Formal Verification of Skiplist Algorithms

Cong Quy Trinh
Abstract

Formal Verification of Skiplist Algorithms

Cong Quy Trinh

Dynamic memory allocation and dynamic structures are very important data structures. Many algorithms including both sequential and concurrent algorithms have been introduced to access to and manipulate these structures. However, inferring and verifying invariants of these algorithms are challenging tasks. In this thesis, we consider the problem of automatically verifying correctness of concurrent algorithms with an unbounded number of threads that access a shared heap. Such algorithms are used, e.g., to implement data structures in common concurrency libraries. Here, we consider algorithms in which each heap node has several pointer field, in particular algorithms that manipulate skiplist-like data structures. For such algorithms, we propose a methodology for generating and checking program invariants. Particular difficulties are incurred by the necessity to use universal quantification over heap nodes. We present a suitable form for such invariants, together with techniques for inferring them and checking their validity. We manually apply the technique to a concurrent skiplist algorithm from the literature. We believe that the techniques, possibly slightly modified, will provide a suitable basis for automated implementation.
Acknowledgements

Firstly, I would like to say many thanks to my supervisor Prof. Bengt Jonsson. Thank you so much for giving me the opportunity to do my thesis with you, for your course Verification Methods which led me to the world of verification, and for teaching me how to write and present an academic article. Without your helps I would not be able to finish this thesis.

Secondly, my sincere thanks go to my reviewer Prof. Parosh Abdulla for approving my thesis and awarding me a thesis "gift". Many thanks to my friends and colleagues in UPMARC group, especially Dr. Ahmed Rezine for helping and discussing with me when I was stuck in my thesis.

Thirdly, I wish to express my appreciations to my family for their love, support and encouragement during my master study in Uppsala.

In the end, I am grateful to my close friends, thanks Thanh Truong for walking with me to the university when my bike got broken, thanks Paul Subotic for discussing with me every day about thesis works via Facebook, thanks all of my friends in Uppsala.
List of Figures

2.1 A skiplist example .............................................. 6
4.1 Program to build a simple skiplist .............................. 15
Chapter 1

Introduction

A difficult challenge in software verification is to automate its application to algorithms with an unbounded number of threads that concurrently access a dynamically allocated shared state. Such algorithms are used, e.g., to implement data structures that can be accessed concurrently by a large number of threads. Such implementations are provided by concurrency libraries, e.g., the Intel Threading Building Blocks or the `java.util.concurrent` package. They typically use fine-grained locking or specialized atomic operations for synchronization, and are therefore notoriously hard to get correct, as witnessed by many bugs found in published algorithms (e.g., [3, 8]). A further complication in verification is that algorithms typically use dynamically heap-allocated data, which itself is a challenge for verification. When verifying such algorithms, one must typically generate assertions about the structure of the heap at different points of program execution.

We address the problem of automatically verifying implementations of concurrent data structures based on the skiplist data structure (e.g., [6]). Such data structures pose a hard challenge to automated verification, since they employ several pointer relations, whose interrelation must be discovered during verification. In particular, one must discover, express, and reason about relations between arbitrary heap nodes. Several approaches to this problem, collectively termed shape analyses, have been developed. However, to the best of our knowledge, there has not been any previous work that can automatically verify correctness of skiplist implementations.

In this thesis, we propose an approach for reasoning about concurrent heap-manipulating algorithms with several pointer relations. The approach is general, but its development has been driven by the desire to cope with skiplist algorithms. The starting point for our shape analysis is the transitive closure logic by Bingham and Rakamarić [2]. This approach appears suitable for automation, since there is a bounded number of assertions, and that there are powerful techniques for computing postconditions: indeed it has been used for automated verification of many sequential list-manipulation algorithms. It turns out that we need to extend the logic with quantification over heap nodes, in order to cover important properties of the heap. For example, an invariant for a sorted linked list is that ‘for any two nodes $a$ and $b$, if $a$ is reachable from $b$ then the value of $a$ is larger than value of $b$’. Since automated reasoning with quantifiers is in general rather complex, we have defined a limited form for quantified invariants, which is sufficient for the examples we have considered, and also appears to be tractable for automated reasoning.

In the thesis, we will describe a typical algorithm that we have considered for verification [6], and the form of assertions that we use to represent properties of the program state. We thereafter describe first how we generate candidate assertions, and thereafter how the validity of such assertions are checked by reasoning about the effects of individual program statements. For lack of time, we
have not yet implemented our approach. In principle, we conjecture that our approach is suitable for automation, but it is possible that a straight-forward implementation would have scalability problems. However, our expectation is that these can be overcome, possibly with some effort. In any case, the thesis provides sufficient insights into the techniques that are necessary and sufficient for automatically analyzing algorithms based on skiplists and similar data structures.

In summary, the main contributions of the thesis are (1) a simple technique to generate and check quantified assertions over dynamically allocated heaps, and (2) understanding of which techniques should be used for automatically verifying algorithms based on skiplists and similar data structures.

Related Work: One of the most well-known shape analysis tools is the Three Valued Logic Analyzer (TVLA) [7] which uses abstract interpretation with a three value logic abstract domain to represent reachable states of a system. The abstract semantics of program statements are defined by abstract translators generated by TVLA. However, the abstract invariant is not always precise therefore improving the precision of such domains is an ongoing research challenge.

The paper [5] shows a method for automatic inferring quantified invariants of programs manipulating array and pointer structures. A primary component of its method is an automatic technique to transform quantifier-free abstract domains into universally quantified domains by using over-approximation and under-approximation techniques. Then, based on the obtained quantified domains, it infers quantified invariants of the programs. However, its quantified domain construction technique only works well with array domains, the technique is not suitable for constructing quantified pointer domains because pointer over-approximation operations are hard to compute and give large tradeoffs between original and approximated domains.

Another approach to verification of programs that manipulate linked lists is proposed in [2] in which they use predicate abstraction to abstract concrete systems by using transitive closure logic to assert program states. Properties of the abstract systems are verified throughout decision procedures which can be called several time during verification processes. However, in this paper, the properties are given by verifiers and its method can not verify quantified properties.
Chapter 2

Background

2.1 Skiplists

A skiplist is a list consisting of a number of sorted linked lists, each of which is located at a layer. Each skiplist node has a key value, one successor at each layer in which it participates, and is assigned a height. Figure 1 shows a skiplist with integer keys where the node of key value 5 has height 2 and one next pointer to the next node of key 10. There is an important property of skiplist stating that the list at any higher layer is a sublist of the list at the layer beneath it. A skiplist has head and tail nodes with the maximum heights at the head and tail positions respectively. The head’s key value is assigned $-\infty$, and the tail’s key value is $+\infty$. Initially, when the skiplist is empty, the head is the successor of the tail at all layers. Figure 2.1 shows a skiplist with maximum height of 2, the number below each node is the key of that node, with $-\infty$ and $+\infty$ as the keys for the head and tail nodes respectively.

![Figure 2.1: A skiplist example](image)

2.2 Program Annotation

In this thesis, to verify program invariants we verify them at assertions at all program points. Each program point assertion is a conjunction of atomic predicates which are valid at that program point. These predicates are taken from abstract domains. An abstract domain is represented by a finite set of atomic formulas. Each atomic predicate describes relations ranging over program variables. By choosing a suitable abstract domain, we can verify invariants of infinite programs.

In order to compute assertions at program points, we need a method to compute the assertion at successor of any program point which has already been evaluated. In this method, if $A$ is a program point then to compute the assertion at its successor we check validity of every given atomic predicate $P$ from a given abstract domain. If $P$ is valid at the successor of $A$ then it is added to the desired
assertion. Each checking time involves a call to a decision procedure \[2\] to determine if the following formula is unsatisfiable:

\[
post(A) \land \neg P
\]

Here \(post(A)\) is the strongest postcondition of \(A\) which computes the strongest set of predicate conjunctions that characterize possible program states after the program has executed the statement from the program assertion at \(A\). \(Post(A)\) computing is described more detail in \[2\]. Since program annotation is an incomplete approach, when it fails to verify the invariants, either because the concrete program system actually violates the invariants or the loss of information in the assertions. Choosing right abstract domains is not an easy task. In our current approach, we manually choose them by counterexample behaviors; discovering more precise abstraction domains is an important part of our future work.

