Verifying Data Structures Using Jahob

Feride Cetin
Computer Science Masters Student
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
Email: feride.cetin@epfl.ch

Kremena Diatchka
Computer Science Masters Student
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
Email: kremena.diatchka@epfl.ch

***Abstract*—We have implemented some simple data structures in Java, provided specifications for properties they must specify, and attempted to verify that the implementation is correct with respect to these specifications using the verification tool Jahob. We have fully verified the implementations of a singly-linked list with a header node and a queue. All methods of a singly-linked cyclic list, doubly-linked cyclic list and leaf-linked tree also verify, but these classes require further work to ensure the specifications are correct in the initial state of the class. This report presents a brief overview of data structure verification using Jahob, and then continues with a detailed explanation of the specifications we provided. We conclude by summarizing our contributions, commenting on what we have learned, and suggesting directions for future work.**

I. Introduction

This paper describes our project for the Software Analysis and Verification class during the summer semester of 2007. We chose to learn about the subject of data structure verification in order to gain some insight into how much effort is required to formally verify that a data structure implementation is correct under all program executions. Achieving this goal means that, even for a simple piece of code, one has to think about many details and is forced to examine the implementation from all possible angles. We hoped that this experience would help us not only write better code in the future but also appreciate the effort required to annotate a program with specifications and why it is worth it to do so.

As outlined in [2], there are many good reasons why program verification is an important field of active research. Firstly, there has been progress in the development of techniques for program verification such as abstract interpretation and data-flow analysis, model checking, and advanced type systems. We have been introduced to these methods in class, and have seen that they may be useful as part of general verification systems. Secondly, research has been making advances in the fields of automated theorem proving and decision procedures, which may be used to check the validity of properties specified by the developer. Thirdly, advances in hardware allow the use of sophisticated, computationally intensive techniques for program verification. Finally, as computers continue making their way into many aspects of our lives, it is becoming increasingly important to attempt to use the results of these research efforts to create safe software with few errors to help save many headaches, money and, most importantly, lives.

One approach to program verification is the modular analysis of dynamically allocated linked data structures. As described in [2], such data structures are useful and thus common in many programs; however, they may be difficult to reason about, which means they are likely sources of errors which cause software to behave in an undesirable way. This type of analysis requires precise specification of program properties and is therefore not well-suited for whole program verification. The idea, then, is to reduce the number of errors in a program by verifying that common components of the program behave correctly.

These are the ideas that motivated the development of the Jahob verification system, which is currently focusing on data structure verification. However, the author of Jahob believes that it may also be promising as a more general purpose verification system [2]. Jahob is described in more detail in the Related Work section.

The goal of our project was to implement some simple linked data structures, provide their specifications, and use Jahob to verify the implementation. In the following sections we outline related work that helped us understand the process of data structure verification and gain a better grasp of field constraint analysis, which is a useful technique when it is not possible to deterministically define the set of objects a field may point to with respect to the backbone. We continue with a small sample program which illustrates the idea of data structure verification using developer-specified properties. Then we present our modest contributions, and finish with a discussion about the limitations we faced and directions for future work.

Related Work

a) Jahob: Jahob was developed by Viktor Kuncak and his collaborators as part of his PhD work at MIT. The following description of the Jahob system is mostly taken from Section 1.3 and 1.4 from the thesis [2] which resulted from his work.

Jahob is designed to be a usable verification system which can easily be learned so that any programmer may begin using it quickly. A subset of Java was chosen as the implementation language because it is a familiar, imperative, memory-safe language and existing Java tools can be used to write and compile the programs. The specifications are provided as

special comments in the code. The chosen subset is sufficient for implementing many data structures.

The programmer provides the specification of the program using classical higher-order logic(HOL) formulas, which is a notation that most computer scientists and mathematicians are comfortable with. The syntax is identical to that of the language of the theorem proving environment [4]. The advantage of this is that Isabelle can be used to prove formulas that Jahob cannot prove automatically.

