A component-based approach to proving the correctness of the Schorr-Waite algorithm

Amrinder Singh

Thesis submitted to the faculty of the Virginia Polytechnic
Institute and State University in partial fulfillment
of the requirements for the degree of

Master of Science
In
Computer Science

Dr. Gregory W. Kulczycki, Chair
Dr. Ing-Ray Chen
Dr. Chang Tien Lu

August 9th 2007
Falls Church, Virginia

Keywords: Formal specification, modular reasoning, graph marking, memory management

A component-based approach to proving the correctness of the Schorr-Waite algorithm

Amrinder Singh

Abstract

This thesis presents a component-based approach to proving the correctness of programs involving pointers. Unlike previous work, our component-based approach supports modular reasoning, which is essential to the scalability of systems. Specifically, we specify the behavior of a graph-marking algorithm known as the Schorr-Waite algorithm, implement it using a component that captures the behavior and performance benefits of pointers, and prove that the implementation is correct with respect to the specification. We use the Resolve language in our example, which is an integrated programming and specification language that supports modular reasoning. The behavior of the algorithm is fully specified using custom definitions, pre- and post-conditions, and a complex loop invariant. Additional operations for the Resolve pointer component are introduced that preserve the accessibility of a system. These operations are used in the implementation of the algorithm. They simplify the proof of correctness and make the code shorter.

Acknowledgements

I would like to thank my advisor, Gregory Kulczycki, for his support, care, and patience during my time as a graduate student. His insight and ideas helped form the foundation of this thesis. I would also like to express my gratitude to Ing-Ray Chen and Chang Tien Lu for being on my thesis committee.

I am also thankful to the Computer Science department, the faculty and staff. Being here has been an incredible learning experience. I shall always remember the years spent here as a highpoint in my life.

Finally I would like to thank my family and friends for encouraging me during the uneasy times.

Table of Contents

List of Figures

1. Introduction

Static reasoning requires programmers to be able to predict the behavior of a program without executing it. The ideal of static reasoning is full formal verification. To formally verify a section of programming code, one must prove that the code is correct with respect to its specification. This means that the intended behavior of the code must be formally specified. A formal specification of a section of code is a mathematical description of the code's behavior.

Introductory examples of formal specification tend to be simplistic [14]. Specification of code involving pointers or references is more complex than specification of code that does not, because the specification of pointer-based code must account for the extra level of indirection represented by the pointers [8]. The specification and proof of the Schorr-Waite algorithm is considered a non-trivial, essential first step for any formal approach to reasoning about pointers.

In past research efforts, various proof systems have been used along with various implementation languages to prove the correctness of the Schorr-Waite algorithm. However, none of these approaches were component-based, and none of them demonstrated how they supported modular reasoning. In a component-based approach, a program is made of up individual components. Each component consists of a specification and an implementation. More than one implementation may satisfy a specification. A component-based approach facilitates modular reasoning. In modular reasoning, components can be proved correct using only the specifications (not the implementations) of the components they import. Modularity in reasoning is essential to scalability.

1.1 Objective

The objective of the thesis is to show that pointer programs can proved correct in a component-based system that supports modular reasoning. We do this by specifying the behavior of the Schorr-Waite algorithm in the Resolve specification language [12], implementing the Schorr-Waite algorithm using the Resolve pointer component [5], and proving that the implementation is correct with respect to the specification.

Resolve is an integrated programming and specification language that has been developed with full formal verification in mind. It uses value-based semantics, is component-based, and supports modular reasoning. The Resolve pointer component was presented in [5]. It has been applied to a simple splice example in [6].

The major contributions of this thesis are:

- A full formal specification in Resolve of the operation "Mark_Accessible_From," which is the operation that the Schorr-Waite algorithm implements. The specification includes custom definitions, and pre- and postconditions.
- An introduction to swapping-based, accessibility-preserving operations. These are a set of new operations that are being proposed for inclusion in the Resolve pointer component as primary operations. These operations decrease the code size of the Schorr-Waite algorithm and aid in the proof of certain reachability assertions.
- Two implementations of the Schorr-Waite algorithm using the Resolve pointer component. One uses more traditional operations similar to the implementations found in [9] and [4], and the other uses the accessibility-preserving operations introduced in this thesis. Both implementations contain the same sophisticated loop invariant, and a decreasing metric for the loop.
- A mathematical proof that the accessibility-preserving implementation of the Schorr-Waite algorithm is correct with respect to its specification. The proof centers around the proof of the complex loop invariant, and consists of three main parts: initialization, termination and maintenance.

1.2 Modular Reasoning

One of the main reasons to define fully specified software components is that they facilitate modular reasoning. Reasoning about a component is modular when it depends only on the specifications of other components and does not require the programmer to reason about other implementations. Because modular reasoning depends on specifications, it is also called specification-based reasoning. Implementation-based

reasoning is reasoning about a component that requires the implementation of other components. To see why this is problematic, consider the system in Figure 1.1.

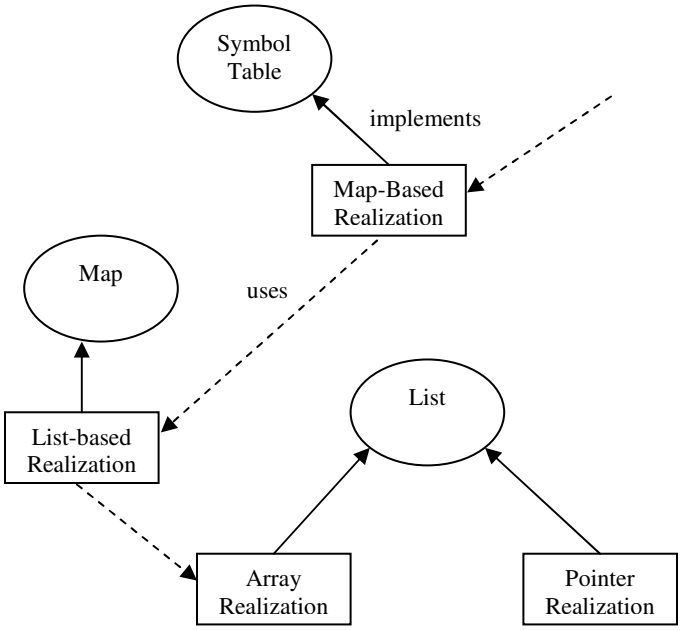


Figure 1.1 Implementation-based Reasoning for Symbol Table Component

The system uses implementation-based reasoning. The map-based implementation for Symbol Table depends on the list-based implementation for Map which in turn depends on an array-based implementation for List. To reason about the symbol table, we must understand the list-based Map, which, in turn, requires us to understand the array-based list. This means that a small change in the array-based implementation of List can force us to modify the both the list-based Map and the map-based Symbol Table. In general, we may have to modify all the components above it. At the very least, we will have to reason about these components again to ensure their correctness. Also, we also cannot swap one implementation for another without potentially changing the code in all the components that depend on it. Such a system is not scalable. We need to be able to modify or replace a component without having to re-reason about the entire system. This is what modular reason allows us to do.

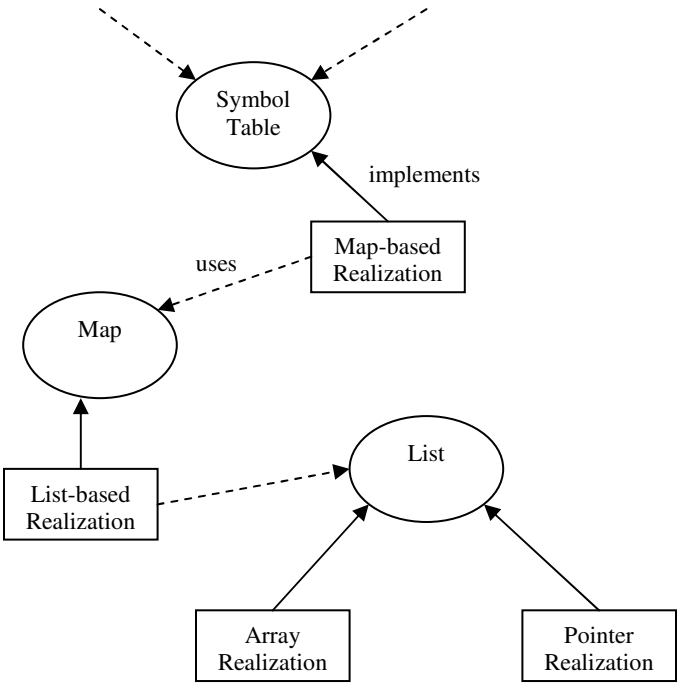


Figure 1.2 Specification-based Reasoning for Symbol Table Component

In specification-based reasoning, the implementation of any component depends only on the *specifications* of the components it imports. Figure 1.2 shows an example of a system that uses specification-based reasoning. Changes made to an implementation of a component do not require us to re-reason about components that depend on it. For example, in the system in Figure 1.2, we can substitute the pointer-based implementation of List for the array-based implementation of List, without having to re-reason about the entire system. We need only ensure that the pointer-based implementation is correct with respect to the List specification.

Specification-based reasoning is also known as modular reasoning. Modular reasoning is important for maintainability of systems and essential to scalability. Scalability is achieved because reasoning about a component can be localized: it does not require programmers to reason transitively about all dependent components and all subcomponents. The specifications of the subcomponents provide all the information needed by the component, and the specification of the component provides all the information need by dependent components.

In component-based software development, a program can be viewed as a sophisticated component built over several large components that are themselves comprised of more basic structures. Therefore, all components ranging from the most basic component all the way to the system level has a specification and one or more corresponding implementations associated with it. Assuming that a program's specification accurately reflects what the program is intended to do, a bug in the program means that the implementation is not correct with respect to the specification. This can happen because of an error in the code of the top-level component, or it can happen because one of the subcomponents is not correct with respect to its specification. Either way, when errors exist, we know that some contract is being violated and we can find out where it happens.

1.3 Outline

The rest of the thesis is organized as follows. Section 2 provides an informal description of the Schorr-Waite algorithm and a summary of some of the recent research papers that demonstrate the correctness of the Schorr-Waite algorithm under various proof systems. It also gives an overview of the Resolve pointer component.

Section 3 provides a formal specification of the Mark_Accessible_From operation in Resolve. This operation is an enhancement to the Resolve pointer component. This section also gives two different implementations of Mark_Accessible_From. Both of them use the Schorr-Waite algorithm. The first uses traditional operations from the Resolve pointer component, and the second uses accessibility-preserving operations, which are introduced in this section as well.

Section 4 presents a mathematical proof that the implementation of the Schorr-Waite algorithm is correct with respect to its specification. The proof centers on the proof of the complex loop invariant in the algorithm. It includes sub-proof for initialization, termination and maintenance of the invariant.

Section 5 contains the direction for future work and the conclusions.

2. Background and Related Work

The formal proof of the Schorr-Waite algorithm has been a target of several research efforts. It is typically used as evidence that (1) the pointer code or pseudo-code used in the research is sufficient to implement the algorithm, and (2) the formalism presented for the particular implementation of pointers is sound – i.e., it can prove the correctness of the algorithm. In [1], Bornat states that "The Schorr-Waite algorithm is the first mountain that any formalism for pointer aliasing should climb" (page 20).

2.1 The Schorr-Waite Algorithm

The first step of a Mark-Sweep garbage collector involves marking all nodes that are accessible to the program. Any graph marking algorithm can be used to mark nodes accessible from a given node. Most of them involve recursively moving through the graph in a depth-first or breath-first fashion. However, using recursion uses a lot of memory for the system calls required. Using iteration, along with a stack, is another possible implementation for the marking phase of the garbage collector. Iteration will be less demanding on resources than recursion. However, using a stack data structure means that procedure calls must be used. These calls can be expensive. Therefore, for cases where memory and time are scarce, as in garbage collection, an alternative method is required.

The Schorr-Waite algorithm marks the nodes of a graph in a depth-first fashion without the use of an explicit stack. It manipulates the links between locations to form a conceptual stack. This stack can then be used to backtrack through the graph and mark all locations that are reachable from a given location. Two local variables are used by this algorithm: the variable *t* keeps track of the current node, and *p* keeps track of the predecessor node. Since we know the location of both *t* and *p*, the link from the *p* to *t* can be used to serve other purposes. The algorithm uses this link to point the predecessor of *p*. These links form a conceptual stack from *p* all the way back to the root node. Using this stack in conjunction with local variables *p* and *t*, we can backtrack through the graph and restore all links to their original positions.

Each location is associated with two state variables which reflect (1) whether the given location is marked and (2) whether the control bit is set for the location. We assume that

each node can have just two links, a left link and a right link. However, the algorithm can be generalized to handle an arbitrary number of links.