2.3 Thread Modular Analysis

Recall that, the goal of our analysis is to compute a finite set of concurrent program assertions. Hence, to get a smaller domain of assertions, we employ thread-modular technique which correlates only the local variables of one thread with the global variables. Furthermore, our analysis is thread-modular in the sense that for each program point, it infers assertions that correlate global variables with an arbitrary copy of the local variables. In more detail, assertions are computed by two types of program steps:

**Local Steps:** For program steps that are performed by the thread which is considered in the assertions, assertions are computed as described in the Section 2.2.

**Interference Steps:** For program steps that are performed by other threads than the current thread in which we are trying to compute the assertions, it must be checked that they preserve the assertions. The computation of the assertions is slightly more involved, we use local variables of both current and interfering threads, therefore the assertions at each program point in the current thread are computed as the coordinations of global effects made by the interfering threads and the local assertion of the current thread at that point.

2.4 Overview

In this section, we give a brief overview of our technique for verifying concurrent algorithms with several pointer relations. The approach first generates and thereafter checks a potential invariant of the algorithm. We must first define a form for invariants, which can be tractably generated, and still express the necessary properties of program states. After some experimentation, and with inspiration from \[5\], we decided to use invariants which for each program point consists of an assertion of form

\[
E \land \forall u, v \bigwedge_{i=0}^{n} (F_i(u, v) \Rightarrow e_i(u, v)) .
\]

Here \(E\), called the environment is a conjunction of atomic predicates over local and global program variables; each \(F_i(u, v)\) is conjunction of atomic predicates over (local and global) program variables, and the quantified variables \(u\) and \(v\), which range over heap nodes, and each \(e_i(u, v)\) is an atomic
predicate over $u$ and $v$, which is fixed by the user. Typical examples of consequents $e_i(u, v)$ include stating that $u \rightarrow^m v$ (node $v$ is reachable from node $u$ via $\rightarrow$-pointers), or $u < v$ (the key at node $u$ is less than the key at node $v$). The atomic predicates are binary predicates using relations including $\rightarrow$ (direct successor under the $\rightarrow$ pointer relation), $\rightarrow^m$ (transitive closure of $\rightarrow$), $<$ (comparison of keys), $=$ (aliasing), and their negations. An example of a quantified assertion appearing in our skiplist example is

$$\forall u, v \{ u \rightarrow^m v \land u \neq v \Rightarrow u < v \},$$

which intuitively states that the skiplist is sorted.

It should be noted that assertions may contain variables that are local to a thread. In that case, the assertion at a program point $p$ has the interpretation that whenever any thread $t$ is at program point $p$, then the assertion holds, using the values of the local variables of thread $t$. In this way, the assertion has an implicit quantification over program threads, which must be kept in mind when checking generated assertions.

In our methodology, we first infer candidate assertions for each program point by generalizing from a set of program states obtained from a sample of generated program executions. We thereafter check the assertions by proving that they are inductive, i.e., that the validity of the generated assertions is preserved by all program statements. We describe each of these steps in more detail below.

**Inferring Assertions** Our methodology for inferring assertions at each program point consists in collecting a set of reachable program states obtained from a sample generated execution. We could for instance start with an empty list, and insert and delete a small number of arbitrary elements. For each program point, we examine the program states that were observed at that program point, and generate a candidate assertion as follows.

- The environment $E$ is generated as the set of all atomic predicates over program variables that are valid at all states of this program point. It is easy to see that this generates the strongest conjunction that is valid for all the observed states.

- For each user-defined consequent $e_i(u, v)$, we generate the corresponding antecedent $F_i(u, v)$ as a smallest conjunction of atomic predicates over program variables, $u$, and $v$, under which the implication $F_i(u, v) \Rightarrow e_i(u, v)$ is valid in all observed states. If there are several such smallest conjunctions, several conjuncts are generated in the assertion.

**Checking Assertions** To check that the generated assertions indeed are valid for all program executions, we check that they are preserved by all program statements. We must then compute and reason about postconditions of assertions in the form described above. For this, we use the transitive closure logic by Bingham and Rakamarić [2], for reasoning about relationships between heap cells. We extend it to quantification.

To make the analysis cover an unbounded number of threads, it is made thread-modular in the sense of [1]. This means that for each assertion, we must compute its postcondition for two types of program steps.

**Local Steps:** For program steps that are performed by the thread which is considered in the assertion, postconditions are computed as normal.

**Interference Steps:** For program steps that are performed by other threads than the one which is considered in the assertion, it must be checked that they preserve the assertion. The computation
of the postcondition is slightly more involved, since now local variables of two threads are involved, and one should also assume the assertion of the interfering thread.

In both these cases, the transitive closure logic of [2] is ultimately used for reasoning.
Chapter 3

Assertion Inferring Method

To infer assertions at program points, we start with an empty list and iterate over the loop $n$ number of times, where we insert new nodes and delete existing nodes randomly. For each loop iteration, we find environments which are described later at all program points. After loop has been terminated, the final environment at each program point is computed as intersection of environments in all interactions at that point, and then the assertions are inferred based on these common environments.

3.1 Assertion Template

The invariant at each program point consists of an assertion of form

$$E \land \forall u, v \bigwedge_{i=0}^{n} (F_i(u, v) \Rightarrow e_i(u, v)).$$

Here $E$, called the environment is a conjunction of atomic predicates over local and global program variables; each $F_i(u, v)$ is conjunction of atomic predicates over (local and global) program variables, and the quantified variables $u$ and $v$, which range over heap nodes, and each $e_i(u, v)$ is an atomic predicate over $u$ and $v$, which is fixed by the user.

3.2 Inferring Assertion

The process of finding assertions with a fixed right-hand side predicate $e_i(u, v)$, assertions are discovered by following steps

- Firstly, for each loop iteration, the environments at program points are computed. Secondly, after the loop is terminated, for each program point we infer the common environment by taking intersection of environments at that point in all interactions.

- Let $S$ be the set of all possible atomic predicates describing relations between $u$, $v$ and global variables of the algorithms. The relations here include reachability ($x \overset{i}{\rightarrow} y$) and inequality ($x \neq y, d(x) \neq d(y), d(x) \neq v$), non-reachability ($x \overset{i}{\nrightarrow} y$) and equality ($x = y, d(x) = d(y), d(x) = v$). Here, $x \overset{i}{\rightarrow} y$ and $x \overset{i}{\nrightarrow} y$ mean that the node $y$ is reachable from $x$ and $y$ is not reachable from $x$ respectively, $x \neq y$ means that the node $x$ is different from $y$ and $d(x) = d(y)$,
$d(x) = v$ mean that data value of $x$ is different from data value of $y$ and the value $v$ respectively.
Whereas, $x = y$, $d(x) = d(y)$, $d(x) = v$ mean that $x$, $y$ are the same nodes and data value of $x$ is equivalent to data value of $y$ and the value $v$.

From the set $S$ and each program point $j$, find all minimal possible conjunctions $F_i(u,v)$ of these predicates such that $E_j \land F_i(u,v) \Rightarrow e(u,v)$ where $E_j$ is the environment at program point $j$ obtained from the previous step. In order to check whether the formula $E_j \land F_i(u,v) \Rightarrow e(u,v)$ is always true for any assignments of $u$, $v$ or not, we simply check the consistency of the formula $E_j \land F_i(u,v) \land \neg e(u,v)$, if this formula is not consistent for all possible instances of $u$, $v$ then the formula $E \land F_i(u,v) \Rightarrow e(u,v)$ is always true, otherwise $E \land F_i(u,v) \Rightarrow e(u,v)$ is not true for all instances of $u$, $v$.

In our method, we infer assertions based on a small number $n$ of loop iterations. Hence, the accuracy of the method depends on how large $n$ is. Its not guaranteed to ensure that all strongest assertions (minimal conjunction of atomic predicates in the left hand side of quantified parts) found by the method are actual strongest assertions because some of them are properly invalid at iteration number $m$ where $m > n$. However all actual strongest assertions are guaranteed to be found by our method.

**Lemma 1.1** (soundness of the method). For a list $L$, if $P$ is an actual strongest assertions of $L$ then it is guaranteed to be generated by the inferring assertions method.

