, and so on. Module `main` (which defines the transition relation of the overall system) is defined in terms of asynchronous composition of two instantiations of its submodule `put` and two instantiations of its submodule `take`. This models the scenario where there are two processes that add new items to the list and two processes that remove items from the list.

Each action in Action Language defines a single execution step. The actions are specified as transition formulas. We call an expression a *transition formula* if it contains one or more primed variables. In a transition formula, primed variables denote the next-state values for the variables and unprimed variables denote the current-state values. For instance, action `put3` in module `put` states that when `pc=checkNull` and `numItems=0` , then in the next state, `tail` and `head` will be set to `add`, `numItems` will be set to 1 and `mutex` will be set to true and `pc` will be set to `create`.

Asynchronous composition of two actions is defined as the disjunction of their transition relations. However, we also assume that an action preserves the values of the variables which are not modified by itself.

The final line in the definition of the `main` module indicates that we wish the Action Language Verifier to check if the state formula `(head=null and tail=null) <=> numItems=0` is an invariant of this system.

The expressions in Action Language are written as *composite formulas*. A composite formula consists of boolean, integer, and heap formulas connected by logical connectives [YKB02]. Integer formulas consist of linear arithmetic constraints on integer variables and constants connected by logical connectives. Similarly, boolean and heap formulas only refer to boolean and heap variables and constants, respectively.

As mentioned above, a composite formula is a state formula if it contains no primed variables and it is a transition formula if it contains one or more primed variables. The initial condition and the invariants are specified using state formulas and the actions are specified as transition formulas. We will put the following restrictions on the transition formulas:

- A primed heap variable can only appear once in a transition formula in an equality.
- If a heap variable appears primed in a transition formula it can only appear unprimed once in the same transition formula.
- A transition formula is in the form of a conjunction of a heap formula, a boolean formula and an integer formula.
- The heap subformula of a transition formula is a conjunction of a guard which is a heap state formula and an update formula. The update formula is a conjunction of equalities.

3 Verification with Composite Symbolic Representation

A composite symbolic representation maps each variable in the input specification to a *basic symbolic representation*. The sets of system states and transitions are represented as a disjunction of conjunctions of these type-specific, basic symbolic representations. Hence, given a composite formula A its *composite representation* is in the form

$$A \equiv \bigvee_{i=1}^{n} \bigwedge_{t \in T} a_{it}$$

where a_{it} denotes the basic symbolic representation of type t in the ith disjunct, and n and T denote the number of disjuncts and the set of basic symbolic representations, respectively. In the verification algorithms we will present in this

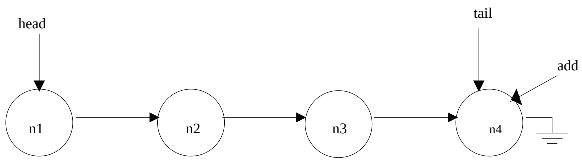

Fig. 2. A linked list with four nodes.

paper $T = \{boolean, integer, heap\}$. We call each disjunct $\wedge_{t \in T} a_{it}$ a *composite atom*.

Each atomic event in the input specification is conjunctively partitioned where each conjunct specifies the effect of the event on the variables represented by a basic symbolic representation. The post-condition computations are computed independently for each symbolic representation by exploiting the conjunctive partitioning of the atomic events.

In this paper we add a new basic symbolic representation for heap variables to the Composite Symbolic Library [YKTB01]. We use sets of shape graphs as a symbolic representation to encode configurations of the heap. The states of boolean and enumerated variables are represented as boolean logic formulas, encoded as BDDs using the Colorado University Decision Diagram Package (CUDD) [CUD]. The states of integer variables are represented as Presburger arithmetic formulas encoded as sets of polyhedra using the Omega Library [Ome]. Basic set operations intersection, union, complement, and subset, equality, and emptiness checks on the disjunctive composite representation can be implemented using corresponding operations on basic symbolic representations as discussed in [YKTB01].