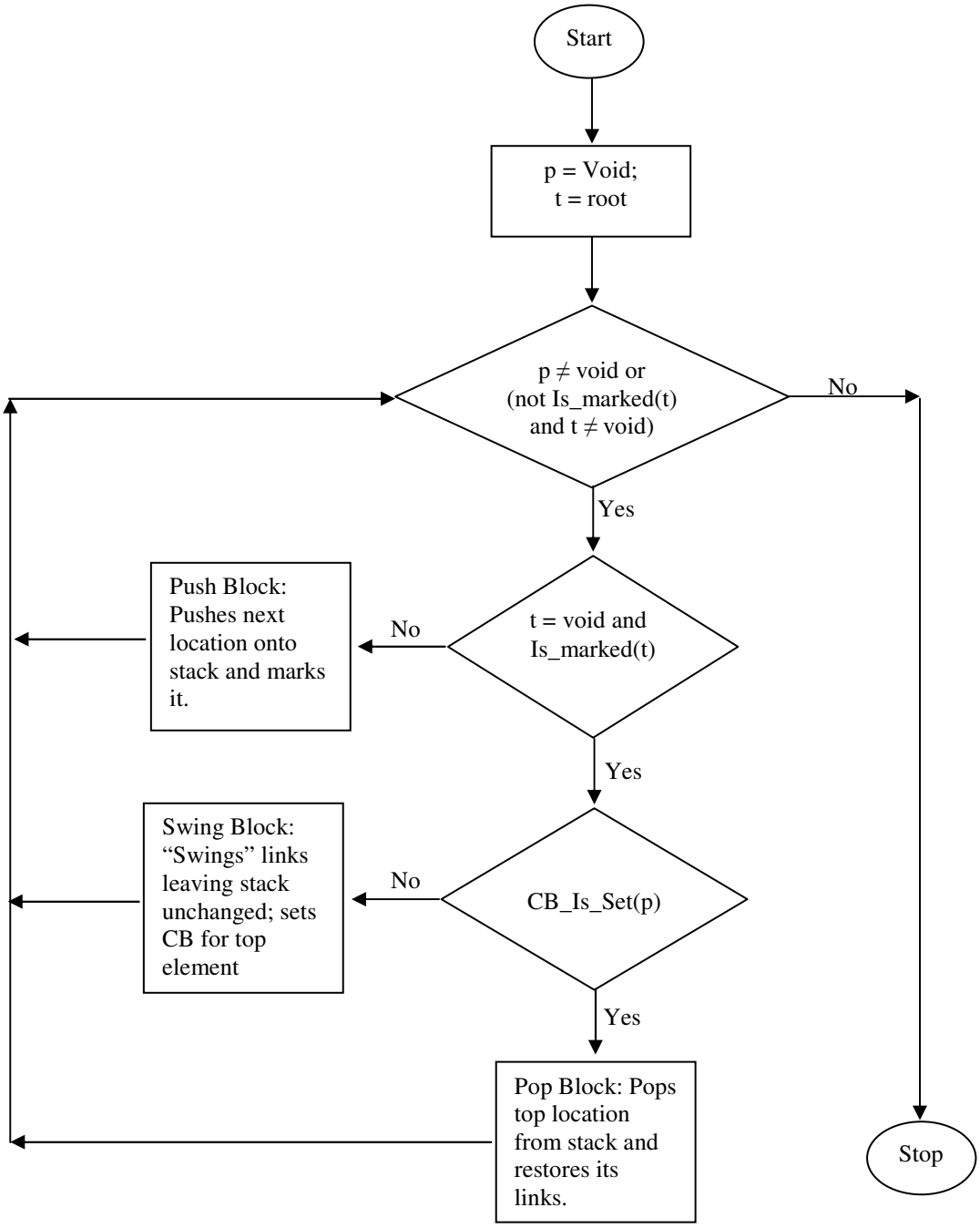


Figure 2.1 Control Flow for the Schorr-Waite algorithm

The algorithm starts with setting t = *root* and initializing p to the Void location. It then executes a loop until all the locations reachable from the root are marked. Depending on the values of p and t, the marked status of these two variables and the control bits associated with them, one of three sets of statements are performed inside the loop. If t is located at a non-Void location that is not marked, then the push block is executed. Otherwise, depending on the control bit of p, either the pop or swing block is executed. The flowchart in Figure 2.1 gives the control flow for the Schorr-Waite algorithm. The push block advances both p and t and makes p's left link to point to its predecessor. The swing block does not change p. It changes t's location to right link of p. It also restores p's left link while making its right link point to its predecessor. The pop block restores the right link of p. It also changes both p's and t's location to that of their predecessors.

The trace in Figures 2.2 through 2.5 shows how the Schorr-Waite algorithm works on a particular example. Each graph represents the state of the system at the beginning of a particular loop iteration. The circles are locations, a solid arrow represents the right link of a location, and a dotted arrow represents the left link of a location. The circle with a slash through it represents the Void location. A location with an M inside of it is marked, and a location with a dotted circle inside of it has its control bit set.

The conceptual stack consists of the grayed locations. The original graph can always be obtained from any intermediate state that the graph is in. The conceptual stack has p as its top element. The next node is obtained by following either the left or right link of the previous node depending on the control bit. If the control bit is not set, the left link is followed to obtain the next node in the location stack. If the control bit is set, the right link is followed to obtain the next node in the location stack.

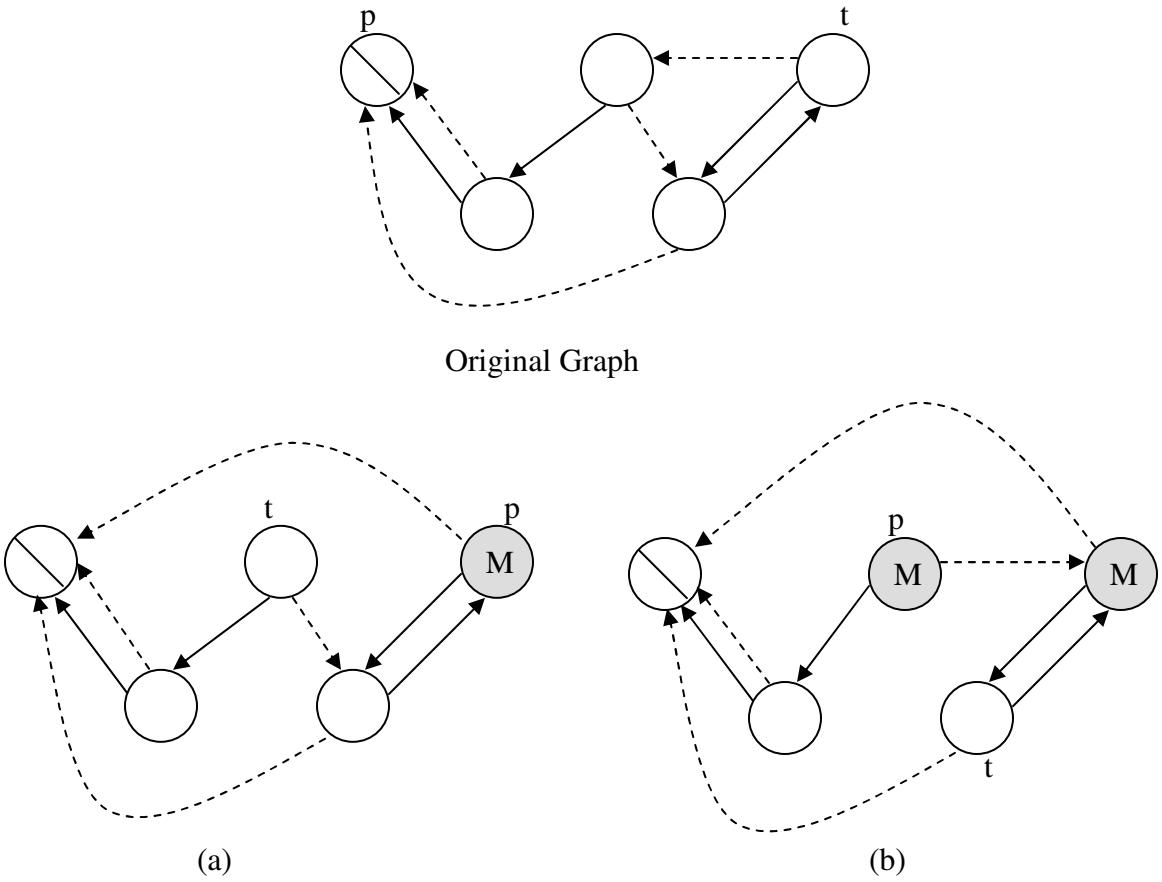


Figure 2.2 System states at the beginning of the first, second, and third iterations of the loop.

The original graph is shown in top of Figure 2.2. During the first iteration the push block is executed, resulting in the graph in (a). The push block is executed again transforming the system to the graph in (b). In both cases, the list of predecessors starts from *p* and ends at the original root, which in turn points to Void. Note that the original left link of *p* in (a) can be restored by making it point to t. The same can be said for *p* in (b). The right links for all nodes remain the same. Both left and right links for all nodes *not* on the conceptual stack are in their original positions. This last condition is true throughout the execution of the algorithm.

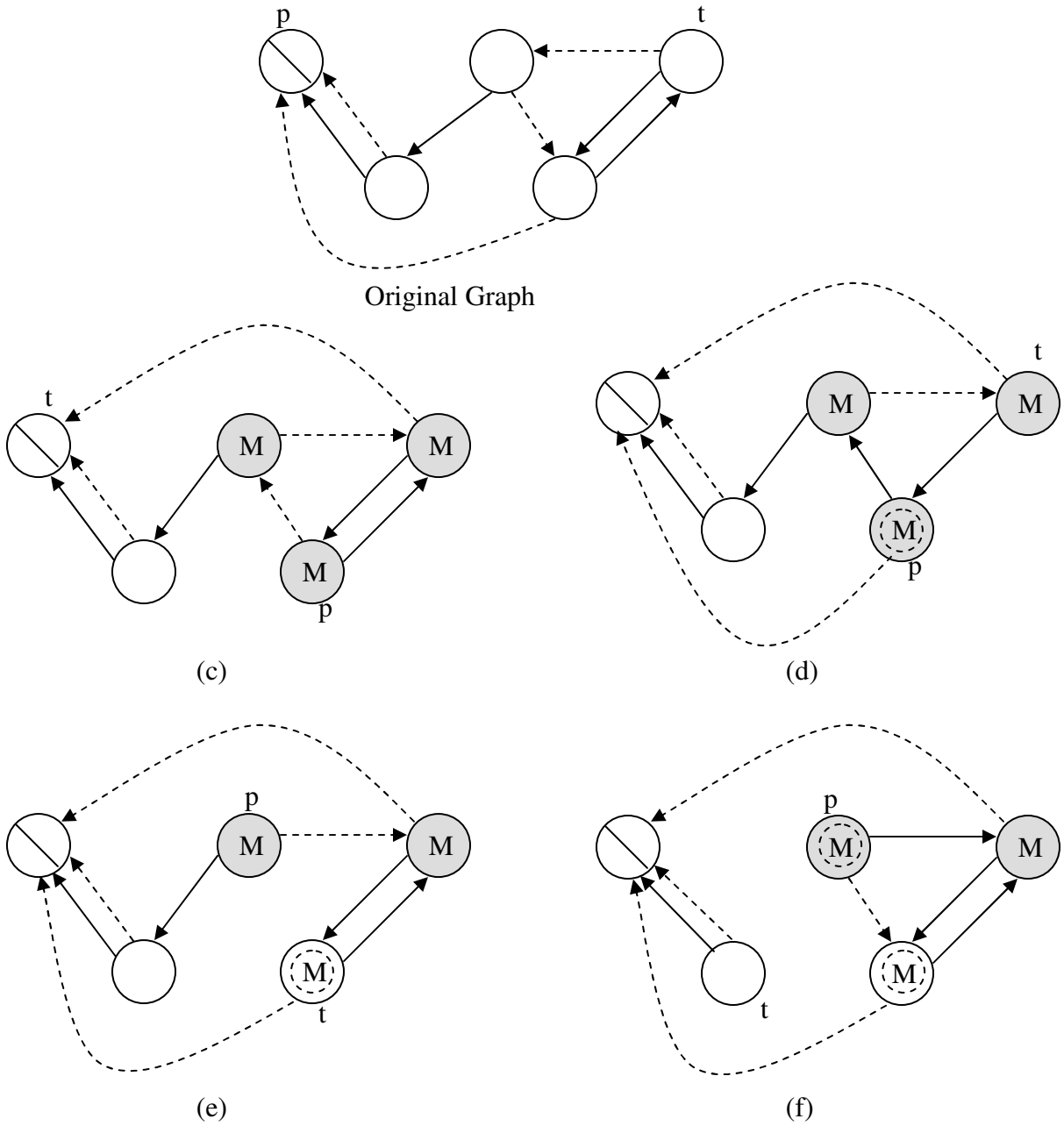


Figure 2.3 Original state and System states at the beginning of iterations four through seven

During the third iteration the push block is executed resulting in the graph in (c). The same conditions hold for this graph as those described for (a) and (b). The swing block is executed during the fourth iteration to obtain graph in (d). We can see that the left link for *p* in now restored and the right link points to its predecessor. All other links remain the same. Also the control bit is set for *p*. The pop block is executed in the fifth iteration. We can see that the list of predecessor nodes reduces by one. Both links for the node removed

from this list are restored. The swing operation is performed in the next iteration as well to get the graph in (e). The left link for *p* is restored and its current right link points to its predecessor.