*Proof:* Assume that $P$ is the actual strongest assertion of list $L$ so it is valid at program points of all loop iterations from 1 to $n$, because our method considers all possible assertions $R$ that are valid at all program points from 1 to $n$ and they are also strongest. Therefore we have $P \subseteq R$ $\square$
Chapter 4

Assertion Verifying

In this chapter, in order to check that the generated assertions are valid for all program executions, we check that they are preserved by all program statements. Because the assertions which we are verifying are inductive, therefore we firstly assume that assertions at initial program points are valid then check for any program transition, if we assume that the assertion at the pre program point is valid then it also holds at the post program point. To do this, the first step is to find a method to compute the assertion at the post program point (post assertion) after statement $S$ from the assertion at the pre program point (pre assertion). Then, we check that the environment of post assertion is implied by postcondition of pre environment, and the quantified part of post assertion implies the pre assertion’s quantified part.

4.1 Post Assertion

Let us firstly show how to compute the post assertion from the pre program point where the invariant that we aim to verify is assumed to be valid. Recall that the assertion at the pre program point is formed as

$$\Psi = E \land \forall u,v \bigwedge_{i=0}^{n} (F_i(u,v) \Rightarrow e_i(u,v))$$  \hspace{1cm} (4.1)

Here $E$ is the environment at the pre program point. Note that, $E$ here not only asserts the pre program point but also satisfies the quantified part $(I)$. After executing statement $S$ we have post assertion called $\Psi'$

$$\Psi' = E' \land \text{post}(\forall u,v \bigwedge_{i=0}^{n} (F_i(u,v) \Rightarrow e_i(u,v)), S)$$  \hspace{1cm} (4.2)

In $\Psi'$ assertion, $E'$ is the post environment of $E$ and it is computed by using the technique described in the background chapter.

Since it is difficult to compute postcondition of the quantified part directly hence we compute its over-approximation. According to the Theorem 2.2, $\Psi'$ is approximated as below

$$\Psi' = E' \land \forall u,v \{\text{post} \bigwedge_{i=0}^{n} (F_i(u,v) \Rightarrow e_i(u,v)), S)\}$$  \hspace{1cm} (4.3)
Because postcondition commutes over conjunction eg. \( \text{post}(A_1 \land A_2, S) \Rightarrow \text{post}(A_1, S) \land \text{post}(A_2, S) \)
we one more time over-approximate \( \Psi' \)
\[
\Psi' = E' \land \forall u, v \{ \big( \bigwedge_{i=0}^{n} \text{post}(F_i(u, v) \Rightarrow e_i(u, v)), S) \} \tag{4.4}
\]

Break implication in \( \Psi' \) we get the new form for \( \Psi' \) as below
\[
\Psi' = E' \land \forall u, v \{ \big( \bigwedge_{i=0}^{n} \text{post}(\bigvee_{j=0}^{m} p_{ij}), S) \} \tag{4.5}
\]

Where \( p_{ij} \) is either a negation of an atomic predicate in \( F_i(u, v) \) or \( e_i(u, v) \). Since postcondition distributes over disjunction, computing \( I' \) is reducible to computing \( \text{post}(p_{ij}, S) \)
\[
\Psi' = E' \land \forall u, v \{ \big( \bigwedge_{i=0}^{n} \bigvee_{j=0}^{m} \text{post}(p_{ij}, S) \} \tag{4.6}
\]

Computing \( \text{post}(p_{ij}, S) \) is quite simple. At first, quantified variables \( u, v \) are instantiated as ghost variables. Ghost variables are assignable variables that appear in program assertions but do not correspond to physical entities so they do not influence on the lists. Then we compute \( \text{post}(p_{ij}, S) \) by using the saturation technique in [2].

In the next subsection 4.2 we show how to check that the assertion at the pre program point is preserved at the post program point when we execute the statement \( S \).

**Theorem 2.2.** For any statement \( S \) and quantified formula \( \forall x, y \phi(x, y) \) with bound variables \( x, y \) we have \( \text{post}(\forall x, y \phi(x, y)) \models \forall x, y \{ \text{post}(\phi(x, y)) \}

**Proof:** Let us check with \( x := x_1, y := y_1 \), for these substitutions \( \text{post}(\forall x, y \phi(x, y)) \models \forall x, y \{ \text{post}(\phi(x, y)) \} \tag{1} \)

becomes \( \text{post}(\phi(x_1, y_1)) \models \phi(x_1, y_1) \) that is always true. Now by induction way, firstly (1) is assumed to be true with \( (x, y) = (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) \) then we need to prove (1) is true with \( (x, y) = (x_{n+1}, y_{n+1}) \). From the assumption we have \( \forall x, y \{ \text{post}(\phi(x, y)) \} \)

with \( (x, y) = (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) \) that means that \( \text{post}(\phi(x_1, y_1)) \land \phi(x_2, y_2) \land ... \land \phi(x_n, y_n) \) \models \( \text{post}(\phi(x_{n+1}, y_{n+1})) \). Applying the theorem 2.1 with \( n = 2 \) and \( A_1 = \phi(x_1, y_1) \land \phi(x_2, y_2) \land ... \land \phi(x_n, y_n) \) and \( A_2 = \phi(x_{n+1}, y_{n+1}) \) we get \( \text{post}(\phi(x_1, y_1) \land \phi(x_2, y_2) \land ... \land \phi(x_n, y_n) \land \phi(x_{n+1}, y_{n+1})) \models \text{post}(\phi(x_{n+1}, y_{n+1})) \) \tag{2}.

From (2) and assumption \( \text{post}(\phi(x_1, y_1) \land \phi(x_2, y_2) \land ... \land \phi(x_{n+1}, y_{n+1})) \models \text{post}(\phi(x_1, y_1)) \land \text{post}(\phi(x_2, y_2)) \land ... \land \text{post}(\phi(x_{n+1}, y_{n+1})) \) \( \square \)

### 4.2 Assertion Verifying

In general, transitive closure first order logic with two variables that we are dealing with is undecidable [4] hence assertion verifying can not be done by complete approaches. Recall that the pre assertion is formed as
\[
\Psi = E \land \forall u, v \{ F_i(u, v) \Rightarrow e_i(u, v) \} \tag{4.7}
\]
To check that such assertion is preserved at the post program point, as already mentioned we check two conditions as below:

Firstly, for the quantifiers free part we check that

\[ \text{post}(E, S) \Rightarrow E' \]  \hspace{1cm} (4.8)

Secondly, for the quantified part we check the validity of the formula

\[ E' \land \forall u, v (\bigwedge_{i=0}^{n} (\bigvee_{j=0}^{m} \text{post}(p_{ij}, S))) \Rightarrow \forall u, v (\bigwedge_{i=0}^{n} (F_i(u, v) \Rightarrow e_i(u, v))) \]  \hspace{1cm} (4.9)

In order to check the validity of first-order formulas of the form (4.9). Firstly, the universal quantifiers at the right hand side of (4.9) are removed using skolemization to get a new formula.

\[ E' \land \forall u, v \left( \bigwedge_{i=0}^{n} \left( \bigvee_{j=0}^{m} \text{post}(p_{ij}, S) \right) \right) \Rightarrow \bigwedge_{i=0}^{n} (F_i(u_1, v_1) \Rightarrow e_i(u_1, v_1)) \]  \hspace{1cm} (4.10)

where \( u_1, v_1 \) are fresh variables.

Secondly, we instantiate quantified variables by replacing \( u, v \) with all concrete terms \( x_i, y_j \) in \( E' \) and \( u_1, v_1 \) such that \( \Phi(u/x_i, v/y_j) \) is not true in \( E' \). Assume that \( u/x_1, v/y_1, u/x_2, v/y_2, \ldots, u/x_n, v/y_n \) are all these possible substitutions, now to check (4.10) we check unsatisfiability of the formula.

\[ E' \land \Phi(x_1, y_1) \land \Phi(x_2, y_2) \land \ldots \land \Phi(x_n, y_n) \land \Phi(u_1, v_1) \land \bigwedge_{i=0}^{n} (F_i(u_1, v_1) \land \neg e_i(u_1, v_1)) \]  \hspace{1cm} (4.11)

The formula (4.11) is checked for unsatisfiability by calling a saturation decision procedure \( \Psi \). If it is unsatisfiable then we say that the formula (4.9) is valid.