Jahob is designed for modular verification of data structures. The programmer can reason about the modular components (classes and procedures) independently, verifying each procedure separately and then assume the correctness of the procedures when verifying the whole program. Reasoning about a small fragment of a program at a time makes the verification process easier and more scalable, and allows the use of very specific, expressive properties.

When verifying procedures, the developer specifies procedure contracts as pre and post conditions and a modifies clause.

- The **preconditions**, written as a `requires` clause, specify the properties that should be true at the entry point of the procedure
- The **frame condition**, written as a `modifies` clause, specifies which parts of the program state the procedure is allowed to modify during its execution
- The **postconditions**, written as an `ensures` clause, specify the properties that should be true after the procedure is finished executing.

Jahob also allows the use of specification variables to support data abstraction. These variables are like concrete Java variables but they exist only to help the programmer reason about the abstract properties of the data structure (as opposed to its concrete implementation) in order to simplify the verification process. Specification variables can be used to denote sets, functions and relations.

We conclude the introduction to Jahob by summarizing the overview of Jahob's approach to data structure verification presented in Section 1.4 of [2]. Jahob takes the HOL formulas provided in the specification of the program and splits them into a conjunction of independent smaller formulas. Each HOL conjunct is approximated by a formula in a logic which is easier to prove and then potentially different reasoning procedures are used to verify these approximations. There are three techniques for performing the approximation: translation to first-order logic, field constraint analysis with monadic second-order logic over trees, and Boolean Algebra with Presburger Arithmetic (BAPA).

First-order theorem provers such as SPASS [1] or E [5] can act as decision procedures for formulas which are translatable to first-order logic, and Jahob can use them to prove many data structure properties that fall in this category. The second technique, and the one we examined most closely in our project, is field constraint analysis. This technique allows the use of decision procedures designed for a restricted set of data structures to be used for a wider range of data structures. In particular, Jahob extends the use of monadic second order logic over trees so that it can be applicable to structures with non-deterministic fields, for example skip lists or linked combinations of data structures. Field constraint analysis is described in more detail in the corresponding section of the Related Work. The third technique allows reasoning about sets of elements and cardinalities of sets. Using BAPA reasoning makes it possible to specify relationships between the sizes of data structures or to specify data structure size using integer variables.

We now present the Pointer Assertion Logic Engine (PALE), which is a tool for data structure verification similar tool to Jahob.

b) The Pointer Assertion Logic Engine (PALE): The Pointer Assertion Logic Engine is a framework presented in [3] for verifying partial specifications of programs in order to catch type and memory errors and check data structure invariants. Unlike the previous specialized tools which were limited to verify only special data structures like lists or trees, the technique used in PALE can verify any data structure that can be expressed as a graph type which is a tree-shaped data structure with extra pointers. Moreover, it is still as fast as restricted verification tools.

In this method, programs are first annotated with partial specifications expressed in Pointer Assertion Logic, a notation for expressing properties of the program store. Then, the programs and partial specifications are encoded as formulas in monadic second-order logic. Finally, MONA tool is used to check the validity of these formulas, which also can provide explicit counterexamples to invalid formulas. In order to use PALE for verification, explicit loop and function call invariants should be provided in the program annotations. In PALE every statement of a given program is analyzed only once therefore, it is highly modular. This tool is implemented for the verification of safety-critical data-type algorithms, where the cost of annotating a program with invariants is justified by the gain from automatic verification of complex properties of the program.

However, PALE has a restriction for the verification of data structures because it requires every pointer field to point to exactly one object determined by the backbone of the data structure at any time during the program execution. This restriction prevents PALE tool from being used for the verification of data structures with non-deterministic pointer fields such as skip lists which have random pointer fields. However, the next method that we will describe in this report, Field Constarint Analysis, over comes this restriction.

c) Field Constraint Analysis: Field constraint analysis is a technique presented in [6] for verifying data structure invariants. A field constraint is a formula specifying a set of objects to which a field can point. Field constraints method first verifies the backbone of the data structure and then verifies the constraints on fields that cross-cut the backbone in arbitrary ways. This method enables the application of decidable logics to data structures which were originally beyond the scope of these logics. In previously implemented tools including PALE, such cross-cutting fields could only be verified when they were

uniquely determined by the backbone, which significantly limits the range of analyzable data structures.