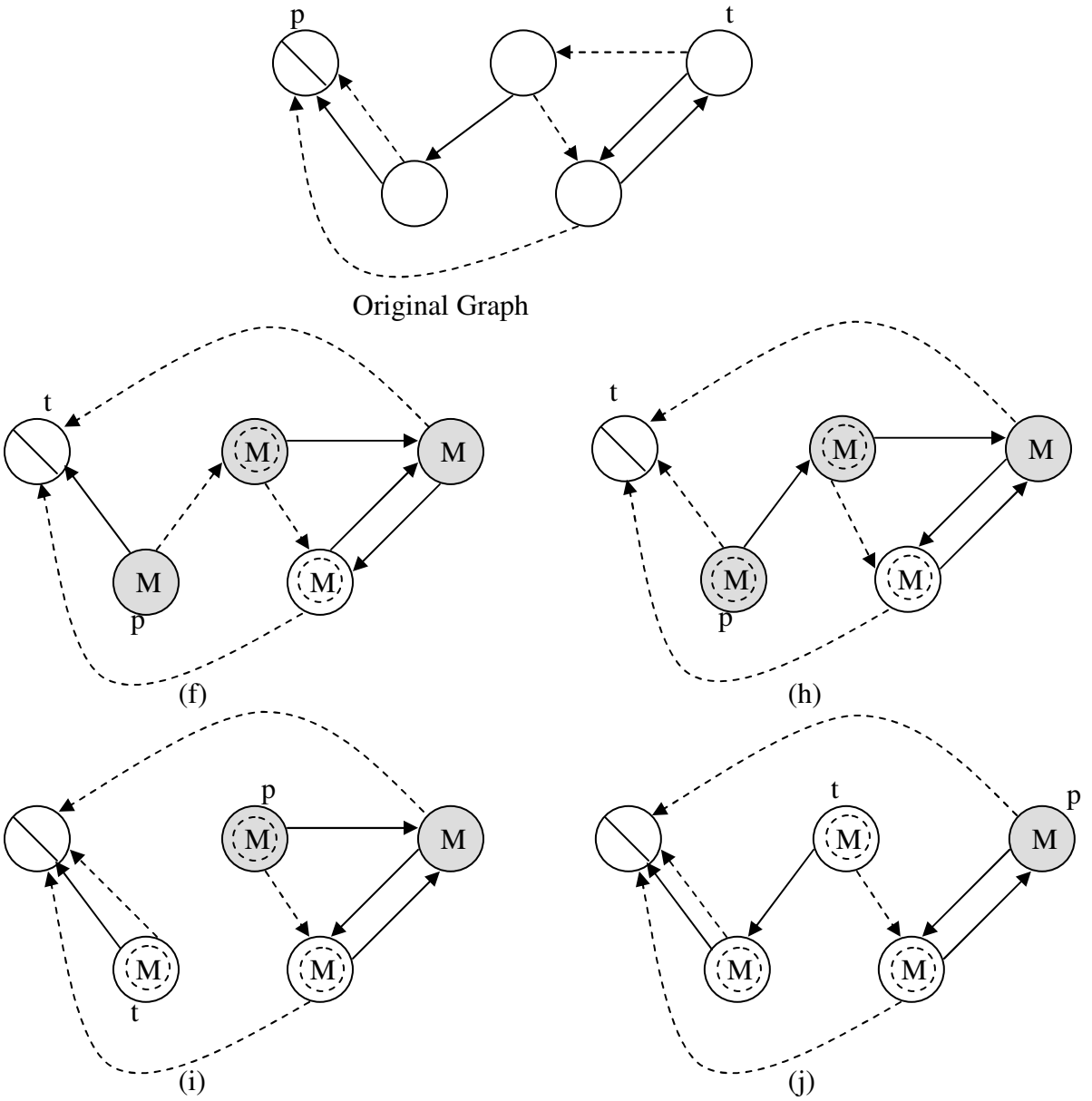


Figure 2.4 Original state and System states at the beginning of iterations eight through eleven

During the next two iterations, the push block is executed followed by the swing block to obtain graphs in (g) and (h). The pop block is executed during the ninth and tenth iterations to obtain graphs in (i) and (j). In graph (g), the left link of *p* now points to its predecessor while the right link remains unchanged. The number of nodes in the list of

predecessor nodes increases by one. The swing block restores the left link of *p* and now its right link points to its predecessor. Execution of both pop blocks result in links of the node removed from the list of predecessor nodes being restored.

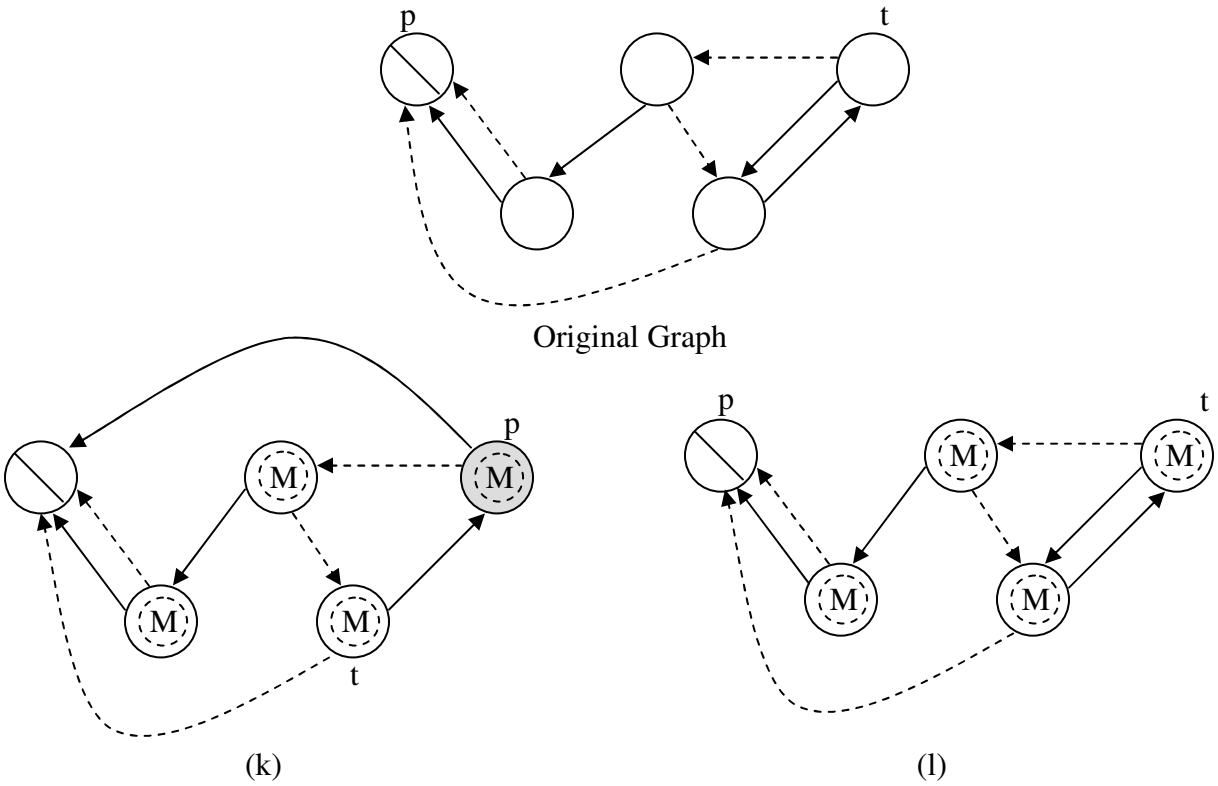


Figure 2.5 Original state and System states at the beginning of twelfth and thirteenth iteration

The swing block is executed in the eleventh iteration to obtain graph in (k). The left link of p is restored and the right link now points to its predecessor. Finally, the pop block is executed and the links of node removed from the list of predecessor nodes are restored. We can see that we have obtained the graph we started out with and all the nodes have been marked.

2.2 Related Work

In general, "the basic idea in all approaches to pointer program proofs is the same and goes back to Burstall [3]: model the heap as a collection of variables of type *address* → *value* and reason about the programs in Hoare logic" [9].

Bornat [1] used the Burstall approach and developed a "component-as-array" trick which essentially broke the heap down into several arrays in such a way that aliasing could not

occur between two arrays. To achieve this, he disallowed the assignment of whole objects. Instead, only parts (fields) of objects can be used in assignment statements. Hence, each array was identified with a field and indexed using the various objects that can access the field. He ignored the question of definedness and his approach could allow indexed arrays to be potentially infinite which made formal proofs difficult and tedious. Also, some of the formulae that one needs to deal with can be too big to handle manually. He used the Jape proof editor which does not provide much automation and the proof is very lengthy.

Another study that proved the correctness of the Schorr-Waite algorithm was presented by Metha and Nipkow [9]. The authors modeled the heap as mappings from addresses to values, similar to Burstall's approach. The higher level data types were used to model pointer structures. The method is claimed to be logically fully sound for the verification of inductively defined data structures. They used the Isabelle/HOL theorem prover to prove the correctness of the Schorr-Waite algorithm. They still had to reason about pointers by using the heap.

Hubert and Marche [4] conducted a case study on C source code verification by taking the Schorr-Waite algorithm as an example. They reused the component-as-array trick from Bornat's paper to model their heap. They added another clause to the proving process called the variant which helped prove termination. In addition to the proof of the correctness of the Schorr-Waite algorithm, they provide facilities which facilitate dereferencing checks. They also are able to prove a subset of post-conditions that can be requested by the user. They developed the CADUCEUS tool to generate the verification conditions.

All the above studies used similar approaches to model the heap and did not discuss components or modular reasoning, which is essential for the scalability of a system. The Resolve group has been conducting research into developing components that support modular reasoning for over twenty years. The correctness of a component implementation is proved using only its specification and specifications from the components it imports. When these components are used in any program, they can then

be assumed correct. This simplifies the verification process for large and complex programs.

Kulczycki et al. present a Reslove pointer component known as the Location_Linking_Template in [5]. The pointer component does not rely on (program-wide) global heap variables. Hence, it does not need any special rules for reasoning; it can be reasoned about as any other component. The implementation and proof of correctness of the Schorr-Waite algorithm presented in this thesis is based largely on the Resolve pointer component. Therefore, the next section briefly reviews the component, its mathematical model, and its operations.

2.3 Overview of Resolve Pointer Component

Informally, the Resolve pointer component describes a system of linked locations. The system is made up of a finite number of locations. Each location is either taken or free and each location holds information and has a fixed number of links to other locations. The kind of information as well as the number of links is determined by the programmer when he creates the system. Each location can be manipulated using a variable of type Position. Any location that can be reached by a position variable is said to be accessible. The system defines a special location called Void which is the default location. Any new position variable is automatically located at the Void location which is perpetually free. Figure 2.6 shows an example system that has one link per location.

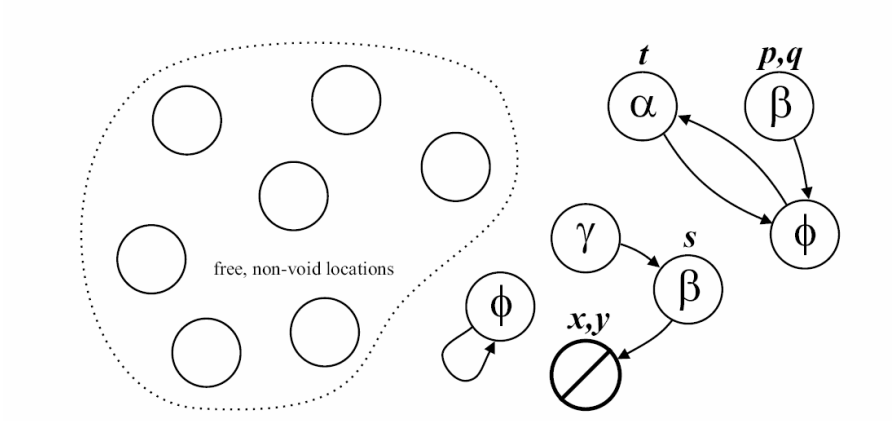


Figure 2.6. An example system with one link per location

In the figure, Greek letters represent the information at each location. The system has six locations that are taken. Position variables are represented by lower case Roman letters. The location where *x* and *y* are located is the Void location. As shown here, cycles may be present in the system and more than one position variable may be located at the same location. A more formal, abbreviated description of the Location_Linking_Template is given in Figure 2.7. The complete specification is provided in [5], and it includes operations for allocation and de-allocation, and lightweight performance specifications.

Concept Location_Linking_Template (**type** Info; **evaluates** k: Integer);

Defines Location: Set;

Defines Void: Location;

Var Target: Location × [1..k] → Location;

Var Contents: Location → Info;

Var Is_Taken: Location → B;

Initialization ensures ∀q: Location, ¬Is_Taken(q);

Type Family Position **is modeled by** Location;

exemplar p;

Initialization ensures p = Void;

Operation Relocate (**updates** p: Position; **preserves** q: Location);

ensures p = q;

Operation Follow_Link (**updates** p: Position; **evaluates** i: Integer);

requires Is_Taken(p) **and** $1 \leq i \leq k$;

ensures p = Target(#p, i);

Operation Redirect_Link (**preserves** p: Position, **evaluates** i: Integer;
preserves q: Position);

updates Target;

requires Is_Taken(p) **and** $1 \leq i \leq k$;

ensures ∀r: Location, ∀j: [1..k],

$$\text{Target}(r, j) = \begin{cases} q & \textbf{if } r = p \textbf{ and } j = i \\ \#\text{Target}(r, j) & \textbf{otherwise}; \end{cases};$$

Operation Relocate_to_Target (**updates** p: Position; **preserves** q: Position,
evaluates i: Integer);

requires Is_Taken(q) **and** $1 \leq i \leq k$;

ensures p = Target(q, i);

Operation Redirect_to_Target (**updates** p: Position; **evaluates** i: Integer;
preserves q: Position; **evaluates** j: Integer);

updates Target;

requires Is_Taken(p) **and** $1 \leq i \leq k$ **and** Is_Taken(q) **and** $1 \leq j \leq k$;

ensures $\forall$r: Location, $\forall$m: [1..k],

$$\text{Target}(r, m) = \begin{cases} \#\text{Target}(q, j) & \textbf{if } r = p \textbf{ and } m = i \\ \#\text{Target}(r, m) & \textbf{otherwise}; \end{cases};$$

Operation Swap_Contents (preserves p: Position; updates I: Info);

updates Contents;

requires Is_Taken(p);

ensures I = #Contents(p) and $\forall$r: Location,

$$\text{Contents}(r) = \begin{cases} \#I & \textbf{if } r = p \\ \#\text{Contents}(r) & \textbf{otherwise}; \end{cases}$$

end Location_Linking_Template;

Figure 2.7 Formal specification of the ResolvePointer Component

The set of locations is modeled as a mathematical set with Void being defined as a specific location in the set. The concept is parameterized by the type of information each location can hold and the precise number of links that can originate from a location. The type of information can be an integer, a character, a tree structure, or any arbitrary type. The concept defines three state or conceptual variables: *Target*, *Contents* and *Is_Taken*. Target(q, i) is the location targeted by the i^{th} link of location *q*, Contents(q) is the information at location *q*, and Is_Taken(q) is true if and only if location *q* is allocated. These variables are used only for specification and reasoning.