**Lemma 2.1** (soundness of assertion verifying). If there exists \( (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n), (u_1, v_1) \) such that \( E \land \Phi(x_1, y_1) \land \Phi(x_2, y_2) \land \ldots \land \Phi(x_n, y_n) \land \Phi(u_1, v_1) \land \neg P(u_1, v_1) \) is unsatisfiable then \( E \land \forall u, v \Phi(u, v) \Rightarrow \forall u, v P(u, v) \)

**Proof:** From \( E \land \Phi(x_1, y_1) \land \Phi(x_2, y_2) \land \ldots \land \Phi(x_n, y_n) \land \Phi(u_1, v_1) \land \neg P(u_1, v_1) \) is unsatisfiable so we have \( E \land \Phi(x_1, y_1) \land \Phi(x_2, y_2) \land \ldots \land \Phi(x_n, y_n) \land \Phi(u_1, v_1) \Rightarrow \forall u, v P(u, v) \). It’s easy to see that \( \forall u, v \Psi(u, v) \) implies \( \Phi(x_1, y_1) \land \Phi(x_2, y_2) \land \ldots \land \Phi(x_n, y_n) \land \Phi(u_1, v_1) \) because of universal quantifier property hence \( E \land \forall u, v \Phi(u, v) \Rightarrow \forall u, v P(u, v) \)

As can be seen from our method, its inefficiency increases exponentially depending on the number of bound and ground variables. However, in our case we verify assertions with only two bound variables and the number of program variables in algorithms are limited. Therefore our method is sufficient and efficient enough to verify assertions of skiplist algorithms as well as other simple list algorithms which use few program variables
procedure Build(Node* head, Node* tail)
For i = 2 to i = 3

{ t := i
addNode(head, tail, t)
}
end procedure

bool addNode(Node* head, Node* tail, int t){
int found = findNode(t, sPred, sSucc, mPred, mSucc)
if (found <> -1)
return false
else
Node* n = new Node(t, 2);
n.mNext = mSucc;
mPred.mNext = n;
n.sNext = sSucc;
sPred.sNext = n;
return true;
}

int findNode ( int t, Node* sPred, Node* sSucc,
Node* mPred, Node* mSucc){
int found = -1;
Node* pred = head;
Node* curr = pred.sNext;
while ( t > curr ) {
pred = curr ; curr = pred.sNext;
}
if ( t == curr )
found = 2;
}

sPred = pred; sSucc = curr;
Node* curr = pred.mNext;
while ( t > curr ) {
pred = curr ; curr = pred.mNext;
}
if (found == -1 & t == curr ){
found = 1;
}
mpred = pred; msucc = curr;
return found ;
}

Figure 4.1: Program to build a simple skiplist

4.3 Illustrative Example
To give an idea of our method, we show how the verification can be carried out on a small simplified program which builds up a skiplist with only two layers, the lower layer $m$ and the higher $s$ from an empty skiplist as in Section 2. In this program, we have removed the mechanisms for concurrency such as locking and marking to make it simple. In short, the procedure build inserts some nodes into a skiplist, for each loop iteration, it calls the function findNode to find the suitable position then insert the node into the skiplist at that position by the function addNode.

Inferring Assertions For each program point, we try to find an assertion of the form

$$E \land \forall u, v \bigwedge_{i=0}^{n} (F_i(u, v) \Rightarrow e_i(u, v)) .$$
In the assertion, we fix $e_i(u,v)$ as either \( u \xrightarrow{m}{v} \) or $u < v$. $F_i(u,v)$ is the smallest conjunction of atomic predicates over program variables, $u$, and $v$. If there are several such $F_i(u,v)$, several conjuncts are generated in the assertion. $E$ is conjunction of atomic predicates that are valid in the program point. Each atomic predicate is of the form

$$ x \circ y \text{ or } \neg(x \circ y) \text{ or } local^i(x), $$

where $\circ$ can be either $\rightarrow$, $\leftarrow$, $<$, or $=$ and $i$ can be either $m$ or $s$, whereas $x, y$ can be any different local or global variables. Here, $local^i(x)$ means the node $x$ cannot be reached from any global variable by $i$ pointer of any thread.

Let us first illustrate how our method works. Here we run two iterations of a loop to insert two new nodes into the empty list. In the example, we examine the point $A_{11}$.

After inserting the first node, the following atomic predicates are valid:

$$ head \xrightarrow{m}{tail} \quad head \xrightarrow{s}{tail} \quad n \xrightarrow{m}{mSucc} \quad mPred \xrightarrow{m}{n} \quad n \xrightarrow{s}{sSucc} \quad sPred \xrightarrow{s}{n} \quad mSucc = tail \quad mPred < n \quad sSucc = tail \quad head = mPred \quad head = sPred \quad n < mSucc \quad sPred < n \quad n < sSucc \quad mPred = sPred $$

Here, predicates which are trivially implied by other predicates are not shown in our example if these other predicates are valid at the same point. For example, the predicate $head \xrightarrow{s}{sPred}$ is not shown at the point $A_{11}$ because at that point the predicate $head = sPred$ is valid.

After inserting the second node, the only difference from the first iteration is that $head \xrightarrow{m}{mPred}$ and $head \xrightarrow{s}{sPred}$ are valid in the second iteration, but in the first iteration, $head = mPred$ and $head = sPred$ are valid.

By taking the intersection of the predicates in the two iterations, we get the common environment:

$$ \phi_{A_{11}} \equiv \quad head \xrightarrow{m}{tail} \land head \xrightarrow{s}{tail} \land mPred \xrightarrow{m}{n} \land n \xrightarrow{m}{mSucc} \land n \xrightarrow{s}{sSucc} \land sPred \xrightarrow{s}{n} \land mSucc = tail \land sSucc = tail \land head = mPred \land head = sPred \land n < mSucc \land n < sSucc \land mPred = sPred $$

Next, let us find the minimal conjunction $F_1(u,v)$ and $F_2(u,v)$ of atomic predicates in program point $A_{11}$ which is asserted as

$$ \phi_{A_{11}} \land \forall u,v( [F_1(u,v) \Rightarrow u \xrightarrow{m}{v}] \land [F_2(u,v) \Rightarrow u < v] ) $$

Indeed, one minimal $F_1(u,v)$ can be $u \xrightarrow{s}{v}$. For example, if $(u, v)$ is substituted by $(head, n)$ then we have

$$ P \equiv \phi_{A_{11}} \land head \xrightarrow{s}{n} \Rightarrow head \xrightarrow{m}{n} $$

$P$ is obviously true in this case because $mPred \xrightarrow{m}{n}$ and $head \xrightarrow{m}{mPred}$ are in $\phi_{A_{11}}$, therefore $head \xrightarrow{m}{n}$ is also valid according to transitive property of reachability relations [2]. It is easy to see
that $P$ is also true for all other assignments to $u, v$. In a similar way, $F_2(u, v)$ is found as $u \overset{s}{\rightarrow}_m v \land u \neq v$

Therefore, the quantified part of the final assertion at $A_{11}$ is

$$\forall u, v \left[ u \overset{s}{\rightarrow}_m v \land u \neq v \implies u < v \right] \land \left[ u \overset{s}{\rightarrow}_m v \implies u \overset{s}{\rightarrow}_m v \right]$$

**Checking Assertions** Let us analyze the two important program points $A_{10}$ and $A_{11}$ in the first iteration of the previous part to see how the assertion at $A_{10}$ is preserved at $A_{11}$. To do this, we first show that the environment $\phi_{A_{11}}$ at $A_{11}$ is implied by the postcondition of $\phi_{A_{10}}$ after statement $S$ which is $sPred_sNext := n$. Thereafter, the quantified part $\forall u, v \Psi_{A_{10}}$ at $A_{10}$ is implied by $\phi_{A_{11}} \land \forall u, v \text{ post}(\Psi_{A_{10}}, S)$.

Let us first show why $\phi_{A_{11}}$ is implied by $\text{post}(\phi_{A_{10}}, S)$. Assume that the invariant holds in $A_{10}$, we assert $A_{11}$ as below

$$\phi_{A_{10}} \land \forall u, v \Psi_{A_{10}}(u, v).$$

In which:

$$\Psi_{A_{10}}(u, v) \equiv \left[ u \overset{m}{\rightarrow}_v \land u \neq v \implies u < v \right] \land \left[ u \overset{s}{\rightarrow}_m v \implies u \overset{s}{\rightarrow}_m v \right].$$

$$\phi_{A_{10}} \equiv \text{head} \overset{s}{\rightarrow}_m \text{tail} \land \text{head} \overset{s}{\rightarrow}_s \text{head} = m\text{Pred} \land m\text{Pred} \rightarrow n \land m\text{Pred} \land m\text{Succ} = \text{tail} \land m\text{Succ} = \text{tail} \land n \land m\text{Pred} < n \land m\text{Pred} = \text{tail} \land m\text{Succ} = \text{tail} \land n < m\text{Succ} \land m\text{Pred} = s\text{Pred}$$

As can be seen, the difference between $\phi_{A_{11}}$ (already computed in the previous part) and $\phi_{A_{10}}$ is that, the new node $n$ is totally inserted in $A_{11}$ so the atomic predicate $local^h(n)$ is not in $\phi_{A_{11}}$, whereas this predicate is in $\phi_{A_{10}}$ of program point $A_{10}$.