One can use post-condition computation to compute the set of reachable states of a system. The (exact) reachable state-space of a system is the least fixpoint $RS \equiv \mu x \,.\, I \ \vee \ \textsc{post}(x, R)$, where R is the transition relation of the system. We can compute this fixpoint by starting the iteration from the set of initial states and iteratively applying the post-condition computation. However, in an infinite state system convergence is not guaranteed. Even though the exact fixpoint computations are not guaranteed to converge, they may converge to the least fixpoint for some systems. If we cannot compute the set of reachable states RS exactly we can try to compute approximations to it RS^+ or RS^-, such that, $RS^- \subseteq RS \subseteq RS^+$. To verify an invariant we can compute RS^+ and check if all the states in RS^+ satisfy the invariant. To falsify an invariant we can compute RS^- and check if there is a state in RS^- which violates the invariant.

4 Integrating Shape Analysis to Composite Symbolic Representation

In this section we will explain the integration of the heap representation to the composite symbolic representation. We represent the set of states of the heap

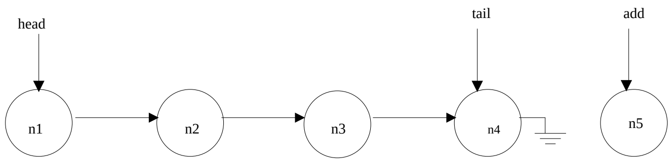

Fig. 3. A linked list with four nodes.

Table 1. List of the heap formulas that can appear in *guard* and *update* parts of a heap transition formula (id_1 and id_2 are heap variables and f is a selector field)

Guard Formulas		Update Formulas	
$id_1 = id_2$	$id_1.f = id_2$	$id'_1 = id_2$	$id'_1 = id_2.f$
$id_1.f = id_2.f$	$id_1 \neq id_2$	$id'_1.f = id_2$	$id'_1.f = id_2.f$
$id_1.f \neq id_2$	$id_1.f \neq id_2.f$	$id'_1 = null$	$id'_1.f = null$
$id_1 = null$	$id_1.f = null$	$id'_1 = new$	$id'_1.f = new$

variables and the heap using a list of *shape graphs*. A shape graph sg is a tuple (N, SR, PT) where

- N is the set of nodes in the shape graph where each node corresponds to a heap cell,
- $SR : Sel \to \mathcal{P}(N \times (N \cup \{null\}))$ is a function that maps each selector field $f \in Sel$ to a set of tuples R_f such that $\forall (m_1, m_2) \in R_f$ s.t. $m_1 \in N$ and $m_2 \in N \cup \{null\}$, $m_1.f = m_2$, and
- $PT : Var \to N \cup \{null, undef\}$ is a function that maps each heap variable either to a heap cell, to $null$, or to $undef$ which means uninitialized.

For example, let $Var = \{head, tail, add\}$, and, let $Sel = \{next\}$, then, the state of the heap for the linked list shown in Figure 2 is represented by the following shape graph:

$$N = \{n1, n2, n3, n4\},\ PT(head) = n1,\ PT(tail) = n4,\ PT(add) = n4,$$
$$SR(next) = \{(n1, n2), (n2, n3), (n3, n4), (n4, null)\}$$

We include a node in a shape graph only if it is reachable from a heap variable, i.e., $\forall n \in N, \exists (n_1, n_2, ..., n_k), n_k = n \wedge (\exists v \in Var, PT(v) = n_1) \wedge \forall 1 \leq i < k, (n_i, n_{i+1}) \in SR$.

A heap transition formula, r, can be formulated as $r = r_{guard} \wedge r_{update}$ where r_{guard} has only current state variables and r_{update} has both current state and next state variables. For example, for the formula $a = b.next \wedge a'.next = b$, r_{guard} part is the formula $a = b.next$ and the r_{update} part is the formula $a'.next = b$. Table 1 shows the possible statements that may appear in r_{guard} and r_{update} parts of a heap transition formula r, respectively.