The position variables can be manipulated using only the operations given the concept. To better understand how this specification works, let us take a closer look at the

specification of Redirect_Link. The operation takes three parameters: *p*, *i*, and *q*. It also updates the conceptual variable *Target*. All operations in Resolve have an implicit frame property. This property ensures that all variables not mentioned in the parameter list or the updates clause will not be modified. In the case of Redirect_Link, the only variables that can be modified are *p*, *i*, *q* and *Target*; no other variables in the program state are effected by a call to this operation.

Note that all parameters in these operations are associated with a parameter mode. These modes help the client understand what happens to the arguments of a call. The *preserves* mode ensures that the value of the argument does not change. In Redirect_Link, this is true for both *p* and *q*. The *updates* mode indicates that the argument will be modified during the execution of the operation. The *evaluates* mode indicates that it expects an expression as an argument – this mode is commonly used for small types, such as boolean and integers. The *clears* mode ensures that the argument gets an initial value of its type after the call. The *replaces* mode ensures that the value passed in will be discarded and be replaced by a new value.

All operations have a requires and an ensures clause. The requires clause is a precondition – it asserts what must be true about the program state in order for the client to call the operation. The requires clause for Redirect_Link states that *p* must be a *taken* location. The ensures clause is a postcondition – it asserts what will be true about the program state just after the operation is executed. The ensures clause for Redirect_Link says that the i^{th} link of *p* is redirected to *q*, and that this is the only mapping in Target that is modified, i.e. all the other links remain the same.

Figure 2.8 shows how some of the operations defined in the concept manipulate the position variables. The Relocate operation moves *p* to location represented by *q*. The Follow_Link operation moves *p* to the location originally pointed to by *p*'s i^{th} link. The Redirect_Link modifies the conceptual variable Target. It redirects *p*'s i^{th} link to *q*'s location. The Relocate_to_Target operation relocates *p* to the location pointed to by *q*'s ith link. The Redirect_to_Target operation redirects *p*'s i^{th} link to the location pointed to by *q*'s j^{th} link.

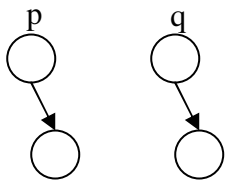
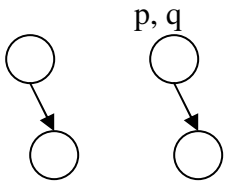
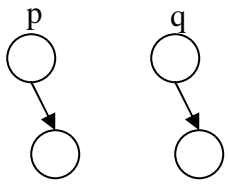
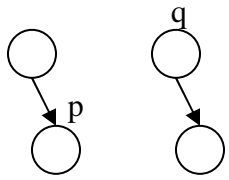
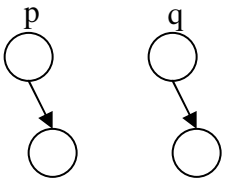
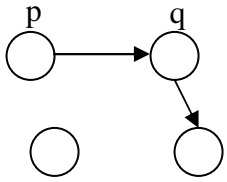
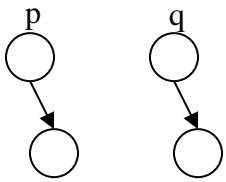
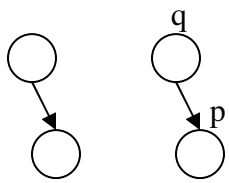
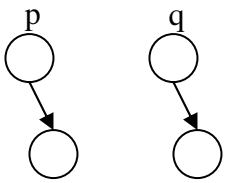
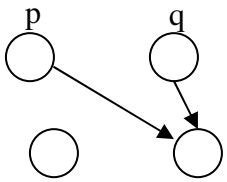
Procedure Call	Before	After
Relocate(p, q)		
Follow_Link(p,1)		
Redirect_Link(p, 1, q)		
Relocate_to_Target(p, q, 1)		
Redirect_to_Target(p, 1, q, 1)		

Figure 2.8 Effect of selected calls on a system

Unlike other components, the pointer component will be implemented by the compiler writer. To achieve the performance benefits of pointers, calls to pointer operations are translated as simple instructions at the machine level. For example, a programmer can reason about Relocate(p, q) based on its specification, but the call will be implemented by the compiler writer as a simple copy instruction.

3. Specification of Schorr-Waite algorithm

In this section we will look into implementing and specifying the Schorr-Waite algorithm using the Resolve pointer component discussed in the previous section. Before we discuss the Schorr-Waite algorithm, we need to introduce an extension known as Bit_Op_Capability, which is given in Figure 3.1. An extension extends a concept in this case the Location_Linking_Template with additional operations. The implementation of an extension has access to both the specification and realization of the concept it is extending.

Extension Bit_Op_Capabality **for** Location_Linking_Template;

Var Is_Marked: Location → **B**;
Var CB_Is_Set: Location → **B**;

Operation Mark_Location(**preserves** p: Position);
updates Is_Marked;
ensures Is_Marked(p);

Operation Unmark_Location(**preserves** p: Position);
updates Is_Marked;
ensures not Is_Marked(p);

Operation Set_Control(**preserves** p: Position);
updates CB_Is_Set;
ensures CB_Is_Set (p);

Operation Unset_Control (**preserves** p: Position);
updates CB_Is_Set;
ensures not CB_Is_Set (p);

end Bit_Op_Capability;

Figure 3.1 Formal Specification of Bit_Op_Capability Extension

The Bit_Op_Capability extension consists of two conceptual variables: *Is_Marked* and *CB_Is_Set*. These are mappings from locations to boolean values. These can be implemented as single bits associated with each location. The *Is_Marked* variable

signifies whether a location is marked or not. The CB_Is_Set variable tells whether if the control bit has been set for that location. The extension also provides operations which manipulate these variables. The Mark_Location and Unmark_Location operations change the Is_Marked variable, and the Set_Control and Unset_Control operations change the CB_Is_Set variable. With this extension, we now have enough information to discuss the specification and implementation of the Schorr-Waite algorithm.

3.1 The Specification of Mark_Accessible_From

In this section we introduce an enhancement to the concept Location_Linking_Template known as Mark_Capability. An enhancement is also the addition of features to an already developed concept. However, a realization of an enhancement is able to view just the specification of the concept. The Mark_Capability enhancement extends the pointer component with a new operation: Mark_Accessible_From. The operation Mark_Accessible_From will be implemented using the Schorr-Waite algorithm. The enhancement includes some basic definitions that are helpful in specifying the operation. The specification for the Mark_Capability enhancement is given in figure 3.2.

Enhancement Mark_Capability **for** Location_Linking_Template

extended by Bit_Op_Capability;

Definition Is_Path(ρ: Str(Locations); p, q: Location,
targ: Location × (LEFT, RIGHT) → Location): **B =**
p Is_Prefix ρ **and** q Is_Suffix ρ **and** ∀u, v: Location,
if ⟨ u ⟩ o ⟨ v ⟩ Is_Substring ρ **then** ∃j: [LEFT, RIGHT] targ(u, j) = v;

Definition Is_Reachable(p, q: Location,
targ: Location × (LEFT, RIGHT) → Location): **B** =
∃ ρ: Str(Locations), Is_Path(ρ, p, q, targ);

Operation Mark_Accessible_From(**preserves** root: Position);
updates Is_Marked;
alters CB_Is_Set;
restores Target;
requires ∀x: Location, **if** Is_Reachable(x, root, Target) **then not** Is_Marked(x);

ensures ∀q: Location,

$$\text{Is\_Marked(q)} = \begin{cases} \textbf{true} & \text{Is\_Reachable(q, root);} \\ \#\text{Is\_Marked(q)} & \text{otherwise;} \end{cases}$$

end Mark_Capability;

Figure 3.2 Formal Specification of Mark_Capability Enhancement

The enhancement defines Is_Path and Is_Reachable. Is_Path (ρ , p , q, targ) is true if ρ is a path between *p* and *q* under the mapping *targ*. The *targ* mapping is a specific instance of the *Target* variable in the pointer component. It indicates which links point to which locations. Obviously, if there is a path between two locations for one value of *Target*, it does not mean that the path will still be there for another value of *Target*. The definition Is_Reachable establishes reachability of one location from another. Is_Reachable (p, q, targ) is true if *q* is reachable from *p* under the mapping *targ*.

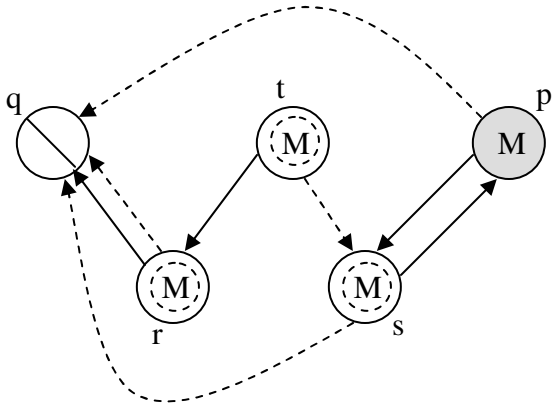


Figure 3.3 Intermediate system state during the execution of Schorr-Waite algorithms

Consider the system of linked locations in Figure 3.3. Let *targ* be the *Target* variable that defined the links in the figure. Then Is_Path({t,r,q}, t, q, targ) = **true** becuase there is a link from *t* to *r* and a link from *r* to *q* under the mapping *targ*. Also, Is_reachable(t, q, targ) = **true** since we know that there is path from *t* to *q*. However, Is_Path({p,s,r,q}, p, q, targ) = **false** since no link exists from *s* to *r* under the current mapping. Is_Reachable(p, t, targ) = **false** as no path exists between *p* and *t*.

Using the definitons we can now easily specify the operation Mark_Accessible_From, which takes a Position variable as a parameter. It requires that if a location is reachable

from *root*, then the location is not marked. It ensures that after a call, all locations originally reachable from *root* are marked. The next two sections give implementations for Mark_Accessible_From using the Schorr-Waite algorithm.

3.2 Traditional implementation

The traditional realization for the enhancement above uses the operations defined by the Location_Linking_Template. It is based on the code described in [Mehta] and [Marche]. The realization consists of both programming statements as well as mathematical assertions. The mathematical assertions are only used for program verification and are not actually executed when the program is run. There are some definitions added to the realization that are not present in the enhancement. These definitions help to specify mathematical assertions such as the loop invariant. The complete realization is provided in Figure 3.4.