With field constraint analysis it is possible to verify the invariants for data structures such as skip lists because it permits non-deterministic field constraints on cross-cutting fields. Nondeterministic field constraints also enable the verification of invariants between different data structures, which provides an expressive generalization of static type declarations. An implementation of this technique is presented in [6] which was implemented as a part of a symbolic shape analysis deployed in the context of the Hob system (predecessor of Jahob) for verifying data structure consistency. This implementation allows verification of data structures with non-deterministic fields and invariants between different data structures that were impossible to verify with similar techniques.

A Small Example

We use a short method written in Java to illustrate the approach of data structure verification we have taken, and to motivate its usefulness. Consider the following method which adds a node to a doubly-linked list:

```
public static void add(Node n) {
        n.next = first.next;
        n.prev = first;
        first.next.prev = n;
        first.next = n;
}
```

Such pointer manipulation is tedious and easy to get wrong, especially without spending some time to carefully sketch out exactly what each operation is doing. However, if the programmer takes the time to reason about the properties that a doubly-linked list should satisfy (for example, the backbone nodes should form a tree), then the main part of annotating the above method is specifying that the node being added was not in the list before the method was called, and that it is in the list after the method is finished. Then running Jahob on the above code will tell you if the implementation conforms to its specifications.

The idea is that reasoning about the properties of a program that should always hold is a more intuitive and more tractable process than attempting to manually go through the code and consider all possible effects of each of the operations.

II. Results

We now present the main work of the project - our attempts at verifying increasingly more complicated data structures. We begin with a simple linked list with a header node, then move to a cyclic singly and doubly linked lists and continue with queue with a header node and instantiable queue and then finish with a leaf-linked tree. In most of the implementations all the class variables and methods are static because for the purposes of verification we assume there is a single instance of the class. The idea is that it is most important to provide the specification and verify the implementation for a single instance, and the specification can then be extended, if necessary, to account for multiple instances of the same class.

While working on the project, we were made aware of the difference between complete and partial specifications, and the trade-offs one must consider when deciding the amount of detail to provide in a specification. A *complete* specification is one that characterizes precisely what the code does; any other specification which leaves room for ambiguity and uses some level of abstraction to describe the properties of the program is called a *partial* specification.

Here is a list of advantages and disadvantages of **partial specifications** that our professor reminded us about:

+ easier to write
+ easier to change implementation later
+ simplifies reasoning about the code (for example replacing a linked structure with a set), whereas a full specification would make it as difficult as inlining the implementation.
- the fact that the code meets the partial specification does not guarantee it is correct
- may not be sufficient to prove some properties (for example, if a property relies on the order of elements, and the structure is approximated with a set).

We kept these points in mind as we wrote the specifications for the programs we were trying to verify. In some cases, for simple methods, it was easy to write more complete specifications, but in some other instances we opted to abstract away some details and focus on specifying the most important properties.

Jahob Specification Constructs

The following explanations about the features and specification constructs in Jahob are taken from Section 3.2 of [2].

Claimed fields: Claimed fields allow the fields of one class to be declared local to another class. Semantically, a field is a function from an object to an object, so a field claimed by a class `C` is like the function is a private variable declared in `C`.

NoteThat statement: A `NoteThat e` statement is the same as an `Assert e` followed by an `Assume e`. It is sound because Jahob checks e before assuming it, and it can use e as a lemma when verifying subsequent formulas.

Ghost specification variables: A ghost variable is independent from any other variables and the only way its value can be changed is by a specification assignment of the form $ghost_var := e$ in the body of a procedure, where e is a valid HOL formula of the same type as the ghost variable.

Linked list with header node

In the following sections, we describe the specifications of the classes we worked on verifying using sentences rather than HOL formulas for readability. The formulas expressing the described properties can be found in the code submitted with this paper.

First we implemented a simple linked list class with a header node, which has three methods: `init()`, `add(Node n)` and

`member(Node n)`. This very simple implementation is only around 20 lines of code, and requires about as many lines of specification so that Jahob verifies it.