Given a set of shape graphs s and a heap transition formula r we compute the post-condition of s with respect to r as follows:

$$\textsc{post}(s, r) = \bigvee_{sg \in s} \textsc{post}(sg, r)$$

where $\text{POST}(sg,r)$ updates the representation of the heap state (that is $sg.N$, $sg.SR$, and $sg.PT$) according to the assignment part r_{update} of transition relation r if the heap state sg satisfies the enabling condition r_{guard}.

For instance, given the shape graph shown in Figure 2 and the heap transition formula $r \equiv add' = new$, the post-condition algorithm will output the shape graph given in Figure 3. Note that the *new* keyword generates a new node in the heap.

4.1 Encoding Shape Graphs with BDDs

We encode shape graphs symbolically using BDDs. We represent the nodes in the shape graph with minterms on a set of boolean variables (conjunctions of boolean variables or their negations). For example, the nodes of shape graph in Fig. 2 can be represented using two boolean variables b_1, b_2 where the minterm $\neg b_1 \wedge \neg b_2$ encodes node $n1$, $b_1 \wedge \neg b_2$ encodes node $n2$, $\neg b_1 \wedge b_2$ encodes node $n3$, $b_1 \wedge b_2$ encodes node $n4$. Then, function PT corresponds to a function from heap variables to boolean logic formulas on boolean variables b_1 and b_2. For example, $PT(head)$ is the boolean logic formula $\neg b_1 \wedge \neg b_2$ for the shape graph in Fig. 2 based on the encoding given above.

Each selector defines a binary relation on N, the set of nodes in the shape graph. To encode binary relations defined by the selectors, we duplicate the boolean variables used to encode the nodes in the heap. For example for the above example, we add two new boolean variables b'_1 and b'_2. Now, the binary relation on N defined by the selector *next* can be represented as a boolean logic formula on variables b_1, b_2, b'_1, and b'_2. The formula that corresponds to the shape graph in Fig. 2 based on the encodings of the nodes given above is:

$$\neg b_1 \wedge \neg b_2 \wedge b'_1 \wedge \neg b'_2 \;\vee\; b_1 \wedge \neg b_2 \wedge \neg b'_1 \wedge b'_2 \;\vee\; \neg b_1 \wedge b_2 \wedge b'_1 \wedge b'_2$$

This type of encoding is essentially the same encoding used in BDD based model checking to encode transition relations. Note that *null* corresponds to the boolean value $false$. Boolean value *true* on the other hand is used to encode the configurations where the selector for a node is not initialized.

To keep the size of the BDD encoding small we introduce boolean variables only when they are needed. In our encoding, a shape graph with the set of nodes N will be encoded with $2 \times \log_2(|N|)$ boolean variables. Since new nodes can be added to the heap using the *new* keyword, we introduce new boolean variables to the encoding dynamically if necessary.

4.2 Truncated Fixpoints Computations

Using composite symbolic representation, with shape graphs as a symbolic representation for the heap variables, we can compute the reachable states of a system from the initial set of states up to a certain execution step. Each iteration of the least fixpoint computation for the set of reachable states will give a lower bound for the reachable states RS. Hence, if we truncate the fixpoint computation after a finite number of iterations we will have a lower bound $RS^- \subseteq RS$ for the reachable states. Our verification tool has a flag which can be set to determine the bound on number of fixpoint iterations. If the obtained result is not precise enough to find an error, the precision can be improved by running more fixpoint iterations. This way we can detect errors up to a certain number of execution steps. However, this kind of bounded verification is not sound, i.e., if there are no states violating the invariants in RS^- this does not mean that there are no errors in the system.

5 Approximations: Summarization and Widening

The heap state representation that we described in Section 4 is a precise representation, i.e. each node in N corresponds to a concrete heap cell. One implication of using a precise representation is that the shape graphs may grow arbitrarily large. To guarantee the termination of the fixpoint computation for the set of reachable states we need to introduce some abstraction. We use the notion of *summary* nodes [CWZ90,SRW98,DRS00] to map more than one concrete heap cell into an abstract node in the shape graph. In this section, we will focus on singly linked lists, i.e., we assume that the selector field set Sel has only one element f.