Realization Trad_Realization **for** Mark_Capability;

Definition Is_Unmarked_Reachable(x: Location, y: Location,
targ: Location × (LEFT, RIGHT) → Location): **B** =
∃ ρ: Str(Locations) **such that** Is_Path(ρ, p, q, targ) **and**
∀u, Locations, **if** ⟨u⟩ Is_Substring(ρ) **then not** Is_Marked(u);

Implicit Definition Loc_Stack (z: Location,
targ: Location × (LEFT, RIGHT) → Location, CB: Location → **B**): Str(Location)

$$= \begin{cases} \langle\rangle & \textbf{if } z = \text{void} \\ \langle z\rangle \circ \text{Loc\_Stack}(\text{targ}(z, \text{LEFT}), \text{targ}, \text{CB}) & \textbf{if } z = \text{void } \textbf{and } \text{CB}(z) = \text{false} \\ \langle z\rangle \circ \text{Loc\_Stack}(\text{targ}(z, \text{RIGHT}), \text{targ}, \text{CB}) & \textbf{if } z = \text{void } \textbf{and } \text{CB}(z) = \text{true} \end{cases}$$

Implicit Definition StackOK(q: Location, s: String(Location),
targ: Location × (LEFT, RIGHT) → Location, CB: Location → **B**): **B is**
(i) Stack_OK(q, ⟨ ⟩, targ, CB) = **true**;
(ii) Stack_OK(q, ⟨p⟩ o s, targ, CB) = (StackOK(p, s, targ. CB) **and**
TargetOK(p, q, targ, CB));

Definition TargetOK(p: Location, q: Location,

targ: Location × (LEFT, RIGHT) → Location, CB: Location → **B**): **B =**

$$\#\text{Target}(p, \text{LEFT}) = \begin{cases} \text{Target}(p, \text{LEFT}) & \textbf{if } \text{CB\_Is\_Set}(p) \\ q & \textbf{otherwise} \end{cases} \textbf{ and}$$

$$\#\text{Target}(p, \text{RIGHT}) = \begin{cases} q & \textbf{if } \text{CB\_Is\_Set}(p) \\ \text{Target}(p, \text{RIGHT}) & \textbf{otherwise} \end{cases};$$

Definition Unset_CB_Count = || { x: Location |

⟨x⟩ Is_Substring Loc_Stack(p, Target, CB_Is_Set) **and not** CB_Is_Set(x) } ||

Definition Unmarked_Loc_Count = || { x: Location |

Is_Reachable(root, x, #Target) **and not** Is_Marked(x) } ||

Procedure Mark_Accessible_From(**preserves** root: Position);

Var t, p, q: Position;

Relocate(t, root);

While not Is_At_Void(p) **or** (**not** Is_At_Void(t) **and not** Is_Marked(t))

maintaining ∀x: Location, (

(i1) (Is_Reachable(x, root, #Target) **iff**

(Is_Reachable(x, t, Target) **or** Is_Reachable(x, p, Target))) **and**

(i2) (**if not** Is_Reachable(x, root, #Target) **then**

Is_Marked(x) = #Is_Marked(x)) **and**

(i3) (**if** ⟨x⟩ Is_Substring Loc_Stack(p, Target, CB_Is_Set) **then**

Is_Marked(x)) **and**

(i4) (**if** Is_Reachable(x, root, #Target) **and not** Is_Marked(x) **then**

(Is_Unmarked_Reachable(x, t, Target) **or**

∃y: Location **such that**

⟨y⟩ Is_Substring Loc_Stack(p, Target, CB_Is_Set) **and**

Is_Unmarked_Reachable(x, Target(y, RIGHT), Target))) **and**

(i5) (**if not** ⟨x⟩ Is_Substring Loc_Stack(p, Target, CB_Is_Set) **then**

Target(x, RIGHT) = #Target(x, RIGHT) **and**

Target(x, LEFT) = #Target(x, LEFT)) **and**

(i6) Stack_OK(t, Loc_Stack(p, Target, CB_Is_Set), Target, CB_Is_Set));

```
      decreasing     3 * Unmarked_Loc_Count + Unset_CB_Count +
                              |Loc_Stack(root)|;
    do
      If Is_At_Void(t) or Is_Marked(t) then
        If Control_Is_Set(p) then
          Relocate(q, t);
          Relocate(t, p);
          Follow_Link(p, RIGHT);
          Redirect_Link(t, RIGHT, q);
        else
          Relocate(q, t);
          Relocate_to_Target(t, p, RIGHT);
          Redirect_to_Target(p, RIGHT, p, LEFT);
          Redirect_Link(p, LEFT, q);
          Set_Control(p);
        end;
      else
        Relocate(q, p);
        Relocate(p, t);
        Follow_Link(t, LEFT);
        Redirect_Link(p, LEFT, q);
        Mark_Location(p);
        Unset_Control(p);
      end;
    end;
  end Mark_Accessible_From;
end Trad_Realization;
```

Figure 3.4 Traditional implementation for Mark_Accessible_From

The realization provided for the enhancement implements the Mark_Accessible_From operation using the Schorr-Waite graph marking algorithm. It uses the operations defined

in the Bit_Op_Capability extension. Some terms are defined to help write the loop invariant. Is_Unmarked_Reachable(q, p, targ) is true if *q* is reachable from *p* by traversing locations that are not marked under the mapping *targ*. For a location to be unmarked reachable, there must a path between the two locations under the mapping specified and all the locations on the path must not be marked.

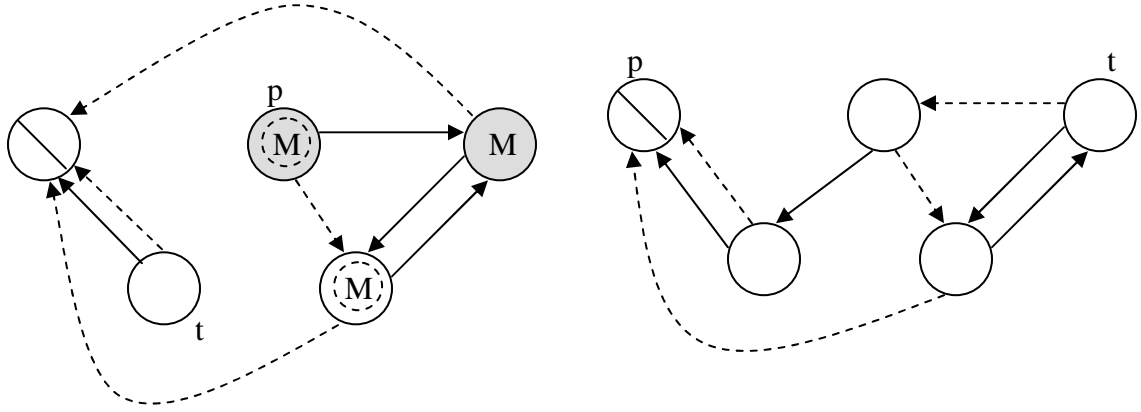
Since the Schorr-Waite algorithm does not use a stack, it needs to modify the existing pointers to keep track of the predecessors of various nodes that it passes through. The definition Loc_Stack defines a conceptual stack that can be obtained by traversing these pointers. The Loc_Stack is a string of all locations through which we need to backtrack to able to mark the graph completely. All the locations on this stack have some of their links modified. To obtain this string of locations, we need to know which link points to the predecessor (LEFT or RIGHT). To know this, we have a control bit (which is defined in the Bit_Op_Capability extension) for each location, which tells us which link to follow. This is defined formally using the implicit definition Loc_Stack which defines the Loc_Stack recursively. Clearly, the Loc_Stack starting from the Void location is empty. The Loc_Stack from any other location is defined as that location concatenated with the Loc_Stack from the location pointed to by its LEFT or RIGHT link depending on whether its control bit is set or not.

The definition Stack_OK helps us to determine whether we can restore all of the original links for each location on the Loc_Stack. This is important since at the end of the algorithm all links must be restored to return the original graph. Stack_OK is true for an empty string. For the non-trivial case where the Loc_Stack is not empty, we iterate through the stack and check if the original LEFT and RIGHT links can be obtained through the information present. The definition Target_OK defines what the original LEFT and RIGHT links point to depending on whether the CB_Is_Set for the Location. This definition is called for all locations on the Loc_Stack.

A mathematical assertion that helps us prove the correctness of the Schorr-Waite algorithm is the loop invariant. An invariant must be is true at the beginning and end of any iteration of the loop. In addition, it must be string enough to allow you to prove the ensures clause of the operation. The invariant clause for the Schorr-Waite is very

complex, so we break it down in several conjuncts. The first conjunct states that anything that was initially reachable from the root is now reachable through *p* and *t*. The second conjunct states that the marked status for any location that was not initially reachable from the root remains unchanged. The third conjunct states that all locations on the location stack are marked. The fourth conjunct states that any location that was initially reachable from the root and is not marked is now unmarked-reachable either from *t* or a location pointed to by the right link of a location on the stack. The fifth conjunct states that for all locations not on the stack, the links are unchanged. The final conjunct states that Stack_OK is true, i.e. for all locations on the location stack, the original links can be restored.

To illustrate how the invariant works for a given state in the Schorr-Waite algorithm, consider Figure 3.5.



(a) Intermediate system state (b) Original state

Figure 3.5 Intermediate system state and original system state for the Schorr-Waite algorithm

The graph on the left shows the current state while the graph on the right represents the original graph. In the original graph, all the locations shown are reachable from the root. In the current graph, all the locations shown are reachable either from *p* or *t*. Therefore the first conjunct of the invariant is true. The second conjunct is also true since there are no locations that are originally reachable from the root. The location stack consists of all the locations that are grayed. Clearly, *p* is on the location stack. Since the control bit is set for *p*, we need to follow its right link to get the next node and so on. We can see that all

locations on the location stack are marked. This satisfies the third conjunct. There is only one location that is unmarked and it is unmarked reachable from *t*. Hence the fourth conjunct is satisfied. We can see that for all locations not on the location stack, all the links are the same as that in the original graph. This makes the fifth conjunct true. The final conjunct states that for all locations on the location stack, the original links can be restored. For *p*, the right link is the same as the one in the original graph. The left link can be restored by redirecting it to *t*. For the second location on the location stack, the left link is the same as the original graph and its right link can be restored by redirecting it to *p*. Hence, Stack_OK is true for the location stack.

As noted early, the loop body of the Schorr-Waite algorithm can be broken down into three parts depending on the status of *p* and *t*. If the control bit is set for *p* and *t* is located at void or *t* is marked, then the pop block is executed. If the control bit is not set for *p* and *t* is located at void or *t* is marked, then the swing block is executed. In all other cases, the push block is executed.

3.3 Accessibility-Preserving Implemention

The Schorr-Waite algorithm does not use a stack to perform garbage collection. It makes use of the pointers of locations to keep track of the predecessors. However, once the algorithm is finished, all links are restored to back to their original states. Though the links are changed during the algorithm, the accessibility of the system should not change as it would defeat the purpose of garbage collection itself. This is ensured by the first conjunct in the invariant. Proving the first conjunct using the traditional realization is tedious and complex. We introduce a new realization called Acc_Preserv_Realization that helps us prove the first conjunct in an easier fashion. This realization uses some new operations which maintain the accessibility of the system. These operations are called accessibility-preserving operations.

Before we introduce the accessibility preserving operations, we note that the operations used in the traditional realization do not always preserve accessibility. This is true of Relocate, Redirect_Link and Follow_Link. Figure 3.6 shows how a call Relocate(p, q) can cause a memory leak. The location originally represented by p is inaccessible after the call to relocate. The call Follow_Link(p, NEXT) will have the same result. Figure xxx

shows how the operation Redirect_Link(p, NEXT, q) can create a memory leak in the system. The location originally pointed by p's NEXT link becomes inaccessible.

Figure 3.6 Effect of Relocate(p, q) or Follow_Link(p, NEXT)

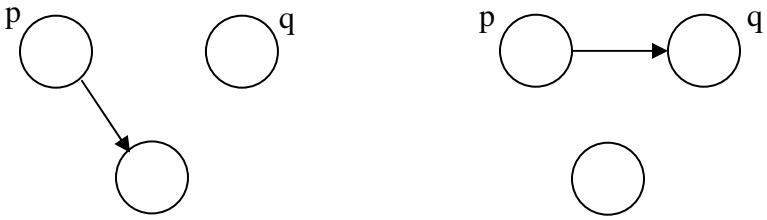


Figure 3.7 Effect of Redirect_Link(p, NEXT, q)

Let us now introduce three operations which preserve the accessibility of the system. The simplest of these operations is the swap operation, p :=: q, which just swaps the two pointers. It is easy to see that this operation preserves the accessibility of the system. Any location that was initially accessible from *p* (including *p* itself) is now accessible from *q* and vice versa. Another accessibility-preserving operation swaps any two links. These links may originate from different locations or the same location. The operation is called as Swap_Links(p, m, q, n). It redirects *p*'s m^{th} link to the location originally pointed to by *q*'s n^{th} link, and vice versa. Accessibility is maintained in this case because all locations originally accessible from *q*'s n^{th} link are now accessible from *p*'s m^{th} link and all locations originally accessible from *p*'s m^{th} link are now accessible from *q*'s n^{th} link. Figure 3.8 shows how the operation modifies the system and maintains accessibility.

Figure 3.8 Effect of Swap_Links(p, NEXT, q, NEXT)

The formal specification of the Swap_Links operation is given below.

Operation Swap_Links (**preserves** p: Location; **evaluates** m: Integer;
preserves q: Location; **evaluates** n: Integer);

updates Target;

requires Is_Taken(p) and Is_Taken(q) and $1 \leq m \leq k$ and $1 \leq n \leq k$;

ensures $\forall$x: Location, $\forall$j: [1..k],

$$\text{Target}(x, j) = \begin{cases} q & \textbf{if } x = p \textbf{ and } j = m \\ p & \textbf{if } x = q \textbf{ and } j = n \\ \text{Target}(x, j) & \textbf{otherwise}; \end{cases}$$

Both operations above either swap a pointer with another pointer or a link with another link. The final operation we are going to look at swaps a pointer with a link. It is known as Swap_Link_Pos. Swap_Link_Pos(p, NEXT, q) redirects the NEXT link of *p* to *q*'s original location and *q* is relocated to the location originally pointed to by *p*'s NEXT link. Figure 3.9 shows how the operation modifies the system and maintains accessibility.

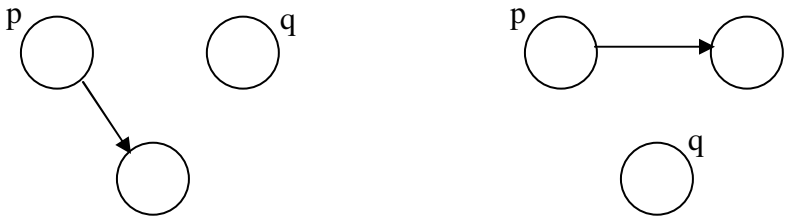


Figure 3.9 Effect of Swap_Link_Loc(p, NEXT, q)

The formal specification of the Swap_Link_Loc operation is given below.