The postcondition of $\phi_{A_{10}}$ is computed by using saturation rules described in [2] as below

$$\text{post}(\phi_{A_{10}}, S) \equiv \text{head} \overset{s}{\rightarrow}_m \text{tail} \land \text{head} = m\text{Pred} \land m\text{Pred} \rightarrow n \land m\text{Pred} \land m\text{Succ} = \text{tail} \land m\text{Succ} = \text{tail} \land n < m\text{Succ} \land m\text{Pred} \land m\text{Pred} = s\text{Pred}$$

The difference between $\phi_{A_{11}}$ and $\text{post}(\phi_{A_{10}}, S)$ is that we can not find the predicate $\text{head} \overset{s}{\rightarrow}_s \text{tail}$ which is in $\phi_{A_{11}}$ in $\text{post}(\phi_{A_{10}}, S)$. However, this predicate is implied by the conjunction

$$\text{head} = s\text{Pred} \land s\text{Pred} \rightarrow n \land m\text{Succ} = s\text{Succ}$$

which is already in $\text{post}(\phi_{A_{10}}, S)$. Therefore we can say that $\phi_{A_{11}}$ is implied by $\text{post}(\phi_{A_{10}}, S)$ (a).

Secondly, we compute postcondition of $\Psi_{A_{10}}(u, v)$ after statement $S$. Firstly, each implication in $\Psi_{A_{10}}(u, v)$

$$\Psi_{A_{10}}(u, v) \equiv [u \overset{s}{\rightarrow}_m v \land u \neq v \implies u < v] \land [u \overset{s}{\rightarrow}_m v \iff u \overset{s}{\rightarrow}_m v]$$

is translated to disjunctive normal form

$$\Psi_{A_{10}}(u, v) \equiv [u \overset{s}{\rightarrow}_m v \land u = v \lor u < v] \land [u \overset{s}{\rightarrow}_m v \lor u \overset{s}{\rightarrow}_m v]$$
By considering $u, v$ as ghost variables which can be any variables in the program and apply saturation rules in [2], we obtain $\text{post}(\Psi_{A10}(u, v), S)$ as below

$$\text{post}(\Psi_{A10}(u, v), S) \equiv [u \xrightarrow{s} v \lor (u \xrightarrow{s} \text{sPred} \land n \xrightarrow{s} v) \lor u \xrightarrow{m} v] \land [u \xrightarrow{s} v \lor u = v \lor u < v]$$

To show that the quantified part of the assertion at $A_{10}$ is preserved at $A_{11}$, we prove the validity of the formula below

$$\phi_{A_{11}} \land \forall u, v \text{ post}(\Psi_{A10}(u, v), S) \Rightarrow \forall u, v \Psi_{A10}(u, v)$$ (4.12)

To prove (1), we prove the unsatisfiability of the below formula

$$\phi_{A_{11}} \land \forall u, v \text{ post}(\Psi_{A10}(u, v), S) \land \exists u, v \neg \Psi_{A10}(u, v)$$ (4.13)

The formula (2) is equivalent to

$$\phi_{A_{11}} \land \forall u, v \text{ post}(\Psi_{A10}(u, v), S) \land \neg \Psi_{A10}(u_0, v_0)$$ (4.14)

Where $u_0, v_0$ are two random variables, the formula (3) is unsatisfiable because when we substitute $(u, v)$ with $(u_0, \text{sPred})$, $(n, v_0)$ and $(u_0, v_0)$ the substituted formula below

$$\text{post}(\Psi_{A10}(u_0, \text{sPred}), S) \land \text{post}(\Psi_{A10}(n, v_0), S) \land \text{post}(\Psi_{A10}(u_0, v_0), S) \land \phi_{A_{11}} \land \neg \Psi_{A10}(u_0, v_0)$$

is unsatisfiable. Hence, the quantified part $\forall u, v \Psi_{A10}$ is preserved at the program point $A_{11}$ (b). From (a) and (b) we can say that the assertion at $A_{10}$ is preserved by the transition $A_{10} \xrightarrow{s} A_{11}$. 

18
Chapter 5

Proof Outlines

This chapter describes the proof outlines of three main concurrent skiplist algorithms taken from the paper [6]. To make our proof easy to read, we simplify these algorithms to the two layered skiplist algorithms but keep the mechanisms for concurrency.

5.1 Skiplist Invariants

By using our assertion inferring method, we infer the quantified parts at all program points as below

$$\forall u, v [u \stackrel{s}{\rightarrow} v \Rightarrow u \stackrel{m}{\rightarrow} v] \land [u \stackrel{m}{\rightarrow} v \land u \neq v \Rightarrow u < v]$$

In words, it means that the list at the lowest layer $m$ is sorted in increasing order of node’s key values. It also says that the $s$ list is a sublist of $m$ list meaning that whenever two node are connected at $s$ layer they are also connected at $m$ layer.

5.2 Skiplist Concurrent Algorithms

5.2.1 Find node

F01: \textbf{int findNode ( int t, Node* sPred, Node* sSucc, Node* mPred, Node* mSucc)}{
F02: \textbf{int lFound = 1;}
F03: \textbf{Node* pred = L ;}
F04: \textbf{Node* curr = pred.sNext;}
F05: \textbf{while ( t > curr )} {
F06: \textbf{pred = curr ; curr = pred.sNext;}
F07: \textbf{}}
F08: \textbf{if ( t == curr )} {
F09: \textbf{lFound = 2;}
F10: \textbf{}}
F11: \textbf{sPred = pred; sSucc = curr;}
F12: \textbf{Node* curr = pred.mNext;}
F13: \textbf{while ( t > curr )} {
F14: \textbf{pred = curr ; curr = pred.mNext;}
F15: \textbf{}}

19
Shorthand point. Before that, we write down some shorthand notations. When we see more than one assertions at a point that means there are some program cases at that point. Before that, we write down some shorthand notations.

Let us show how the annotation works, each program point corresponds to one or several assertions. We cannot show how the annotation works, each program point corresponds to one or several assertions. When we see more than one assertions at a point that means there are some program cases at that point. Before that, we write down some shorthand notations.

**Shorthand:**

<table>
<thead>
<tr>
<th>R</th>
<th>l  \xrightarrow{s} r \wedge l  \xrightarrow{m} r</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_{SP}</td>
<td>l  \xrightarrow{s} sPred</td>
</tr>
<tr>
<td>Ord_1</td>
<td>sPred &lt; t \wedge mPred &lt; t \wedge t &lt; mSucc \wedge t &lt; sSucc</td>
</tr>
<tr>
<td>Ord_2</td>
<td>sPred &lt; t \wedge mPred &lt; t \wedge t \leq mSucc \wedge t = sSucc</td>
</tr>
<tr>
<td>Ord_3</td>
<td>sPred &lt; t \wedge mPred &lt; t \wedge t = mSucc \wedge t &lt; sSucc</td>
</tr>
<tr>
<td>R_{SPS}</td>
<td>sPred  \xrightarrow{s} sSucc</td>
</tr>
<tr>
<td>R_{mPS}</td>
<td>mPred  \xrightarrow{m} mSucc</td>
</tr>
<tr>
<td>mNotReached(x)</td>
<td>x is not reached by any m pointer</td>
</tr>
<tr>
<td>sNotReached(x)</td>
<td>x is not reached by any s pointer</td>
</tr>
</tbody>
</table>

\( \phi_{F01} = R \)
\( \phi_{F05} = R \wedge l\text{Found} = -1 \wedge \text{pred} = l \wedge \text{pred} \rightarrow \text{curr} \)
\( \phi_{F07} = R \wedge l\text{Found} = -1 \wedge l \xrightarrow{s} \text{pred} \wedge \text{pred} \rightarrow \text{curr} \wedge \text{pred} < t \)
\( \phi_{F08} = R \wedge l\text{Found} = -1 \wedge l \xrightarrow{s} \text{pred} \wedge \text{pred} \rightarrow \text{curr} \wedge \text{pred} < t \wedge t \leq \text{curr} \)
\( \phi_{F11} = R \wedge l\text{Found} = 2 \wedge L \xrightarrow{s} \text{pred} \wedge \text{pred} \rightarrow \text{curr} \wedge \text{pred} < t \wedge t = \text{curr} \)
\( \phi_{F11} = R \wedge l\text{Found} = -1 \wedge l \xrightarrow{s} \text{pred} \wedge \text{pred} \rightarrow \text{curr} \wedge \text{pred} < t \wedge t < \text{curr} \)