We define an abstraction function $M : N \rightarrow \hat{N}$ that maps the nodes in a concrete shape graph $sg = (N, SR, PT)$ to the nodes of an abstract shape graph $\hat{sg} = (\hat{N}, \hat{SR}, \hat{PT}, SM)$. Since we assume that there is a single selector field, we encode the selector relation as $SR \subseteq N \times (N \cup \{null\})$. We define the set of summary nodes SM as the abstract nodes which represent more than one concrete node:

$$SM = \{\hat{n} \mid \hat{n} \in \hat{N} \wedge \exists n_1, n_2 \in N, n_1 \neq n_2 \wedge M(n_1) = \hat{n} \wedge M(n_2) = \hat{n}\}$$

We impose the following constraints on function M:

1. $\forall \hat{n_i} \in SM,\ \forall n,\ M(n) = \hat{n_i}, \forall v \in Var, PT(v) \neq n,$
2. $\forall \hat{n_i} \in SM,\ \forall n,\ M(n) = \hat{n_i}, \neg(\exists s, t,\ (s, n) \in SR \wedge (t, n) \in SR, s \neq t),$
3. $\forall \hat{n_i} \in SM,\ \forall m,\ M(m) = \hat{n_i},\ \exists n,\ M(n) = \hat{n_i} \wedge ((m, n) \in SR \vee (n, m) \in SR)$

Constraint (1) states that no heap variable points to any of the concrete nodes that are mapped to a summary node. Constraint (2) states that there cannot be more than two heap pointers to any of the concrete nodes that are mapped to a summary node. Constraint (3) states that all the concrete nodes mapped to the same summary node are connected. Note that, based on these constraints, each summary node represents a chain of nodes such that no heap variable points to any of the nodes in the chain and no node in the chain is pointed by more than one heap pointer.

For instance, the set of abstract nodes, $\hat{N}$, of the abstract shape graph for the linked list in Figure 2 is $\{\hat{n_1}, \hat{n_2}, \hat{n_3}\}$ where $M(n_1) = \hat{n_1}$, $M(n_2) = \hat{n_2}$,

$M(n_3) = \hat{n_2}$, and $M(n_4) = \hat{n_3}$. The set of summary nodes for this abstract shape graph is $SM = \{\hat{n_2}\}$.

The abstract selector relation $\hat{SR}$ is defined as follows:

$$\begin{aligned}\hat{SR}(f) = \{(\hat{n_1}, \hat{n_2}) \mid\ & \hat{n_1} \in \hat{N} \wedge \hat{n_2} \in \hat{N} \cup \{null\} \ \wedge \\ & \exists n_1, n_2,\ M(n_1) = \hat{n_1} \wedge M(n_2) = \hat{n_2} \wedge \\ & (M(n_1) \neq M(n_2) \vee M(n_1) \notin SM \vee M(n_2) \notin SM) \wedge (n_1, n_2) \in SR\ \}\end{aligned}$$

For instance, for the abstract shape graph that corresponds to the linked list in Figure 2, $\hat{SR} = \{(\hat{n_1}, \hat{n_2}), (\hat{n_2}, \hat{n_3}), (\hat{n_3}, null)\}$. Note that, the internal edges of the chain represented by a summary node is not represented in $\hat{SR}$.

$\hat{PT}$ is defined as follows:

$$\hat{PT}(v) = \begin{cases} undef & PT(v) = undef \\ null & PT(v) = null \\ \hat{n} & M(m) = \hat{n} \wedge PT(v) = m \end{cases}$$

For instance, for the abstract shape graph that corresponds to the linked list given in Figure 2, $\hat{PT}(head) = \hat{n_1}, \hat{PT}(tail) = \hat{PT}(add) = \hat{n_3}$.