Operation Swap_Link_Loc (**preserves** p: Location; **evaluates** m: Integer;
updates q: Location);

updates Target;

requires Is_Taken(p) $1 \leq m \leq k$;

ensures $\forall$x: Location, $\forall$j: [1..k],

$$\text{Target}(x, j) = \begin{cases} q & \textbf{if } x = p \textbf{ and } j = m \\ \text{Target}(x, j) & \textbf{otherwise}; \end{cases} \textbf{ and}$$

q = #Target(p, m);

We are proposing that these operations be added to the Resolve pointer component as primary operations. These operations can simplify proofs in which accessibility needs to

be preserved. We looked at the use of these operations for other components – such as lists – in [11] and found that they simplified reasoning in cases where accessibility needs to be preserved.

Using these operations, we can implement the Schorr-Waite algorithm in such a way that accessibility is preserved at each step in the process. This will make the first conjunct of the loop invariant easier to prove. In addition it has the fortunate side-effect of reducing the code size and allowing us to eliminate a local variable (q in the traditional realization). The accessibility preserving realization is presented in Figure 3.10. The definitions, loop invariants and decreasing metric do not change from the traditional realization.

```
Realization Acc_Preserv_Realization for Mark_Capability;
  …
  Procedure Mark_Accessible_From(preserves root: Position);
    Var t, p: Position;
    Relocate(t, root);
    While not Is_At_Void(p) or ( not Is_At_Void(t) and not Is_Marked(t) )
      maintaining …
      decreasing …
    do
      If Is_At_Void(t) or Is_Marked(t) then
        If Control_Is_Set(p) then
          Swap_Link_Loc(p, RIGHT, t);
          p :=: t;
        else
          Swap_Links(p, LEFT, p, RIGHT);
          Swap_Link_Loc(p, LEFT, t);
          Set_Control(p);
        end;
      else
        p :=: t;
        Swap_Link_Loc(p, LEFT, t);
```

```
            Mark_Location(p);
            Unset_Control(p);
          end;
        end;
    end Mark_Accessible_From;
end Acc_Preserv_Reazlization.
```

Figure 3.10 Accessibility-preserving implementation for Mark_Accessible_From

4. Proof of Schorr-Waite

We now take a look at the proof for correctness of the Mark_Accessible_From function. We can see that the entire function consists of the while loop with the statement t := root before the while loop. Therefore, the proof of correctness for the procedure amounts to a proof that the loop invariant is correct. For this, we need to prove initialization, termination, and maintenance for the invariant. The proof for initialization shows that the invariant is true at the beginning of the first iteration. The proof for termination shows that given the invariant and the negation of the while condition is true at the end of the last iteration, the ensures clause for the operation is true. The maintenance clause shows that given the invariant is true at the beginning of an arbitrary iteration, it is true at the end of the iteration. We now look at each of these proofs in detail.

4.1 Proof for Initialization

To prove the initialization condition for the invariant, we must show that the invariant is true at the beginning of the first loop iteration. At the beginning of the procedure, we may assume that the requires clause is true. Therefore, we know that:

∀x: Location, **if** Is_Reachable(x, root, Target) **then not** Is_Marked(x);

Declarations of local position variables *t* and *p* put both variables at the Void location. Immediately after these declarations, *t* is relocated to *root*'s location. Therefore, just before the loop begins, we also know that:

t = root **and** p = Void

None of the statements before the loop modify the Target variable, so we also know that:

Target = #Target

Given these facts, we should be able to prove that the following invariant is true:

∀x: Location,

(i1) (Is_Reachable(x, root, #Target) **iff** (Is_Reachable(x, t, Target) **or** Is_Reachable(x, p, Target))) **and**

(i2) (**if not** Is_Reachable(x, root, #Target) **then** Is_Marked(x) = #Is_Marked(x)) **and**

(i3) (**if** ⟨x⟩ Is_Substring Loc_Stack(p, Target, CB_Is_Set)) **then** Is_Marked(x)) **and**

(i4) (**if** Is_Reachable(x, root, #Target) **and not** Is_Marked(x) **then**
(Is_Unmarked_Reachable(x, t, Target) **or**
∃y: Location **such that**
⟨y⟩ Is_Substring Loc_Stack(p, Target, CB_Is_Set) **and**
Is_Unmarked_Reachable(x, Target(y, RIGHT), Target))) **and**

(i5) (**if not** ⟨x⟩ Is_Substring Loc_Stack(p, Target, CB_Is_Set) **then**
Target(x, RIGHT) = #Target(x, RIGHT) **and**
Target(x, LEFT) = #Target(x, LEFT)) **and**

(i6) Stack_OK(t, Loc_Stack(p, Target, CB_Is_Set), Target, CB_Is_Set)

We prove the invariant by showing that each conjunct is true.

(i1) *Locations reachable from the root should be reachable from t or p.*

We know that *t* = *root* and Target = #Target. Substituting *t* for *root* and Target for #Target in conjunct (i1) yields:

Is_Reachable(x, root, Target) **iff** (Is_Reachable(x, root, Target) **or**
Is_Reachable(x, p, Target))

which is clearly true.

(i2) *The "marked" status of locations not reachable from the root should not change.*

No locations have been marked or unmarked yet, so we know that Is_Marked(x) = #Is_Marked(x). The consequent in (i2) is true, therefore (i2) must be true.

(i3) *All locations on the location stack should be marked.*

We know that p = Void. By the definition of Loc_Stack, we know that Loc_Stack(Void, Target, CB_Is_Set) is the empty string for any values of Target and CB_Is_Set. Hence, there is no location *x* that satisfies the antecedent of (i3). Therefore, (i3) is true.

(i4) *A location should either be unmarked-reachable from t or from the right link of a location on the location stack.*

Assertion (i4) has the form A ⇒ (B ∨ C). Therefore, it suffices to show that A ⇒ B. In this case, A ≡ "**if** Is_Reachable(x, root, #Target) **and not** Is_Marked(x)," which is true because it follows from the requires clause. Therefore, we must prove that B ≡ "Is_Unmarked_Reachable(x, t, Target)" is true for all relevant *x*. From the requires clause, we know that all locations reachable from *root* (which is the same as *t*) are not marked. Furthermore, no locations are marked before entering the loop. Therefore, all locations originally reachable from *root* are also reachable from *t* through an unmarked path. Thus, (i4) is true.

(i5) *All locations that are not part of the location stack should have their original links.*

We know that #Target = Target, making the consequent true, therefore the entire implication must be true.

(i6) *Locations on the stack should be able to have their original links restored.*

As demonstrated above in the proof for (i3), *p* = Void implies that Loc_Stack(p, Target, CB_Is_Set) is the empty string. By the definition of Stack_OK, when Loc_Stack is empty, Stack_OK is true.

4.2. Proof for Termination

To prove the termination condition for the loop, we need to show that the loop invariant is strong enough to prove what we want to prove. In this case, we must show that the ensures clause for the procedure is true at the end of the loop. At the end of the loop, we know that invariant is true and the negation of the while condition given below is true.

p = Void **and** (t = Void **or** Is_Marked(t))

Given these facts, we should be able to prove that the following ensures clause is true:

Target = #Target **and**

∀q: Location, Is_Marked(q) =

$$\begin{cases} \textbf{true} & \textbf{if } \text{Is\_Reachable}(q, root, \#Target); \\ \#\text{Is\_Marked}(q) & \textbf{otherwise}; \end{cases}$$

From (i5), we know that

∀x: Location, **if not** x Is_Substring Loc_Stack(p, Target, CB_Is_Set) **then**
(Target(x, RIGHT) = #Target(x, RIGHT) **and**
Target(x, LEFT) = #Target(x, LEFT))

In other words, all locations not on the Loc_Stack have their links unchanged.

We know *p* = Void. Therefore, by the definition of Loc_Stack, we know Loc_Stack is empty. Hence, all locations have their links unchanged, i.e. Target = #Target.

When we take a look at the second conjunct of the ensures clause, we see that it can be further broken down into two conjuncts,

∀q: Location,
if not Is_Reachable(q, root, #Target) **then** Is_Marked(q) = #Is_Marked(q) **and**
if Is_Reachable(q, root, #Target) then Is_Marked(q) = **true**

The first conjunct directly follows from conjunct (i2) of the invaraint which states that:

∀q: Location,
if not Is_Reachable(q, root, #Target) **then** Is_Marked(q) = #Is_Marked(q).

Therefore, it suffices to prove that

if Is_Reachable(q, root, #Target) then Is_Marked(q) = **true**.

We know that *p* = Void and we have proved that Target = #Target. Substituting these values in conjunct (i1) of the invariant we get

∀x: Location, Is_Reachable(x, root, #Target) **iff** (Is_Reachable(x, t, #Target) **or**
Is_Reachable(x, void, #Target)

From this we get, ∀x: Location, Is_Reachable(x, root, #Target) **iff** (Is_Reachable(x, t, #Target). If *root* = Void, we know that nothing is reachable from the *root* and the conjunct is true. If root ≠ Void, we can infer *t* ≠ Void. From the negation of the while condition we know *t* = void **or** Is_Marked(t), therefore Is_Marked(t) = true.

We know *p* = void, so Loc_Stack is empty. Hence conjunct (i4) becomes

∀x: Location **, if** Is_Reachable(x, root, #Target) **and not** Is_Marked(x) t**hen** Is_Unmarked_Reachable(x, t, Target).

Using the fact we proved earlier, ∀x: Location, Is_Reachable(x, root, #Target) **iff** (Is_Reachable(x, t, #Target), we can substitute Is_Reachable(x, root, #Target) with Is_Reachable(x, t, Target). Thus (i4) becomes

∀x: Location**, if** Is_Reachable(x, t, #Target) **and not** Is_Marked(x) t**hen** Is_Unmarked_Reachable(x, t, Target)

By the contrapositive rule, we get,

∀x: Location**, if not** Is_Unmarked_Reachable(x, t, Target) **then not** Is_Reachable(x, t, Target) **or** Is_Marked(x).

Since *t* is marked, there is no location *x*, for which Is_Unmarked_Reachable(x, t, Target) is **true**. Also, we know that Is_Reachable(x, t, Target) is **true** for all *x*. Hence Is_Marked(x) must be **true**.

4.3. Proof for Maintenance

To prove maintenance we assume that the invariant is true at the beginning of an arbitrary iteration and show that it is still true at the end of the iteration. In general, we should be able to prove that $\text{invariant}_b \Rightarrow \text{invariant}_e$, where invariant_b is the invariant with *p*, *t*, *Target*, *CB_Is_Set*, and *Is_Marked* replaced by p_b, t_b, $Target_b$, $CB\_Is\_Set_b$, and Is_Marked_b. Thus, invariant_b is the following assertion.

∀x: Location,

($i1_b$) (Is_Reachable(x, root, #Target) **iff** (Is_Reachable(x, t_b, Target_b) **or** Is_Reachable(x, p_b, Target_b))) **and**

($i2_b$) (**if not** Is_Reachable(x, root, #Target) **then** Is_Marked_b (x) = #Is_Marked(x)) **and**

($i3_b$) (**if** ⟨x⟩ Is_Substring Loc_Stack(p_b, Target_b, CB_Is_Set_b)) **then** Is_Marked_b(x)) **and**

($i4_b$) (**if** Is_Reachable(x, root, #Target) **and not** Is_Marked_b(x) **then** (Is_Unmarked_Reachable(x, t_b, Target_b) **or**

∃y: Location **such that**

⟨y⟩ Is_Substring Loc_Stack(p_b, $Target_b$, $CB\_Is\_Set_b$) **and**

Is_Unmarked_Reachable(x, $Target_b$(y, RIGHT), $Target_b$)) **and**

($i5_b$) (**if not** ⟨x⟩ Is_Substring Loc_Stack(p_b, $Target_b$, $CB\_Is\_Set_b$) **then**

Target(x, RIGHT) = #Target(x, RIGHT) **and**

Target(x, LEFT) = #Target(x, LEFT)) **and**

($i6_b$) Stack_OK(t_b, Loc_Stack(p_b, $Target_b$, $CB\_Is\_Set_b$), $Target_b$, $CB\_Is\_Set_b$)

Likewise, $invariant_e$ is the invariant with *p*, *t*, *Target*, *CB_Is_Set*, and *Is_Marked* replaced by p_e, t_e, $Target_e$, $CB\_Is\_Set_e$, and Is_Marked_e. Thus, $invariant_e$ is the following assertion.