The helper class `Node` only contains a `next` field. The `List` class only has one public variable `first`, which should point to the header node of the list. Since procedure contracts of public procedures cannot mention private variables, we left `first` as public for convenience of writing the procedure contracts.

Global specification variables

content: A set of objects which includes all the nodes reachable from the node after the header node via the `next` field.

pointed: A function from object to boolean which returns true if the another node's `next` field points to the object.

Class invariants

First unaliased: States that no other node's `next` field should point to the header node.

isTree: States that the list has the properties of a tree with the `next` field as the backbone.

Procedure contracts and other annotations

init():

requires: the header node has not been instantiated

modifies: `first`

ensures: the header node has been instantiated and content is empty

add(Node n):

requires:

- the header node has been instantiated
- the node being added is not the header node
- the node being added is not already a member of the list
- the reference to the node is not null
- the `next` field of the node being added does not point to anything
- no other node points to the node being added (`pointed` specification variable used)

modifies: `content` specification variable and `pointed` specification variable (because the node being added is now being pointed to)

ensures: the node being added is inserted into the list

member(Node n):

requires:

- the header node has been instantiated
- the node being added is not the header node

modifies: nothing is modified

ensures: the method returns true if the node passed in is in the list, and false otherwise

Local specification variables: The ghost variable `seen` is a set of objects which have been "touched" as we walk over the linked list checking each node against the parameter node. One node gets added to it at every iteration of the while loop. After the while loop, a `NoteThat` statement expresses the fact that if we have walked through all the nodes and none of them are equal to the parameter node, then the `seen` set should be equivalent to the `content` set (i.e. we must have checked all the nodes in the list before we are sure we can say that the parameter node is not in the list).

Loop invariant:

- the current node we are examining must either be null or a member of the list
- `seen` must contain all the nodes up to the current node, and no nodes in the list after the current node
- none of the nodes in `seen` must be equal to the parameter node (otherwise we should already have returned `true`)
- the node being added is not the header node

QUEUE WITH HEADER NODE

We implemented a `Queue` class, which is quite similar to the `List` class, but it has the methods `enqueue(Node n)` which inserts a node at the end of the list (instead of at the beginning, like the `add(Node n)` method of the `List` class) and `dequeue()`, which removes a node from the front of the list. To facilitate this operation, the class has private `last` field of type `Node`.

The specification variables are the same as that for the `List` class, with the following additions:

Global specification variables

is Last: Returns true if a node is the last in the list (i.e. its `next` field is null), and false otherwise.

Class invariants

empty List: Specifies that a queue is empty if both the `first` and the `last` nodes are null (if one of them is null, then the other must be too).

Last is last: Ensures that the `last` field really points to the last node in the queue.

Procedure contracts and other annotations

The procedure contract for `enqueue(Node n)` is the same as the `add(Node n)` of the `List` class.

dequeue():

requires: does not require anything

modifies: `first`, `content` and `pointed` fields

ensures: if there is no header node or the header node is the only element in the queue, then the method returns null and `content` does not get modified. If there is at least one element besides the header node in the queue, then the node returned is a member of the queue and a node gets removed from the queue.

This is an example of a partial specification, because we do not verify that the node removed is the last one in the queue, only that a node does in fact get removed. If a

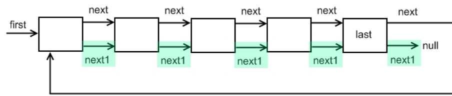

Fig. 1. Structure of a singly-linked cyclic list

particular application requires that specifically the last node gets removed, then the specification would have to be modified to include this property.

CYCLIC SINGLY-LINKED LIST

The next data structure that we implemented and verified is a cyclic singly-linked list with a header node, which has three methods: `init()`, `add(Node n)` and `member(Node n)`. Similar to the linked list structure we described above, it is a simple implementation which requires about the same amount of specifications. Its structure is illustrated in Figure 1.