Abstraction of a concrete shape graph is an abstract shape graph with minimum number of nodes which does not violate any of the conditions discussed above. Note that, this abstraction causes a loss of precision by mapping chains of arbitrary lengths to summary nodes. Concretization of an abstract shape graph would generate an infinite set of concrete shape graphs where each summary node is mapped to an infinite number of chains. Based on the abstraction defined above, the set of all possible abstract shape graphs for singly linked lists with a given number of heap variables is finite [YKB02]. (Note that, we are only representing nodes reachable from heap variables.) Since the set of all possible abstract shape graphs are finite, we can guarantee termination for the fixpoint computations using the abstract shape graphs. However, we lose the precision that existed in the concrete representation. Below, we propose a technique that: 1) guarantees termination and 2) yields a more precise representation than the abstract shape graphs that only use summary nodes.

We achieve the above features by associating each summary node s_i in an abstract shape graph sg with a summary count $count_{s_i}$ that keeps the number of concrete nodes that s_i represents. As we discussed earlier a composite symbolic representation is in the form $S \equiv \bigvee_{i=1}^{n} s_{i_{bool}} \wedge s_{i_{int}} \wedge s_{i_{heap}}$ where each composite atom $s_{i_{bool}} \wedge s_{i_{int}} \wedge s_{i_{heap}}$ consists of a boolean logic formula $s_{i_{bool}}$ (encoded as a BDD), a Presburger arithmetic formula $s_{i_{int}}$ (encoded as a set of polyhedra), and a set of shape graphs $s_{i_{heap}}$. We encode the valuations for the summary count variables for the shape graphs in $s_{i_{heap}}$ in the Presburger arithmetic formula $s_{i_{int}}$.

Initially, when a summary node is created, its summary count is assigned to a value equal to the number of heap cells it summarizes. Whenever we update a shape graph during the post-condition computation based on an update formula given in Table 1 we need to update the summary counts accordingly. We achieve this by generating an integer formula which specifies how the summary counts

SUMMARIZE(shape graph $sg = (\hat{N}, \hat{SR}, \hat{PT}, SM)$)
  Remove all the nodes that are not reachable from a heap variable
  Initialize *constant* and *coef* arrays to 0, *marked* array to false and *removed* to $\emptyset$
  for each heap variable v **do** $marked[\hat{PT}(v)] \leftarrow true$;
  for each node n **do**
    if $\exists s, t, s \neq t \wedge (s, n) \in \hat{SR} \wedge (t, n) \in \hat{SR}$ **then**
      $marked[n] \leftarrow true$;
  for each heap variable v such that $\hat{PT}(v) \notin \{undef, null\}$ **do**
    $s \leftarrow \hat{PT}(v)$;
    while $\exists (s, t) \in \hat{SR} \wedge t \neq null$ **do**
      if $marked[s] \vee marked[t]$ **then**
        $s \leftarrow t$;
      else
        if $s \notin SM \wedge t \in SM$ **then**
          $\hat{N} \leftarrow \hat{N} \setminus \{s\}$;
          $\hat{SR} \leftarrow \hat{SR} \setminus (\{(s, t)\} \cup \{(m, s) | (m, s) \in \hat{SR}\}) \cup \{(m, t) | (m, s) \in \hat{SR}\}$;
          $constant[t] \leftarrow constant[t] + 1$;
          $s \leftarrow t$;
        else
          $\hat{N} \leftarrow \hat{N} \setminus \{t\}$;
          $SM \leftarrow SM \cup \{s\}$;
          $\hat{SR} \leftarrow \hat{SR} \setminus (\{(s, t)\} \cup \{(t, m) | (t, m) \in \hat{SR}\}) \cup \{(s, m) | (t, m) \in \hat{SR}\}$;
          if $s \in SM \wedge t \in SM$ **then**
            $SM \leftarrow SM \setminus \{t\}$;
            $coef[s, t] \leftarrow 1$;
            $removed \leftarrow removed \cup \{t\}$;
          else if $s \in SM \wedge t \notin SM$ **then**
            $constant[s] \leftarrow constant[s] + 1$;
          else // $s \notin SM \wedge t \notin SM$
            $constant[s] \leftarrow 2$;
  $sg.F \leftarrow \bigwedge_{t \in removed} count'_t = 0 \wedge$
    $\bigwedge_{s \in SM} count'_s = count_s + \sum_{t \in removed} coef[s, t] \times count_t + constant[s]$;

Fig. 4. Summarization algorithm

should be updated for each generated shape graph during the post-condition computation.