∀x: Location,

($i1_e$) (Is_Reachable(x, root, #Target) **iff** (Is_Reachable(x, t_e, $Target_e$) **or**

Is_Reachable(x, p_e, $Target_e$))) **and**

($i2_e$) (**if not** Is_Reachable(x, root, #Target) **then** Is_Marked_e (x) = #Is_Marked(x)) **and**

($i3_e$) (**if** ⟨x⟩ Is_Substring Loc_Stack(p_e, $Target_e$, $CB\_Is\_Set_e$)) **then** Is_Marked_e (x))

and

($i4_e$) (**if** Is_Reachable(x, root, #Target) **and not** Is_Marked_e (x) **then**

(Is_Unmarked_Reachable(x, t_e, $Target_e$) **or**

∃y: Location **such that**

⟨y⟩ Is_Substring Loc_Stack(p_e, $Target_e$, $CB\_Is\_Set_e$) **and**

Is_Unmarked_Reachable(x_e, $Target_e$ (y, RIGHT), $Target_e$))) **and**

($i5_e$) (**if not** ⟨x⟩ Is_Substring Loc_Stack(p_e, $Target_e$, $CB\_Is\_Set_e$) **then**

Target(x, RIGHT) = #Target(x, RIGHT) **and**

Target(x, LEFT) = #Target(x, LEFT)) **and**

($i6_e$) Stack_OK(t_e, Loc_Stack(p_e, $Target_e$, $CB\_Is\_Set_e$), $Target_e$, $CB\_Is\_Set_e$)

Figure 4.1 shows the structure of the while loop. The loop has three blocks that are governed by various conditions: one for Pop, one for Swing, and one for Push. For any

given iteration of the loop, only one of the three blocks of code will be executed. Hence, we can break up the maintenance proof into these three distinct cases.

```
While not Is_At_Void(p) or ( not Is_At_Void(t) and not Is_Marked(t) ) do
   If Is_At_Void(t) or Is_Marked(t) then
      If Control_Is_Set(p) then
         (* Pop *)
      else
         (* Swing *)
      end;
   else
      (* Push *)
   end;
end;
```

Figure 4.1 Structure of the while loop in the implementation of Mark_Accessible_From

4.3.1 Case 1 – Pop

The Pop block is executed when the following condition holds.

```
CB_Is_Set(p) and (t = Void or Is_Marked(t))
```

The following code is executed.

```
Swap_Link_Loc(p, RIGHT, t);
p :=: t;
```

Based on the specification, the only variables potentially modified by Swap_Link_Loc are *t* and Target. The only variables modified by the swap statement are *p* and *t*. Therefore, the only variables potentially modified by this block are *p*, *t*, and *Target*. Note that *CB_Is_Set* and *Is_Marked* do not change. Let t_b, p_b, $Target_b$, Is_Marked_b, and $CB\_Is\_Set_b$ represent the values of these variables at the beginning of an arbitrary

iteration and t_e, p_e, $Target_e$, $CB\_Is\_Set_e$ and Is_Marked_e represent the values of these variables at the end of the iteration. After the call to Swap_Link_Loc, we know from the ensures clause that:

$$\forall q: \text{Location}, \forall j: \{\text{LEFT}, \text{RIGHT}\},$$

$$\text{Target}(q, j) = \begin{cases} t_b & \textbf{if } q = p \textbf{ and } j = \text{RIGHT} \\ \text{Target}_b(q, j) & \textbf{otherwise}; \end{cases} \textbf{ and}$$

$$t = \text{Target}_b(p, \text{RIGHT}) \textbf{ and } p = p_b$$

Where *p*, *t*, and *Target* represent the current values of those variables. To obtain the final state (after swapping *p* with *t*), we replace *p* with t_e and *t* with p_e in the assertion above. Also, *Target* does not change, so we can also replace *Target* with $Target_e$. Finally, we know that *CB_Is_Set* and *Is_Marked* do not change, so we end up with the following assertion.

$$\forall q: \text{Location}, \forall j: \{\text{LEFT}, \text{RIGHT}\},$$

$$\text{Target}_e(q, j) = \begin{cases} t_b & \textbf{if } q = t_e \textbf{ and } j = \text{RIGHT} \\ \text{Target}_b(q, j) & \textbf{otherwise}; \end{cases} \textbf{ and}$$

$$p_e = \text{Target}_b(p_b, \text{RIGHT}) \textbf{ and } t_e = p_b \textbf{ and}$$

$$\text{Is\_Marked}_e(q) = \text{Is\_Marked}_b(q) \textbf{ and}$$

$$\text{CB\_Is\_Set}_e(q) = \text{CB\_Is\_Set}_b(q)$$

The path condition for Pop allows us to use the following assertion.

$$\text{CB\_Is\_Set}_b(p_b) \textbf{ and } (t_b = \text{Void} \textbf{ or } \text{Is\_Marked}_b(t_b))$$

Given these facts, and assuming $invariant_b$, we prove $invariant_e$ by showing that each conjunct ($i1_e$ – $i6_e$) is true.

($i1_e$) *Locations reachable from the root should be reachable from t or p.*

Both The Swap_Link_Loc operation and the swap statement are accessibility preserving operations, i.e. they do not change the accessibility of the graph. Furthermore, since these statements only involve locations *p* and *t*, we know that anything that was originally reachable from p_b or t_b is now reachable from p_e or t_e. In other words,

(Is_Reachable(x, t_b, Target) **or** Is_Reachable(x, p_b, Target) **iff**

(Is_Reachable(x, t_e, Target) **or** Is_Reachable(x, p_e, Target).

From conjunct ($i1_b$) in $invariant_b$, we know that everything initially reachable from root was reachable through p_b and t_b. Therefore, Is_Reachable(x, root, #Target) **iff** (Is_Reachable(x, t_e, $Target_e$) **or** Is_Reachable(x, p_e, $Target_e$).

($i2_e$) *The "marked" status of locations not reachable from the root should not change.*

From the current facts we know that ∀q: Location, $Is\_Marked_e(q) = Is\_Marked_b(q)$. Also, from $invariant_b$ we know $Is\_Marked_b(q) = \#Is\_Marked(q)$ for all locations not originally reachable from the root. Hence, $Is\_Marked_e(q) = \#Is\_Marked(q)$ for all locations *q* not originally reachable from the root.

($i3_e$) *All locations on the location stack should be marked.*

We show that the location stack at the end of the iteration is the same as the location stack at the beginning of the iteration, except that the top element, p_b, has been removed. In other words:

$Loc\_Stack(p_b, Target_b, CB\_Is\_Set_b) = p_b \circ Loc\_Stack(p_e, Target_e, CB\_Is\_Set_e)$

Let us consider the left-hand side of this equation.

$Loc\_Stack(p_b, Target_b, CB\_Is\_Set_b)$

$= p_b \circ Loc\_Stack(Target_b(p_b, RIGHT), Target_b, CB\_Is\_Set_b)$

(* from definition of Loc_Stack and fact that $CB\_Is\_Set_b(p_b)$ = true *)

$= p_b \circ Loc\_Stack(p_e, Target_b, CB\_Is\_Set_b)$

(* from fact that $Target_e(p_b, RIGHT) = Target_e(p_b, RIGHT) = p_e$ *)

$= p_b \circ Loc\_Stack(p_e, Target_b, CB\_Is\_Set_e)$

(* from fact that ∀q: Location, $CB\_Is\_Set_e(q) = CB\_Is\_Set_b(q)$ *)

We know that $t_e = p_b$. This location cannot occur on $Loc\_Stack(p_e, Target_b, CB\_Is\_Set_b)$ as there are no cycles on Loc_Stack. We also know that $Target_e(q, j) = Target_b(q, j)$ for all locations $q \neq t_e$ and all links *j*. Therefore, $p_b \circ Loc\_Stack(p_e, Target_b, CB\_Is\_Set_e) = p_b \circ Loc\_Stack(p_e, Target_e, CB\_Is\_Set_e)$, which is the right-hand side of the equation above.

Since the resulting location stack is a substring of the original location stack, and the code does not change the "marked" status of any locations, we know that ($i3_e$) is true.

($i4_e$) *A location should either be unmarked-reachable from t or from the right link of a location on the location stack.*

From conjunct ($i4_b$) of $invariant_b$ we know this is true at the beginning of the iteration. Therefore we can divide our proof up into two cases.

Case 1: *A location is unmarked-reachable from t_b.* From the path condition we know that t_b is at void or t_b is marked. Therefore, there cannot be any location that is unmarked-reachable from t_b.

Case 2: *A location is unmarked-reachable from the right link of a location on the location stack.* From ($i3_b$) we know that all locations on the stack are marked. From the facts about the program state, we know that the "marked" status of all locations remains the same. Therefore, we know that p_b is marked. Also, since CB_Is_Set(p_b) is true, Target(p_b, RIGHT) was a location on the Loc_Stack and hence was marked as well. So we know that there was no location that was unmarked-reachable from the right location of p_b. Also, we know the rest of the stack remains the same. Therefore, we have:

if Is_Reachable(x, root, #Target) **and not** Is_Marked(x) **then**
(Is_Unmarked_Reachable(x, t_e) **or**
∃y: Location **such that** ⟨y⟩ Is_Substring Loc_Stack(p_e, $Target_e$, $CB\_Is\_Set_e$) **and**
Is_Unmarked_Reachable(x, $Target_e$(y, RIGHT), $Target_e$))

Which is what we needed to show.

($i5_e$) *All locations that are not part of the location stack should have their original links.*

We showed in ($i3_e$) that the location stack at the beginning of the iteration was the same as p_b concatenated with the location stack at the end of the iteration. In the code, the only change we make to the target variable involves t_e which is the same as p_b. Therefore, all locations not on the stack will have their original links, with the possible exception of p_b. Therefore, it suffices to show that p_b has its original links. Using the definition of Stack_OK on p_b, and the fact that Control_Is_Set(p_b), we know that:

#Target(p_b, LEFT) = Target$_b$(p_b, LEFT)

#Target(p_b, RIGHT) = t_b

From the facts about the program state, we know that Target$_e$(p_b, LEFT) = Target$_e$(p_b, LEFT). Hence, #Target(p_b, LEFT) = Target$_e$(p_b, LEFT). We also know that Target$_b$(p_b, RIGHT) = p_e = t_b. Hence, the RIGHT link of p_b now points to the original RIGHT link when the procedure was called. Therefore, we have shown (i5$_e$).

(i6$_e$) *Locations on the stack should be able to have their original links restored.*

This occurs if *Stack_OK* is true for its parameters at the end of the iteration. The location stack at the beginning of the iteration is the same as p_b concatenated with the location stack at the end of the iteration. Since we are not changing the value of the *Target* variable for any location on the location stack, we know that *Stack_OK* will still be true.

4.3.2 Case 2 – Push

The Push block is executed when the following condition holds.

not Is_At_Void(t) **and not** Is_Marked(t)

The following code is executed.

```
p :=: t;
Swap_Link_Loc(p, LEFT, t);
Mark_Location(p);
Unset_Control(p);
```

Based on the specification, the only variables modified by the swap statement are *p* and *t*. The only variables potentially modified by Swap_Link_Loc are *t* and *Target*. The only variable modified by Mark_Location is *Is_Marked* and the only variable modified by Unset_Control is *CB_Is_Set*. Therefore, the only variables potentially modified by this block are *p*, *t*, *Target*, *Is_Marked*, and *CB_Is_Set*. Let t_b, p_b, $Target_b$, Is_Marked_b, and $CB\_Is\_Set_b$ represent the values of these variables at the beginning of an arbitrary iteration and t_e, p_e, $Target_e$, Is_Marked_e, and $CB\_Is\_Set_e$ represent the values of these variables at the end of the iteration. After the swap statement we know that

p' = tb and t' = pb

where *t*' and *p*' represent the current values of the variables.

After the call to Swap_Link_Loc, we know from the ensures clause that:

$\forall q$: Location, $\forall j$: {LEFT, RIGHT},

$$\text{Target}(q, j) = \begin{cases} t' & \textbf{if } q = p' \textbf{ and } j = \text{LEFT} \\ \text{Target}_b(q, j) & \textbf{otherwise}; \end{cases} \textbf{ and}$$

$t = \text{Target}_b(p', \text{LEFT})$

where *t* and *Target* represent the current values of the variables.

Substituting p_b for *t'*, we get

$\forall q$: Location, $\forall j$: {LEFT, RIGHT},

$$\text{Target}(q, j) = \begin{cases} p_b & \textbf{if } q = p \textbf{ and } j = \text{LEFT} \\ \text{Target}_b(q, j) & \textbf{otherwise}; \end{cases} \textbf{ and}$$

$t = \text{Target}_b(p', \text{LEFT})$

Mark_Location(p) ensures Is_Marked(p) = **true** and Unset_Control(p) ensures CB_Is_Set(p) = false. Hence, after the execution of the "push" block, we have:

$\forall q$: Location, $\forall j$: {LEFT, RIGHT},

$$\text{Target}_e(q, j) = \begin{cases} p_b & \textbf{if } q = p_e \textbf{ and } j = \text{LEFT} \\ \text{Target}_b(q, j) & \textbf{otherwise}; \end{cases} \textbf{ and}$$

$t_e = \text{Target}_b(p_e, \text{LEFT})$ **and** $p_e = t_b$ **and**

$$\text{Is\_Marked}_e(q) = \begin{cases} \textbf{true} & \textbf{if } q = p_e \\ \text{Is\_Marked}_b(q) & \textbf{otherwise}; \end{cases} \textbf{ and}$$

$$\text{Control\_Bit\_Is\_Set}_e(q) = \begin{cases} \textbf{false} & \textbf{if } q = p_e \\ \text{Control\_Bit\_Is\_Set}_b(q) & \textbf{otherwise}; \end{cases}$$

The path condition for Push allows us to use the following assertion.

not Is_At_Void(t) **and not** Is_Marked(t)

Given these facts, and assuming invariant_b, we prove invariant_e by showing that each conjunct ($i1_e$ – $i6_e$) is true.

($i1_e$) *Locations reachable from the root should be reachable from t or p.*

Both the Swap_Link_Loc operation and the swap statement are accessibility preserving operations, i.e. they do not change the accessibility of the graph. The same holds for Mark_Location and Unset_Control. Furthermore, since these statements only involve locations *p* and *t*, we know that anything that was originally reachable from p_b or t_b is now reachable from p_e or t_e. In other words,

(Is_Reachable(x, t_b, Target) **or** Is_Reachable(x, p_b, Target) **iff**
(Is_Reachable(x, t_e, Target) **or** Is_Reachable(x, p_e, Target)

From conjunct ($i1_b$) in $invariant_b$, we know that everything initially reachable from root was reachable through p_b and t_b. Therefore, Is_Reachable(x, root, #Target) **iff** (Is_Reachable(x, t_e, $Target_e$) **or** Is_Reachable(x, p_e, $Target_e$).