\( \phi_{F12} = R \wedge l\text{Found} = 2 \wedge l \xrightarrow{s} \text{pred} \wedge t > s\text{Pred} \wedge s\text{Pred} \rightarrow \text{sSucc} \wedge l \xrightarrow{s} \text{sPred} \wedge t = s\text{Succ} \)

\( \phi_{F12} = R \wedge l\text{Found} = -1 \wedge l \xrightarrow{s} \text{pred} \wedge R_{SPS} \wedge R_{SP} \wedge t = s\text{Succ} \wedge t = s\text{Pred} \xrightarrow{m} \text{pred} \wedge \text{sPred} < t \)

\( \phi_{F13} = R \wedge l\text{Found} = -1 \wedge l \xrightarrow{s} \text{pred} \wedge R_{SPS} \wedge R_{SP} \wedge t = s\text{Succ} \wedge \text{pred} \xrightarrow{m} \text{pred} \wedge s\text{Pred} < t \wedge \text{pred} < t \wedge t \leq \text{curr} \)

\( \phi_{F13} = R \wedge l\text{Found} = 2 \wedge l \xrightarrow{s} \text{pred} \wedge R_{SPS} \wedge R_{SP} \wedge t = s\text{Succ} \wedge \text{pred} \xrightarrow{m} \text{pred} \wedge s\text{Pred} < t \wedge \text{pred} < t \wedge t \leq \text{curr} \)

\( \phi_{F16} = R \wedge l\text{Found} = -1 \wedge R_{PS} \wedge R_{SP} \wedge t < s\text{Succ} \wedge \text{pred} \xrightarrow{m} \text{curr} \wedge \text{sPred} \xrightarrow{m} \text{pred} \wedge \text{pred} < t \wedge \text{pred} < t \wedge t \leq \text{curr} \)

\( \phi_{F16} = R \wedge l\text{Found} = 2 \wedge R_{PS} \wedge R_{SP} \wedge t = s\text{Succ} \wedge \text{sPred} \xrightarrow{m} \text{pred} \wedge \text{pred} \rightarrow \text{curr} \wedge s\text{Pred} < t \wedge \text{pred} < t \wedge \text{curr} < t \)
\[ \phi_{F19} = R \land lFound = 1 \land R_{SP} \land R_{SP} \land t < s\text{Succ} \land \text{pred} \rightarrow curr \land s\text{Pred} \rightarrow pred \] 
\[ \land s\text{Pred} < t \land pred < t \land t = curr \]

\[ \phi_{F19} = R \land lFound = -1 \land R_{SP} \land R_{SP} \land t < s\text{Succ} \land \text{pred} \rightarrow curr \land s\text{Pred} \rightarrow pred \] 
\[ \land s\text{Pred} < t \land pred < t \land t < curr \]

\[ \phi_{F19} = R \land lFound = 2 \land R_{SP} \land R_{SP} \land s\text{Pred} \rightarrow m\text{Pred} \land R_{mPS} \land s\text{Pred} < t \land pred < t \land curr < t \land t = s\text{Succ} \]

\[ \phi_{F20} = R \land lFound = 1 \land R_{SP} \land R_{SP} \land R_{mPS} \land s\text{Pred} \rightarrow m\text{Pred} \land l \rightarrow s\text{Pred} \land m\text{Succ} \rightarrow s\text{Succ} \land \text{Ord}_{3} \]

\[ \phi_{F20} = R \land lFound = -1 \land R_{SP} \land R_{SP} \land R_{mPS} \land s\text{Pred} \rightarrow m\text{Pred} \land l \rightarrow s\text{Pred} \land m\text{Succ} \rightarrow s\text{Succ} \land \text{Ord}_{1} \]

\[ \phi_{F20} = R \land lFound = 2 \land R_{SP} \land R_{SP} \land s\text{Pred} \rightarrow m\text{Pred} \land R_{mPS} \land l \rightarrow s\text{Pred} \land m\text{Succ} \rightarrow s\text{Succ} \land \text{Ord}_{2} \]

### 5.2.2 Add node

```cpp
bool add(int t) {
    int topLevel = random(2);
    while (true) {
        int lFound = findNode(t, sPred, sSucc, mPred, mSucc);
        if (lFound == -1) {
            continue;
        } else if (lFound == 1) {
            Node* nodeFound = mSucc;
        } else {
            Node* nodeFound = sSucc;
        }
        if (!nodeFound.marked) {
            while (!nodeFound.fullyLinked())
                return false;
        }
    }
    int lockLevel = -1;
    try {
        if (topLevel == 1) {
            mPred.lock.lock();
            locklevel = 1;
            valid = !mPred->marked \&\& !mSucc->marker \&\& mPred->mNext == mSucc;
            if (!valid) continue;
            Node* n = new Node(t, 1);
            n.mNext = mSucc;
            mPred.mNext = n;
            n.fullylinked = true;
            return true;
        }
    }
    if (topLevel == 2) {
        // Code continues here
    }
}
```
A31: mPred.lock.lock();
A32: lockLevel = 1;
A33: valid = ¬mPred.marked ∧ ¬mSucc.marked ∧ mPred.mNext = mSucc;
A34: if (¬valid) continue;
A35: if (sPred ≠ mPred)
A36: spred.lock.lock();
A37: lockLevel = 2;
A38: }
A39: valid = ¬sPred.marked ∧ ¬sSucc.marked ∧ sPred.sNext = sSucc;
A40: if (¬valid) continue;
A41: Node* n = new Node(t,2);
A42: n.mNext = mSucc;
A43: mPred.mNext = n;
A44: n.sNext = sSucc;
A45: sPred.sNext = n;
A46: n.fullyLinked = true;
A47: return true;
A48: }
A49: finally {
A50: if (locklevel = 1) unlock(mPred)
A51: if (locklevel = 2)
A52: unlock(mPred);
A53: unlock(sPred);
A54: }
A55: }
A56: }
A57: }

Let us annotate the algorithm with important program points to see how the post assertions are computed after program statements and how to check these assertions. These program points include A31, A39 which we can see interference effects easily and A43, A44, A45, A46 which are four points for inserting a new node. In each program point, when the assertion is checked to be valid, we continue to compute and check its post assertion at the post program point until all important program points have been checked. At the initial point, the invariant is assumed to be valid.

Firstly, each implication in the invariant

\[ \Psi = [u \xrightarrow{s} v \Rightarrow u \xrightarrow{m} v] \land [u \xrightarrow{m} v \land u \neq v \Rightarrow u < v] \]

is translated to disjunctive normal form

\[ \Psi = [u \xrightarrow{s} v \lor u \xrightarrow{m} v] \land [u \xrightarrow{s} v \lor u = v \lor u < v] \]

then we have algorithm’s annotation at these interesting program points as below

• Program Point A31

\[ \phi_{A31} = R \land lFound = -1 \land R_{sPS} \land R_{sP} \land R_{mPS} \land Ord_{1} \land \xrightarrow{l}{sPred} \land mSucc \xrightarrow{s}{sSucc} \land \xrightarrow{m}{sPred} \]
• Program Point A39

\[ \phi_{A39} = R \land \neg lFound = -1 \land R_{sPS} \land R_{sP} \land R_{mPS} \land Ord_1 \land l \xrightarrow{s} mPred \land mSucc \xrightarrow{s} sSucc \land sPred \xrightarrow{s} mPred \]

• Program Point A42

\[ \phi_{A42} = R \land \neg lFound = -1 \land R_{sPS} \land R_{sP} \land R_{mPS} \land sPred \xrightarrow{s} mPred \land n \xrightarrow{s} \text{nil} \land n \xrightarrow{s} \text{nil} \]

\[ \land l \xrightarrow{s} mPred \land mSucc \xrightarrow{s} sSucc \land mNotReached(n) \land sNotReached(n) \land Ord_1 \]