Figure 4 shows the summarization algorithm we use. SUMMARIZE algorithm is called whenever an update to a shape graph may create a chance for summarization. The algorithm first does the garbage collection; it removes the nodes that are not reachable by any heap variable. Then it merges nodes and replaces them with summary nodes respecting the constraints (1), (2), and (3) imposed on M. While updating the summary nodes, SUMMARIZE algorithm also constructs an arithmetic constraint ($sg.F$) which reflects the effect of the updates on the summary counts.

SETNULL algorithm is used to implement the effect of update formulas of type $x' = null$ on shape graphs. Given a shape graph sg and a heap variable x,

POST(composite atom $s \equiv s_{int} \wedge s_{bool} \wedge s_{heap}$, composite transition formula $r \equiv r_{int} \wedge r_{bool} \wedge r_{heap}$): composite formula
  $result$: composite formula
  $heapf$: list of shape graphs
  $intf$: integer formula
  $boolf$: boolean formula
  $result \leftarrow$ false;
  $boolf \leftarrow$ POST(s_{bool},r_{bool});
  $heapf \leftarrow$ POST(s_{heap},r_{heap});
  for each shape graph h_{sg} in $heapf$ **do**
    $intf \leftarrow$ POST($s_{int}, r_{int} \wedge h_{sg}.F$);
    $result \leftarrow result \vee (intf \wedge boolf \wedge h_{sg})$
  return $result$;

Fig. 5. An algorithm for computing the post-condition of a composite atom state formula s with respect to a composite atom transition formula r

SETNULL algorithm first sets the heap variable x to null. Since setting x to null may create a chance for summarization it calls SUMMARIZE algorithm:

SETNULL(shape graph sg, heap variable x)
  $sg.\hat{PT}(x) \leftarrow null$;
  SUMMARIZE(sg);

Given a shape graph sg and an update formula $x' = y.f$, SETTONEXT algorithm (Figure 6) makes heap variable x point to the node that is pointed by $y.f$. It first calls SETNULL algorithm to set heap variable x to null. Summarization is handled inside SETNULL algorithm. Then SETTONEXT algorithm performs a materialization operation if m, where $y.f = m$, is a summary node. In this case two different shape graphs, sg_1 and sg_2, are generated. Both sg_1 and sg_2 are initialized with a copy of sg. sg_1 corresponds to the case that m abstracts exactly two concrete nodes ($count_m = 2$). In sg_1, m is removed ($count'_m = 0$) and it is replaced by two nodes, v and w, each representing a concrete node. At the end sg_1 satisfies $x = y.f = v \wedge v.f = w$. sg_2 corresponds to the case that m abstracts more than 2 concrete nodes ($count_m > 2$). In sg_2, m is kept as a summary node which abstracts one less concrete nodes ($count'_m = count_m - 1$). Only one new node, v, that represents a concrete node is introduced in sg_2. At the end sg_2 satisfies $x = y.f = v \wedge v.f = m$. On the other hand, if m is not a summary node then the algorithm simply sets x to $y.f$ and the resulting shape graph satisfies $x = y.f = m$. The existential quantification and renaming used in combining $sg_1.F$ and $sg_2.F$ with $sg.F$ is required to compose the constraints generated by the summarization and the materialization operations. Since the arithmetic constraint manipulator used in the Composite Symbolic Library can handle Presburger arithmetic we are able to use quantification in the generated formulas.

We compute the post-condition for shape graphs using algorithms SETNULL, SETTONEXT, and similar algorithms for other update formulas as follows:

```
POST(shape graph sg, heap transition formula r): set of shape graphs
    switch (r)
        case (x' = null):
            SETNULL(sg,x);
            return {sg};
        case (x' = y.f):
            return SETTONEXT(sg,x,y);
        ...
```

Using the post condition computation given above we implement the post condition computation for a composite atom as shown in Figure 5. Then, the post condition computation of the overall transition relation is computed as:

```
POST(composite representation S ≡ ⋁_i s_i, composite transition formula R ≡
⋁_j r_j): composite representation
    result: composite representation
    result ← false;
    for each composite atom s_i in S do
        for each disjunct r_j in R do
            result ← result ∨ POST(s_i,r_j);
    return result;
```

where the composite transition formula is stored in a disjunctive form (similar to the composite representation of states).