The helper class `Node` only contains a `next` field. The `List` class only has one public variable `first`, which should point to the header node of the cyclic list. Since procedure contracts of public procedures cannot mention private variables, we left `first` as public for convenience of writing the procedure contracts.

Verification Status

All the methods in this class verify individually, but the initial state of the class does not meet the specifications. In particular, the problem is with the `next1field` invariant and with the field constraint. The class verifies in the initial state if the `next1field` invariant is removed and the field constraint is changed to:

```
ALL x y.
x : Object.alloc & Node.next x = y -->
((last x --> y = first) &
(~ last x --> y = x..next1))
```

However, adding the `x : Object.alloc` condition makes the verification of the `init()` and `member(Node n)` methods fail. We did not have time to investigate the reason for this so we mention it as one of the easy possibilities for further work.

Global specification variables

next1: A ghost specification variable from object to object which defines the backbone of the cyclic list. Basically, `next1` of an object in the cyclic list is equal to the `next` element following that object in the cyclic list. If the input object is the last element of the cyclic list, then its `next1` is set to null.

content: A set of objects which includes all the nodes reachable from the node after the header node via the `next1` specification variable.

isolated: A function from object to boolean which returns true if the input object's `next1` is null and it is not pointed by the `next1` of any not null object.

last: A function from object to boolean which returns true if the input object is not null and reachable from the header node using next1 but its `next1` is equal to null. In short, it returns true only if the input object is the last element of the cyclic list.

Class invariants

next1field: States that any unallocated object should be isolated.

isTree: States that the backbone of the cyclic list defined by the `next1` ghost specification variable has the properties of a tree.

First unaliased: States that no node's `next1` should point to the header node.

Next Field Constraint: States that the next field of any object in the cyclic list should be equal to the next1 of that object unless it is the last element of the cyclic list. The next field of the last element in the list should point to the header node.

Procedure contracts and other annotations

init():

requires: the header node has not been instantiated

modifies: `first`, `content`, `isolated`

ensures: the header node has been instantiated and content is empty

add(Node n):

requires:

- the header node has been instantiated
- the node being added is not the header node
- the node being added is not already a member of the list
- the reference to the node is not null
- the node being added is isolated

modifies: `content` specification variable and `isolated` specification variable (because the node being added is now being pointed to and its next1 also points to another node)

ensures: the node being added is inserted into the list

member(Node n):

requires:

- the header node has been instantiated
- the node being added is not null

modifies: nothing is modified

ensures: the method returns true if the node passed in is in the cyclic list, and false otherwise

Local specification variables: The ghost variable `seen` is a set of objects which have been “touched“ as we walk over the cyclic singly-linked list checking each node against the parameter node. One node gets added to it at every iteration of the while loop. After the while loop, a `NoteThat` statement expresses the fact that if we have walked through all the nodes and none of them are equal to the parameter node, then the `seen` set should be equivalent to the `content` set (i.e. we must have checked all the nodes in the list before we are sure we can say that the parameter node is not in the list).

Loop invariant:

- the current node we are examining must either be the header node or a member of the list
- `seen` must contain all the nodes up to the current node, and no nodes in the list after the current node
- if the current node is the header node then `seen` must contain all the nodes in the cyclic list
- none of the nodes in `seen` must be equal to the parameter node (otherwise we should already have returned `true`)

CYCLIC DOUBLY-LINKED LIST

Then we implemented and verified a very similar data structure, cyclic doubly-linked list with a header node, which also has three methods: `init()`, `add(Node n)` and `member(Node n)`. Similar to cyclic singly-linked list structure we have verified, cyclic doubly-linked list has a simple implementation but it requires slightly more specifications.

The helper class `Node` contains `next` and `prev` fields.

The `List` class only has one public variable `first`, which should point to the header node of the cyclic doubly-linked list. Since procedure contracts of public procedures cannot mention private variables, we left `first` as public for convenience of writing the procedure contracts.

Global specification variables

Global specification variables of the cyclic doubly-linked list are same as the ones we used for the cyclic singly-linked list.

Class invariants

For cyclic doubly-linked list we used the following invariants together with the class invariants we have in cyclic singly-linked list.