For program statement \( S : n.mNext := mSucc \), apply the rule \( x \xrightarrow{s} y \) with \( t_1 = n \) and \( t_2 = mSucc \) for instances \( x = u \) and \( y = v \) we get consequences \( u \xrightarrow{s} n \land mSucc \xrightarrow{s} v \) or \( u \xrightarrow{s} v \) then

\[ x \xrightarrow{s} y \quad \text{mNotReached(y)} \]

from the consequences apply the rule \( x \xrightarrow{s} y = y \) with \( y = n \) for instances \( x = u \) and \( y = v \) we get consequence \( u = n \)

\[ post(\Psi, S) = [u \xrightarrow{s} v \lor u \xrightarrow{s} v] \land [u \xrightarrow{s} v \lor u = v \lor u < v \lor (u = n \land mSucc \xrightarrow{s} v)] \]

• Program Point A43

\[ \phi_{A43} = R \land \neg lFound = -1 \land R_{sPS} \land R_{sP} \land R_{mPS} \land sPred \xrightarrow{s} mPred \land n \xrightarrow{s} mSucc \land n \xrightarrow{s} \text{nil} \]

\[ \land l \xrightarrow{s} mPred \land mSucc \xrightarrow{s} sSucc \land Ord_1 \land mNotReached(n) \land sNotReached(n) \]

The assertion at A42 holds at the program point A43 because the formula

\[ \forall u, v post(\Psi, S) \land \phi_{A43} \land \neg \Psi(u_0, v_0) \]

is unsatisfiable with the two substitutions \((u/mSucc, v/v_0), (u/u_0, v/v_0)\)

For program statement \( mPred.mNext := n \), apply the rule \( x \xrightarrow{s} y \) with \( t_1 = mPred \) and \( t_2 = n \) for instances \( x = u \) and \( y = v \) then we get consequences either \( u \xrightarrow{s} mPred \land n \xrightarrow{s} v \) or \( u \xrightarrow{s} v \)

\[ post(\Psi, S) = [u \xrightarrow{s} v \lor u \xrightarrow{s} v] \land [u \xrightarrow{s} v \lor u = v \lor u < v \lor (u \xrightarrow{s} mPred \land n \xrightarrow{s} v)] \]

• Program Point A44
\[ \phi_{A44} = R \land l\text{Found} = -1 \land R_{sPS} \land R_{sp} \land R_{mPS} \land s\text{Pred} \xrightarrow{m} m\text{Pred} \land n \xrightarrow{m} m\text{Succ} \land \text{Ord}_{1} \land s\text{NotReached}(n) \]

The assertion at A43 holds at the program point A44 because the formula

\[ \forall u, v \text{post}(\Psi, S) \land \phi_{A44} \land \neg\Psi(u_0, v_0) \]

is unsatisfiable with the two substitutions \((u/u_0, v/m\text{Pred}), (u/n, v/v_0)\) and \((u/u_0, v/v_0)\)

For program statement \(S\): \(n.\text{succ} := \text{succ}\), apply the rule \(x \xrightarrow{t_1} y\) with \(t_1 = n\) and \(t_2 = \text{succ}\) for instances \(x = u\) and \(y = v\) we get consequences either \(u \xrightarrow{s} n \land \text{succ} \xrightarrow{v} v\) or \(u \xrightarrow{v} v\)

then from the consequences apply the rule \(x \xrightarrow{y} s\text{NotReached}(y)\) with \(y = n\) for instances \(x = u\) and \(y = v\) we get consequence \(u = n\)

\[ \text{post}(\Psi, S) = \left[ u \xrightarrow{s} v \lor u \xrightarrow{m} v \lor (u = n \land \text{succ} \xrightarrow{v} v) \right] \land \left[ u \xrightarrow{m} v \lor u = v \lor u < v \right] \]

- Program Point A45

\[ \phi_{A45} = R \land l\text{Found} = -1 \land R_{sPS} \land R_{sp} \land R_{mPS} \land s\text{Pred} \xrightarrow{m} m\text{Pred} \land n \xrightarrow{m} m\text{Succ} \land \text{Ord}_{1} \land s\text{NotReached}(n) \]

The assertion at A44 holds at the program point A45 because the formula

\[ \forall u, v \text{post}(\Psi, S) \land \phi_{A45} \land \neg\Psi(u_0, v_0) \]

is unsatisfiable with the two substitutions \((u/s\text{succ}, v/v_0), (u/u_0, v/v_0)\)

For program statement \(S\): \(s\text{Pred}.\text{succ} := n\), apply the rule \(x \xrightarrow{t_1} y\) with \(t_1 = \text{succ}\) and \(t_2 = n\) for instances \(x = u\) and \(y = v\) then we get consequences either \(u \xrightarrow{s} \text{succ} \land n \xrightarrow{v} v\) or \(u \xrightarrow{v} v\)

\[ \text{post}(\Psi, S) = \left[ u \xrightarrow{s} v \lor u \xrightarrow{m} v \lor (u \xrightarrow{v} \text{succ} \land n \xrightarrow{v}) \right] \land \left[ u \xrightarrow{m} v \lor u = v \lor u < v \right] \]

- Program Point A46

\[ \phi_{A46} = R \land l\text{Found} = -1 \land R_{sPS} \land R_{sp} \land R_{mPS} \land s\text{Pred} \xrightarrow{m} m\text{Pred} \land n \xrightarrow{m} m\text{Succ} \land \text{Ord}_{1} \land s\text{NotReached}(n) \]

\[ \land n \xrightarrow{s} \text{succ} \land s\text{Pred} \xrightarrow{n} n \land l \xrightarrow{m} s\text{Pred} \land m\text{Succ} \xrightarrow{m} s\text{succ} \land \text{Ord}_{1} \]
The assertion at A45 holds at the program point A46 because the formula

$$\forall u, v \text{post}(\Psi, S) \land \phi_{A46} \land \neg \Psi(u_0, v_0)$$

is unsatisfiable with the two substitutions \((u/u_0, v/sPred), (u/n, v/v_0)\) and \((u/u_0, v/v_0)\)

Let us now consider the effects of interferences. We choose two possible interfering operations on two program point A31 and A39. We restrict to those that could cause some major global effects. One possible interference effect on the point A31 is that before the current thread locks the node \textit{mPred}, another thread comes earlier and locks \textit{mPred}. Whereas, at program point A39, before the current thread checks the valid condition to insert a new node, another thread comes and marks the node \textit{sSucc} before going to delete it. Let us now consider the effects of these interferences on the annotations above. We obtain:

At the program point A31:

$$\phi_{A311} = R \land \lnot \text{Found} = -1 \land R_{SP} \land R_{SP} \land R_{mPS} \land \text{Ord}_1 \land l \xrightarrow{m} sPred \land m \xrightarrow{m} sSucc$$

$$\land sPred \xrightarrow{m} mPred$$

when the program executes the statement \textit{S}: \textit{mPred.lock}(), we apply the rule \(x\text{.lock} = \text{OTHER}\) \(t \neq x\) \(\bot\)

in \(\phi_{A312}\) with \(t = \text{mPred}\) for instances \(x = \text{mPred}\) then we get consequences is \(\bot\).

At the program point A39

$$\phi_{A391} = R \land \lnot \text{Found} = -1 \land \text{mPred.lock} \land sPred.lock \land R_{SP} \land R_{SP} \land R_{mPS} \land \text{Ord}_1 \land l \xrightarrow{m} sPred$$

$$\land m \xrightarrow{m} sSucc \land sPred \xrightarrow{m} mPred$$

when the program executes the checking statement \(S\): \(\neg \text{sPred.marked} \land \neg \text{sSucc.marked} \land \text{sPred.sNext} = \text{sSucc}\) we get the consequence with only \(\phi_{A391}\) because the predicate \(\text{sPred.marked}\) in \(\phi_{A392}\) does not satisfy the statement \(S\).

As can be seen, the interferences from other threads do not effect on computation of the current thread in this algorithms. In the rest part of this chapter, we will show the program annotation of the remove algorithm.