Note that, the summary counts for summary nodes can increase without a bound. Actually, our input language also allows unbounded integer variables. To force the fixpoint computation for reachable states to converge we use the *widening* technique [CC77]. Note that, the shape graphs without summary counts, and the boolean and enumerated variables are bounded. Hence, the only cause for nontermination is in the integer symbolic representation. For the integer symbolic representation we use the widening operator defined in [BGP99] for Presburger arithmetic constraints by generalizing the convex polyhedra widening operator in [CH78]. The basic idea is to find pairs of polyhedra p and q such that $p \subseteq q$ and set $p \bigtriangledown q$ to conjunction of constraints in p which are also satisfied by q. Intuitively, if a constraint of p is not satisfied by q this means that the iterates are increasing in that direction. By removing that constraint we extend the iterates in the direction of growth as much as possible without violating other constraints.

6 Experimental Results

We experimented our technique by verifying the safety properties of three different concurrent data structure specifications: single lock queue, stack and two lock queue [Yah01]. Table 3 shows the verified safety properties for each data structure specification. All of the examples use an integer variable to keep the number of items stored in the data structure. We verified each instance with

SETTONEXT(shape graph sg, heap variable x, heap variable y): set of shape graphs
$sg \leftarrow$ SETNULL(sg, x);
assert $(\exists(n,m) \in sg.\hat{SR},\ sg.\hat{PT}(y) = n)$;
if $m \in sg.SM$ **then** // materialization operation
$sg_1 \leftarrow getCopy(sg)$;
$sg_1.\hat{N} \leftarrow sg.\hat{N} \setminus \{m\} \cup \{v, w\}$;
$sg_1.SM \leftarrow sg.SM \setminus \{m\}$;
$sg_1.F \leftarrow count_m = 2 \wedge count'_m = 0$;
$sg_1.F \leftarrow \exists count''_m(sg.F[count'_m \mapsto count''_m] \wedge sg_1.F[count_m \mapsto count''_m])$;
$sg_1.\hat{SR} \leftarrow sg.\hat{SR} \setminus (\{(t,m)|(t,m) \in sg.\hat{SR}\} \cup \{(m,t)|(m,t) \in sg.\hat{SR}\})$
$\cup\{(t,v)|(t,m) \in sg.\hat{SR}\} \cup \{(w,t)|(m,t) \in sg.\hat{SR}\} \cup \{(v,w\})$;
$sg_1.\hat{PT}(x) \leftarrow v$;

$sg_2 \leftarrow getCopy(sg)$;
$sg_2.\hat{N} \leftarrow sg_2.\hat{N} \cup \{v\}$;
$sg_2.F \leftarrow count_m > 2 \wedge count'_m = count_m - 1$;
$sg_2.F \leftarrow \exists count''_m(sg.F[count'_m \mapsto count''_m] \wedge sg_2.F[count_m \mapsto count''_m])$;
$sg_2.\hat{SR} \leftarrow sg.\hat{SR} \setminus \{(t,m)|(t,m) \in sg.\hat{SR}\} \cup \{(t,v)|(t,m) \in sg.\hat{SR}\} \cup \{(v,m)\}$;
$sg_2.\hat{PT}(x) \leftarrow v$;
return $\{sg_1, sg_2\}$;
else // $m \notin sg.SM$
$sg_1 \leftarrow getCopy(sg)$;
$sg_1.\hat{PT}(x) \leftarrow m$;
return $\{sg_1\}$;

Fig. 6. An algorithm for computing the post-condition of a heap state formula represented by a shape graph sg with respect to update formula $x' = y.f$

varying number of producer and consumer process types. Table 2 shows the verification timing results (in seconds) for these examples. $PProd$ ($PCons$) means that the specification is verified for arbitrary number of producer (consumer) processes. We used counting abstraction [Del00] to verify parameterized cases.