Prev Field Constraint: States that prev field of the last element in the list should point to the header node and all the nodes pointed by the prev field of another node should point to the same node by their next field.

contentCheck: States that if an objects next field is equal to the objects next1 then the object pointed by next must be an element of the content.

Procedure contracts and other annotations

The procedure contracts and annotations of `init()` and `add(Node n)` methods of cyclic doubly-linked list are same as the ones we used for the cyclic singly-linked list.

member(Node n):

requires:

- the header node has been instantiated
- the node being added is not null

modifies: nothing is modified

ensures: the method returns true if the list is not empty and the node passed in is in the cyclic list, and false otherwise

Local specification variables: The ghost variable `seen` is a set of objects which have been "touched" as we walk over the cyclic singly-linked list checking each node against the parameter node. One node gets added to it at every iteration of the while loop. After the while loop, a `NoteThat` statement expresses the fact that if we have walked through all the nodes and none of them are equal to the parameter node, then the `seen` set should be equivalent to the `content` set (i.e. we must have checked all the nodes in the list before we are sure we can say that the parameter node is not in the list).

Loop invariant:

- the current node we are examining must either be the header node or a member of the list
- `seen` can be empty for the initial case otherwise it must contain all the nodes up to the current node, and no nodes in the list after the current node
- if the current node is the header node then `seen` must contain all the nodes in the cyclic list
- none of the nodes in `seen` must be equal to the parameter node (otherwise we should already have returned `true`)

LEAF-LINKED TREE

The last data structure we have implemented is leaf-linked tree, whose backbone is a binary search tree and whose leaves form a doubly-linked list (see Figure 2). Our motivation for choosing this data structure is its broad functionality and better complexity for some functions achieved by its special structure which combines the tree structure with doubly-linked list. In our implementation all the values inserted in the tree are held as leaves so functions like search, insertion and deletion whose complexity is linear for linked lists is $log(N)$ in the average case for leaf-linked tree. In addition, it is much easier to traverse all the values using the doubly-linked list of leaves compared to a normal tree.

In our implementation we have six methods: `isEmpty()`, `isLeaf(Node n)`, `getRoot()`, `findSmallestLeaf()`, `leafUpdate(Node n)`, `add(int v)`.

`isEmpty()` simply returns true if there is no value inserted in the leaf-linked tree, otherwise it returns false.

`isLeaf(Node n)` returns true if the input Node is a leaf in the leaf-linked tree, otherwise it returns false.

`getRoot()` returns the current root node.

`findSmallestLeaf()` returns the leaf node with the smallest value in the leaf-linked tree.

`add(int v)` inserts the input value in the backbone binary search tree as a value of a leaf node.

`leafUpdate(Node n)` is called inside `add(int v)` method and it updates the doubly-linked list of leaves after the insertion of a new value.

The helper class `Node` contains `right`, `left`, `parent`, `next`, `prev`, `v` fields.

Verification Status

All the methods in this class verify individually, but the initial state of the class does not meet the specifications. In

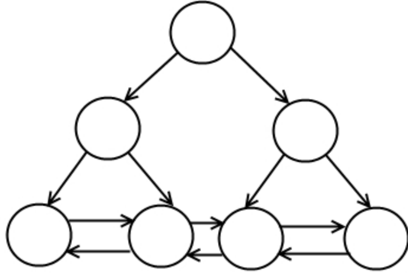

Fig. 2. Structure of a leaf-linked tree

particular, the problem is with the LeftNotRight invariant and with the parent field constraint. If those are slightly changed as the following by adding the condition of the objects being members of nodes and the root case in the parent field constraint, then the initial state of the class verifies but the leaf related functions do not verify probably because of the temporary violation of the invariants.

```
invariant LeftNotRight:
"ALL x. x : nodes -->
x..Node.left ~= x..Node.right";)

invariant ParentFieldConstraint:
"ALL x y. x:nodes & Node.parent x = y -->
((x = root --> y = null) &
(x ~= root -->
(x ~= null -->
((y..Node.left = x |
  y..Node.right = x) &
  (y : internalNodes)))))";
```

Time did not allow us to attempt to verify order properties of the tree, although there are some examples of such specifications which are fairly long and complex.