5.2.3 Remove node

R01:    bool remove ( int t ) {
R02:       Node * nodeToDelete = null;
R03:    bool isMarked = false ;

R04: \textbf{int} topLayer = -1;
R05: Node* sPred, sSucc, mPred,mSucc;
R06: \textbf{while} ( true ) {
R07: \textbf{int} lFound = findnode(t,sPred,sSucc,mPred,mSucc)
R08: if(isMarked \lor lFound = 1 \land okToDelete(mSucc) \lor lFound = 2 \land okToDelete(sSucc)) {
R09: if (\neg isMarked) {
R10: if (lFound = 1) {
R11: nodeToDelete = mSucc ;
R12: topLayer = 1;
R13: }
R14: \textbf{else} {
R15: nodeToDelete = sSucc ;
R16: topLayer = 2;
R17: }
R18: nodeToDelete.lock.lock ( ) ;
R19: if (nodeToDelete.marked) {
R20: nodeToDelete.lock.unlock( ) ;
R21: return false ;
R22: }
R23: nodeToDelete.marked = true ;
R24: isMarked = true ;
R25: }
R26: \textbf{try} {
R27: if (topLayer = 1) {
R28: mPred.lock.lock();
R29: locklevel = 1;
R30: valid = \neg mPred.marked \land mPred.mNext = mSucc;
R31: if (\neg valid) continue;
R32: mPred.mNext = nodeToDelete.mNext;
R33: nodeToDelete.lock.unlock();
R34: return true;
R35: }
R36: if (topLayer = 2) {
R37: mPred.lock.lock();
R38: locklevel = 1;
R39: valid = \neg mPred.marked \land mPred.mNext == mSucc;
R40: if (\neg valid) continue;
R41: if (sPred \neq mPred) {
R42: sPred.lock.lock();
R43: locklevel = 2;
R44: }
R45: valid = \neg sPred.marked \land sPred.sNext == sSucc;
R46: if (\neg valid) continue;
R47: sPred.sNext = nodeToDelete.sNext; mPred.mNext = nodeToDelete.mNext;
R48: nodeToDelete.lock.unlock();
R49: return true;
R50: }

26
We annotate this algorithm with important program points include A28 which we can see interference effects easily and A32, A33, A47, A48 which are four points for removing existed nodes.

- **Program Point R28**
  \[
  \phi_{R28} = R \land lFound = 1 \land R_1 PS \land R_1 P \land R_{m} PS \land sPred \xrightarrow{s} mPred \land Ord_3 \land nodeToDelete = sSucc
  \]
  \[
  \land \text{topLayer} = 1 \land sSucc \xrightarrow{m} m \land l \xrightarrow{s} sPred \land mSucc \xrightarrow{m} sSucc
  \]

- **Program Point R32**
  \[
  \phi_{R32} = R \land lFound = 1 \land R_1 PS \land R_1 P \land R_{m} PS \land sPred \xrightarrow{s} mPred \land Ord_3 \land nodeToDelete = sSucc
  \]
  \[
  \land \text{topLayer} = 1 \land sSucc \xrightarrow{m} m \land l \xrightarrow{s} sPred \land mSucc \xrightarrow{m} sSucc
  \]

For \( S = m\text{Pred} \cdot m\text{Next} := \text{nodeToDelete} \cdot m\text{Next} \)
\[
\text{post}(\Psi, S) = \left[ u \xrightarrow{s} v \lor u \xrightarrow{m} v \land u \xrightarrow{m} m\text{Pred} \land m\text{Succ} \xrightarrow{m} v \land v \xrightarrow{m} s\text{Succ} \right] \land \left[ u \xrightarrow{m} v \lor v = v \lor u < v \right]
\]

- **Program Point R33**
  \[
  \phi_{R33} = R \land lFound = 1 \land R_1 P \land sPred \xrightarrow{s} mPred \land Ord_3 \land nodeToDelete = sSucc \land \text{topLayer} = 1
  \]
  \[
  \land sSucc \xrightarrow{m} m \land l \xrightarrow{s} sPred \land mSucc \xrightarrow{m} sSucc \land mPred \xrightarrow{m}
  \]

The assertion at A32 holds at the program point A33 because the formula
\[
\forall u, v \text{post}(\Psi, S) \land \phi_{A43} \land \neg\Psi(u_0, v_0)
\]
is unsatisfiable with the two substitutions \((u/m\text{Succ}, v/v_0), (u/u_0, v/m\text{Pred})\) and \((u/u_0, v/v_0)\)
\[
\Psi = \left[ u \xrightarrow{s} v \lor u \xrightarrow{m} v \right] \land \left[ u \xrightarrow{m} v \lor v = v \lor u < v \right]
\]

- **Program Point R47**
  \[
  \phi_{R47} = R \land lFound = 1 \land R_1 PS \land R_1 P \land R_{m} PS \land sPred \xrightarrow{s} mPred \land Ord_3 \land nodeToDelete = sSucc
  \]
  \[
  \land \text{topLayer} = 2 \land sSucc \xrightarrow{s} s \land sSucc \xrightarrow{m} m \land l \xrightarrow{s} sPred \land mSucc \xrightarrow{m} sSucc
  \]

For \( S = s\text{Pred} \cdot s\text{Next} := \text{nodeToDelete} \cdot s\text{Next}; m\text{Pred} \cdot m\text{Next} := \text{nodeToDelete} \cdot m\text{Next} \)
\[
\text{post}(\Psi, S) = \left[ u \xrightarrow{s} v \lor u \xrightarrow{m} v \land u \xrightarrow{m} m\text{Pred} \land m\text{Succ} \xrightarrow{m} v \land v \xrightarrow{m} s\text{Succ} \right] \land \left[ u \xrightarrow{m} v \lor v = v \lor u < v \right]
\]

- **Program Point R48**
  \[
  \phi_{R48} = R \land lFound = 1 \land R_1 P \land sPred \xrightarrow{s} mPred \land Ord_3 \land nodeToDelete = sSucc \land \text{topLayer} = 2
  \]
  \[
  \land sSucc \xrightarrow{s} s \land sSucc \xrightarrow{m} m \land l \xrightarrow{s} sPred \land mSucc \xrightarrow{m} sSucc \land mPred \xrightarrow{m} m \land sPred \xrightarrow{s}
  \]
The assertion at A32 holds at the program point A33 because the formula

$$\forall u, v \text{post}(\Psi, S) \land \phi_{A43} \land \neg\Psi(u_0, v_0)$$

is unsatisfiable with the two substitutions \((u/s, v/v_0), (u/v_0, v/s\text{succ})\) and \((u/u_0, v/v_0)\).

For interference test, we can see that at program point R28, one possible interference is that before \(m\text{Pred}\) node is locked by the current thread some other threads can lock the node \(m\text{Pred}\). Therefore, we may get two possible environments:

\[
\phi_{R28_1} = \begin{align*}
R \land l\text{Found} &= 1 \land R_sPS \land R_sP \land R_mPS \land s\text{Pred} \xrightarrow{m} m\text{Pred} \land \text{Ord}_3 \land \text{nodeToDelete} = s\text{succ} \\
\land \text{topLayer} &= 1 \land s\text{succ} \xrightarrow{m} m \land l \xrightarrow{m} s\text{Pred} \land m\text{succ} \xrightarrow{m} s\text{succ}
\end{align*}
\]

\[
\phi_{R28_2} = \begin{align*}
R \land l\text{Found} &= 1 \land R_sPS \land R_sP \land R_mPS \land s\text{Pred} \xrightarrow{m} m\text{Pred} \land \text{Ord}_3 \land \text{nodeToDelete} = s\text{succ} \\
\land \text{topLayer} &= 1 \land s\text{succ} \xrightarrow{m} m \land l \xrightarrow{m} s\text{Pred} \land m\text{succ} \xrightarrow{m} s\text{succ} \land m\text{Pred}\_\text{lock} = \text{OTHER}
\end{align*}
\]

When the current thread tries to lock \(m\text{Pred}\), the environment \(\phi_{R28_2}\) does not satisfy the requirement because its being locked by another thread. Hence, the interference operations from other threads do not effect on computation of this current thread at this program point.
Chapter 6

Conclusions and Future Work

The thesis has employed transitive closure logic in [2] to propose a method to infer and verify universal quantified invariants of algorithms with an unbounded number of threads that access a shared heap. Such algorithms are used to implement heap structures in common concurrency libraries. In particular algorithms implementing heap structures in which each heap node has several pointer field. For such algorithms, we have defined a simple template for their invariants, then inferred and verified the invariants automatically. To show how our method actually works, we have manually applied our method to concurrent skiplist algorithms from the paper [6], and a number of proof outlines have shown the correctness of our methods for these algorithms. However, our methods are sound because of undecidable problems in first order logic [4].

There are some obvious directions left for further improving the method of this thesis. Firstly, we plan to implement an actual framework. Secondly, we plan to develop the refinement technique which is the technique to refine abstraction domain when our framework does not give exact answers because of two reasons, either programs really violate properties or current abstractions are too coarse.
Bibliography