Results show that the verification time is scalable with the increasing number of processes. Most of the time the verification of the specifications for arbitrary number of producer or consumer processes are faster than that of the cases where there are fixed number of processes. This is possibly due to the fact that counting abstraction, in a way, implements a symmetry reduction by abstracting away the information about individual processes. The results also show that the verification time for the case where the emptiness check is expressed using an integer formula is greater than the verification time for the case where the emptiness check is expressed using a heap formula.

7 Conclusions

The approach presented in this paper combines previous work on shape analysis, infinite state verification using arithmetic constraints and symbolic model

Table 2. Verification timing results (in seconds) for the concurrent single lock queue, stack, and two lock queue for the safety properties given in Table 3. CT and VT denote construction time and verification time in seconds, respectively. HC and IC refer to two different specifications for each problem in which the control flow is based only on heap formulas (using tests such as $head \neq null$) or both heap and integer formulas (using tests such as $numItems > 0$), respectively. $\uparrow$ denotes that the verifier ran out of memory.

Number of Processes	Queue				Stack				TwoLockQueue			
	HC		IC		HC		IC		HC		IC	
	CT	VT	CT	VT	CT	VT	CT	VT	CT	VT	CT	VT
1Prod-1Cons	0.01	10.19	0.01	12.95	0.01	4.57	0.02	5.21	0.01	60.5	0.04	58.13
2Prod-2Cons	0.03	15.74	0.09	21.64	0.02	6.73	0.04	8.24	0.69	88.26	0.59	122.47
4Prod-4Cons	9.07	31.55	19.38	46.5	5.56	12.71	4.98	15.11	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
1Prod-PCons	0.01	12.85	0.02	13.62	0.01	5.61	0.02	5.73	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
PProd-1Cons	0.01	18.24	0.02	19.43	0.01	6.48	0.01	6.82	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$

Table 3. Safety properties of the concurrent data structure specifications.

Specification	Verified Invariants
Single Lock Queue	$(head = null) \Leftrightarrow numItems = 0$
	$(head \neq null) \Leftrightarrow numItems > 0$
	$(head = tail \wedge head \neq null) \Leftrightarrow numItems = 1$
	$(head \neq tail) \Leftrightarrow numItems \geq 2$
Stack	$numItems = 0 \Leftrightarrow top = null$
	$numItems > 0 \Leftrightarrow top \neq null$
	$numItems = 2 \Rightarrow top.next \neq null$
Two Lock Queue	$numItems > 1 \Rightarrow head \neq tail$
	$numItems > 2 \Rightarrow head.next \neq tail$

checking to obtain a powerful verification technique. Using Composite Symbolic Library [YKTB01] we were able to integrate shape analysis to the composite representation framework. We think that we will be able to generalize the approach presented here to verification of liveness properties and backward fixpoint computations in the future. Extending the presented approach to heap objects with multiple selector fields (i.e., doubly linked lists) is another future research direction.

References

[AHH96] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

[BYK01] T. Bultan and T. Yavuz-Kahveci. Action Language Verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming*, pages 84–97, 1978.

[CUD] CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`

[CWZ90] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *ACM SIGPLAN Conference on Program Language Design and Implementation*, 1990.

[Del00] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68, 2000.

[DRS00] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In Jens Palsberg, editor, *7th International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer, 200.

[HRP94] N. Halbwachs, P. Raymond, and Y. Proy. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *Proceedings of International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1994.

[LARSW00] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing And Analysis*, 2000.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analysis. In Jens Palsberg, editor, *7th International Static Analysis Symposium*, Lecture Notes in Computer Science. Springer, 200.

[Ome] The Omega project. `http://www.cs.umd.edu/projects/omega/`

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[Yah01] E. Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *ACM Symposium on Principles of Programming Languages*, 2001.

[YKB02] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. Technical Report 2002-19, University of California, Santa Barbara, 2002.

[YKTB01] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representation. In *Proceedings of the Seventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, 2001.