Verifying next node

We created a specification which states that for a leaf L in the linked list, the node that its next field is pointing to is the leaf that is next to it in the tree structure (henceforth referred to as the *neighbor* of L). For this purpose, we first implemented an algorithm which finds the neighbor of a leaf. Then we created a specification variable which is a function that checks whether a node's next field is set correctly when that node is added to the tree. The algorithm we used in our specification is presented in Algorithm 1.

Global specification variables

Nodes: a set of objects which contains all the nodes in the tree reachable from the root node via the left or right fields.

Content: the integer values of the nodes in the Nodes set.

Internal nodes: all those nodes for which at least one of the left or right nodes is not null

input : Node n
output: Neighbor of n in the tree
1 $p \leftarrow$ parent of n;
2 $ancestors \leftarrow$ all nodes reachable from n via parent fields;
3 $descendants \leftarrow$ all nodes in subtree rooted at n;
4 $result \leftarrow$ null;
5 **if** p *is null* **then**
6 | $result \leftarrow$ null;
7
8 **else**
9 | **if** n *is left child of* p **then**
10 | | $p.r \leftarrow$ right child of p;
11 | | $result \leftarrow SmallestNode(p.r)$;
12 | **else**
13 | | $splitNodes \leftarrow$ all ancestors whose right children are not ancestors ;
14 | | $firstSplitNode \leftarrow$ node in $splitNodes$ which does not contain any other members of $splitnodes$ in its $descendants$;
15 | | $s.r \leftarrow$ right child of $firstSplitNode$;
16 | | $result \leftarrow SmallestNode(s.r)$;
17 | **end**
18 **end**
19 **return** $result$;

Algorithm 1: Algorithm to find the neighboring leaf of a leaf

Left nodes(Node n): the set of nodes reachable from n via the left field, including n. Needed for nextLeaf function.

Ancestors (Node n): the set of nodes reachable from n via the parent field, including n. Needed for nextLeaf function.

Descendants (Node n): the set of nodes reachable from n via the left or right fields, not including n (equivalent to the nodes contained in a subtree rooted at n minus n itself. Needed for nextLeaf function.

Split nodes(Node n): contains all the nodes which are members of ancestors of n but whose right child is not a member of ancestors of n. Needed for nextLeaf function.

Split node(Node n): the node which is a member of splitNodes of n and whose descendants are not members of splitNodes of n. Needed for nextLeaf function.

is left child (Node n): returns true if n is a left child of its parent, and false otherwise. Needed for nextLeaf function.

is smallest (Node n, Node aRoot): returns true if n is the left-most leaf of the subtree rooted at aRoot, and false otherwise. Needed for nextLeaf function.

is smallest from root (Node n): returns true if n is the left-most leaf of the whole tree rooted at root, and false otherwise. This was added to so that it may be checked that the public method findSmallestLeaf() method returns the correct result without including root (which is a private variable) in the procedure contract.

next Leaf (Node n, Node nnext): this function returns true if `nnext` is the neighbor of `n`, and false otherwise. This function basically implements Algorithm 1.

Class invariants

Tree invariant: the structure formed by the nodes connected by the `left` and `right` fields satisfies the tree properties.

Left not right: the `left` and `right` fields of a node must not point to the same node.

Root not pointed: if root is not null then no node exists whose `left`, `right`, `next` or `prev` fields point to root.

Root parentless: if root is not null then its parent field should be null.

Field constraint on all Node fields: all `Node` fields must point to nodes in this tree (i.e. nodes that are members of the `Nodes` set).

Field constraint on parent field: if a node `x` has a parent, then there exists a node `y` whose `left` or `right` field points to `x`.

Procedure contracts and other annotations

For brevity, we do not include here the small methods `isEmpty()`, `isLeaf()` and `getRoot()` because they are simple to verify and their specifications are easy to understand.

findSmallestLeaf():

ensures: the return value is either null or a the left-most leaf node of the tree (the specification variable `isSmallestFromRoot(res)` is used here)

Loop invariant: While traversing the binary tree to find the smallest leaf node, we use a temporary node. Since the loop continues as long as the temporary node is not a leaf, the loop invariant ensures that the temporary node is an internal node and that it is reachable via the `left` fields starting at `root` in each iteration.

leafUpdate(Node n):

requires: `n` is a leaf node that has a parent node

ensures: the `next` field of `n` is `n`'s neighbor in the tree.

add(int v):

requires: `v` is not a value in the content set (This can be modified if the programmer decides that a value is allowed to occur in the list more than once)

modifies: `content`, `nodes`, `internalNodes`

ensures: `v` was added to `content`

Loop invariant: While traversing the binary tree to find the correct place to insert the new node, we use a temporary node. The loop invariant checks if the temporary node is a member of `nodes` in each iteration.

III. Conclusion

The final status of our project is as follows: we have fully verified the implementations of a singly-linked list with a header node and a queue, and all methods of a singly-linked cyclic list, doubly-linked cyclic list and leaf-linked tree also verify, but these classes require further work to ensure the specifications are correct in the initial state of the class.

When we began working on this project, we implemented very simple data structures which were easy to implement correctly. Therefore, as we were writing the specifications for those implementations, we were mostly sure they were correct, and a problem with the verification usually resulted from a problem with the specification rather than with the implementation. However, when we were writing the specifications for the leaf-linked tree (especially the neighbor node algorithm), we actually found some bugs in the implementation which we had to fix before the the methods would verify. In this case we really saw the utility of using a verification system like Jahob to verify data structures. It was interesting trying to get both the specification and the implementation correct, because thinking about them at the same time helped us realize mistakes in both. This experience made us realize that although it is a time-consuming and frustrating activity, annotating a program with specifications and checking whether it verifies is actually useful and helpful, even to ordinary programmers like ourselves.

We enjoyed using Jahob because we were already familiar with HOL, so we did not have too many problems with the syntax of the specifications. Another positive aspect of Jahob is its modular approach to data structure verification; verifying fragments of code independently made the task of verifying an entire data structure much easier.

Our main criticism regarding the use of Jahob for data structure verification is the fact that the automation starts to break down even when trying to prove some properties that at first glance seem simple. Some programs require more specifications that actual code, and although we mentioned that we saw the utility in doing this, there is a limit to how much time and effort a programmer is willing to spend to ensure the complete correctness of his/her implementation. However, we note that for most of the project we were Jahob beginners; more experienced programmers and Jahob users could find a tool like Jahob even more helpful than we did.

This project has many possibilities for further work, as we just scratched the surface of verifying simple data structures. Since our work was incomplete, one could continue this project by fully verifying the cyclic list classes and the leaf-linked tree (all methods *and* the initial state). A following step could be working on verifying instantiable classes. We briefly considered this with the queue class, but had some problems and decided we would learn more if we concentrated on verifying classes which have a single instance. Furthermore, we realize that Jahob could be used for verifying more advanced programs, and for specifying more precise properties. It would be interesting to see the kind of implementations for which using Jahob would *not* be appropriate (because of time or complexity of specifications, for example), so that its utility could be determined more concretely.

In conclusion, we have had a positive experience with this project, because we were forced to critically examine every aspect of our implementation as we were writing the specifications. This kind of attention to detail will be useful in any programming work we do in the future, even if we are not formally verifying the program using a verification tool such as Jahob.

References

[1] Max Planck Institut Informatik. Spass: An automated theorem prover for first-order logic with equality. *http://spass.mpi-sb.mpg.de/*.

[2] Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, 2007.

[3] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 221–231, New York, NY, USA, 2001. ACM Press.

[4] Lawrence C. Paulson and Tobias Nipkow. Isabelle theorem proving environment. *http://isabelle.in.tum.de/*.

[5] Stephan Sculz. The e equational theorem prover. *http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover.html*.

[6] T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. On field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpretation, 2006.*, 2006.