($i2_e$) *The "marked" status of locations not reachable from the root should not change.*

From $invariant_e$ (conjunct $i1_e$), we know that Is_Reachable(p_e, root, #Target). Also from the facts we know that ∀q: Location, **if** q ≠ p_e **then** $Is\_Marked_e(q)$ = $Is\_Marked_b(q)$. Therefore, we can deduce that

if not Is_Reachable(x, root, #Target) **then** Is_Marked(x) = #Is_Marked(x)

($i3_e$) *All locations on the location stack should be marked.*

We show that the location stack at the end of the iteration is the same as the location stack at the beginning of the iteration, except that the top element, p_e, has been added. In other words:

$Loc\_Stack(p_e, Target_e, CB\_Is\_Set_e)$ = p_e ∘ $Loc\_Stack(p_b, Target_b, CB\_Is\_Set_b)$

Let us consider the left-hand side of this equation.

$Loc\_Stack(p_e, Target_e, CB\_Is\_Set_e)$
= p_e ∘ $Loc\_Stack(Target_e(p_e, LEFT), Target_e, CB\_Is\_Set_e)$
(* From definition of Loc_Stack and fact $CB\_Is\_Set(p_e)$ = false *)
= p_e ∘ $Loc\_Stack(p_b, Target_e, CB\_Is\_Set_e)$
(* Since $Target_e(p_e, LEFT) = p_b$ *)

We know that $t_b = p_e$ and this location cannot occur on Loc_Stack(p_b, $Target_e$, $CB\_Is\_Set_e$) as there are no cycles on Loc_Stack We also know that $Target_e(q, j) = Target_b(q, j)$ and $CB\_Is\_Set_b(q) = CB\_Is\_Set_e(q)$ for all locations $q \neq p_e$ and all links *j*. Therefore,

$p_e \circ$ Loc_Stack(p_b, $Target_e$, $CB\_Is\_Set_e$) = $p_e \circ$ Loc_Stack(p_b, $Target_b$, $CB\_Is\_Set_b$)

This is the same as the right hand side.

We also know that $Is\_Marked_e(q) = Is\_Marked_b(q)$ for all locations $q \neq p_e$ and $Is\_Marked_e(p)$ = **true**. Hence, we can deduce that all locations on the location stack are marked.

($i4_e$) *A location should either be unmarked-reachable from t or from the right link of a location on the location stack.*

From conjunct ($i4_b$) of $invariant_b$ we know this is true at the beginning of the iteration. Therefore we can divide our proof up into two cases.

Case 1: *A location is unmarked-reachable from t_b*. We know that $t_e = Target_b(p_e, LEFT)$ and $p_e = t_b$. Therefore, we can say that $t_e = Target_b(t_b, LEFT)$. Hence, something unmarked-reachable through left link of t_b is now reachable from t_e. Also, $p_e = t_b$ and $Target_b(p_e, RIGHT) = Target_e(p_e, RIGHT)$. Hence, something unmarked reachable from right link of t_b is now reachable through the right link of p_e which is on the stack. Hence,

if Is_Unmarked_Reachable(x, t_b, $Target_b$) **then**
Is_Unmarked_Reachable(x, t_e, $Target_e$) **or**
Is_Unmarked_Reachable(x, $Target_e(p_e, RIGHT)$, $Target_e$).

Case 2: *A location is unmarked-reachable from the right link of a location on the location stack*. We know, Loc_Stack(p_e) = $p_e \circ$ Loc_Stack(p_b). Hence, any location unmarked reachable from the right link of a location on the original stack is still unmarked reachable from the right link of a location on the new stack.

From case1 and case2 we get,

if Is_Reachable(x, root, #Target) **and not** Is_Marked(x) **then**
(Is_Unmarked_Reachable(x, t_e) **or**

∃y: Location **such that** ⟨y⟩ Is_Substring Loc_Stack(p_e, $Target_e$, $CB\_Is\_Set_e$) **and**
Is_Unmarked_Reachable(x, $Target_e$(y, RIGHT), $Target_e$))

($i5_e$) *All locations that are not part of the location stack should have their original links.*

We showed in ($i3_e$) that the location stack at the end of the iteration was the same as p_e concatenated with the location stack at the beginning of the iteration. In the code, the only change we make to the target variable involves t_b which is the same as p_e which is on the current stack. Therefore, all locations not on the stack will have their original links

if not In_String(x, Loc_Stack(p)) **then**
Target(x, RIGHT) = #Target(x, RIGHT) **and** Target(x, LEFT) = #Target(x, LEFT)

($i6_e$) *Locations on the stack should be able to have their original links restored.*

This occurs if *Stack_OK* is true for its parameters at the end of the iteration. The location stack at the end of the iteration is the same as p_e concatenated with the location stack at the beginning of the iteration. Since we are not changing the value of the *Target* variable for any location on the location stack other than p_e, we know that *Stack_OK* will still be true if we prove that we can get the original links for p_e.

We know that Control_Is_Set(p_e) = **false**. From the facts about the program state we know that $Target_b$(p_e, RIGHT) = $Target_e$(p_e, RIGHT). Since p_e was not a part of the stack at the beginning of the iteration we have $Target_b$(p_e, RIGHT) = #Target(p_e, RIGHT). Therefore, we get $Target_e$(p_e, RIGHT) = #Target (p_e, RIGHT). We also know that $Target_b$(p_e, LEFT) = #Target(p_e, LEFT) because p_e was not a part of the stack at the beginning of the iteration. Also we have, $Target_b$(p_e, LEFT) = t_e. Hence, we get #$Target_b$(p_e, LEFT) = t_e which implies that the original links of p_e can be restored.

4.3.3 Case 3 – Swing

The Swing block is executed when the following condition holds.

not Control_Is_Set(p) **and** (Is_At_Void(t) **or** Is_Marked(t))

The following code is executed.

Swap_Links(p, LEFT, p, RIGHT);

Swap_Link_Loc(p, LEFT, t);

Set_Control(p);

Based on the specification, the only variable potentially modified by the Swap_Links statement is *Target*. The only variables potentially modified by Swap_Link_Loc are *t* and *Target*. The only variable modified by Set_Control is *CB_Is_Set*. Note that *Is_Marked* does not change. Therefore, the only variables potentially modified by this block are *p*, *t*, *Target*, and *CB_Is_Set*. Let t_b, p_b, $Target_b$, Is_Marked_b, and $CB\_Is\_Set_b$ represent the values of these variables at the beginning of an arbitrary iteration and t_e, p_e, $Target_e$, Is_Marked$_e$, and CB_Is_Set_e and represent the values of these variables at the end of the iteration. After the call to Swap_Links we know (from the ensures clause) that:

$\forall$q: Location, $\forall$j: {LEFT, RIGHT},

$$\text{Target'}(q, j) = \begin{cases} \text{Target}_b(p, \text{RIGHT}) & \textbf{if } q = p \textbf{ and } j = \text{LEFT} \\ \text{Target}_b(p, \text{LEFT}) & \textbf{if } q = p \textbf{ and } j = \text{RIGHT} \\ \text{Target}_b(q, j) & \textbf{otherwise}; \end{cases}$$

where Target' is the current value of the Target variable.

After the call to Swap_Link_Loc, we know that:

$\forall$q: Location, $\forall$j: {LEFT, RIGHT},

$$\text{Target}(q, j) = \begin{cases} t_b & \textbf{if } q = p \textbf{ and } j = \text{LEFT} \\ \text{Target}_b(p, \text{LEFT}) & \textbf{if } q = p \textbf{ and } j = \text{RIGHT } \textbf{and} \\ \text{Target}_b(q, j) & \textbf{otherwise}; \end{cases}$$

t = Target'(p, LEFT) = Target_b(p, RIGHT)

where *t*, *p* and *Target* represent the current values of the variables.

Set_Control(p) ensures CB_Is_Set(p) = true. Hence, after the execution of the "push" block, we have:

$\forall$q: Location, $\forall$j: {LEFT, RIGHT},

$$\text{Target}_e(q, j) = \begin{cases} t_b & \textbf{if } q = p \textbf{ and } j = \text{LEFT} \\ \text{Target}_b(p, \text{LEFT}) & \textbf{if } q = p \textbf{ and } j = \text{RIGHT } \textbf{and} \\ \text{Target}_b(q, j) & \textbf{otherwise}; \end{cases}$$

t_e = Target$_b$(p, RIGHT) **and** Is_Marked$_e$(q) = Is_Marked$_b$(q) **and**

$$\text{Control\_Bit\_Is\_Set}_e(q) = \begin{cases} \textbf{false} & \textbf{if } q = p_e \\ \text{Control\_Bit\_Is\_Set}_b(q) & \textbf{otherwise}; \end{cases}$$

The path condition for Swing allows us to use the following assertion.

not Control_Is_Set(p) **and** (Is_At_Void(t) **or** Is_Marked(t))

Given these facts, and assuming invariant$_b$, we prove invariant$_e$ by showing that each conjunct ($i1_e$ – $i6_e$) is true.

($i1_e$) *Locations reachable from the root should be reachable from t or p.*

Both the Swap_Links operation and the Swap_Link_Loc operation are accessibility preserving operations, i.e. they do not change the accessibility of the graph. The same holds for Set_Control. Furthermore, since these statements only involve locations *p* and *t*, we know that anything that was originally reachable from p_b or t_b is now reachable from p_e or t_e. In other words,

(Is_Reachable(x, t_b, Target) **or** Is_Reachable(x, p_b, Target) **iff**
(Is_Reachable(x, t_e, Target) **or** Is_Reachable(x, p_e, Target).

From conjunct ($i1_b$) in invariant$_b$, we know that everything initially reachable from root was reachable through p_b and t_b. Therefore, Is_Reachable(x, root, #Target) **iff** (Is_Reachable(x, t_e, Target$_e$) **or** Is_Reachable(x, p_e, Target$_e$).

($i2_e$) *The "marked" status of locations not reachable from the root should not change.*

From the current facts we know that ∀q: Location, Is_Marked$_e$(q) = Is_Marked$_b$(q). Also, from invariant$_b$ we know Is_Marked$_b$(q) = #Is_Marked(q) for all locations not originally reachable from the root. Hence, Is_Marked$_e$(q) = #Is_Marked(q) for all locations *q* not originally reachable from the root.

($i3_e$) *All locations on the location stack should be marked.*

We show that the location stack at the end of the iteration is the same as the location stack at the beginning of the iteration. In other words:

Loc_Stack(p_e, Target$_e$, CB_Is_Set$_e$) = Loc_Stack(p_b, Target$_b$, CB_Is_Set$_b$)
=> p_e ∘ Loc_Stack(Target$_e$(p_e, RIGHT), Target$_e$, CB_Is_Set$_e$) =

$p_b \circ$ Loc_Stack(Target$_b$(p_b, LEFT), Target$_b$, CB_Is_Set$_b$)

(* We know $p_e = p_b$. From the definition of Loc_Stack and fact that

CB_Is_Set$_e$(p_e) = true, CB_Is_Set$_b$(p_b) = false) *)

=> Loc_Stack(Target$_e$(p_e, LEFT), Target$_e$, CB_Is_Set$_e$) =

Loc_Stack(Target$_b$(p_b, RIGHT), Target$_b$, CB_Is_Set$_b$)

=> Loc_Stack(z, Target$_e$, CB_Is_Set$_e$) = Loc_Stack(z, Target$_b$, CB_Is_Set$_b$)

(* Target$_b$(p_b, LEFT) = Target$_e$(p_e, RIGHT). Let z = Target$_b$(p_b, LEFT) *)

We know that there are no cycles on Loc_Stack We also know that Target$_e$(q, j) = Target$_b$(q, j) and CB_Is_Set$_b$(q) = CB_Is_Set$_e$(q) for all locations $q \neq p_e$ and all links *j*. Therefore, we arrive at the tautology below.

Loc_Stack(z, Target$_b$, CB_Is_Set$_b$) = Loc_Stack(z, Target$_b$, CB_Is_Set$_b$)

We also know that Is_Marked$_e$(q) = Is_Marked$_b$(q) for all locations $q \neq p_e$ and Is_Marked$_e$(p) = true. Hence, we can deduce that all locations on the location stack are marked.

(i4$_e$) *A location should either be unmarked-reachable from t or from the right link of a location on the location stack.*

From conjunct (i4$_b$) of invariant$_b$ we know this is true at the beginning of the iteration. Therefore we can divide our proof up into two cases.

Case 1: *A location is unmarked-reachable from t_b*. From the path condition we know that t_b is at void or t_b is marked. Therefore, there cannot be any location that is unmarked-reachable from t_b.

Case 2: *A location is unmarked-reachable from the right link of a location on the location stack*. We know that the location stack does not change. However, the current right link of *p* points to the location pointed to by the original left link of *p*. Also, t_e resides at the location pointed to by the original right link of *p*. Hence, any location that was unmarked-reachable from the right link of *p* at the beginning of the iteration is wow unmarked-reachable through t_e. For all other locations on the stack, the links are unchanged.

From case1 and case2 we get,

if Is_Reachable(x, root, #Target) **and not** Is_Marked(x) **then**

(Is_Unmarked_Reachable(x, t_e) **or**

∃y: Location **such that** ⟨y⟩ Is_Substring Loc_Stack(p, $Target_e$, $CB\_Is\_Set_e$) **and**

Is_Unmarked_Reachable(x, $Target_e$(y, RIGHT), $Target_e$))

($i5_e$) *All locations that are not part of the location stack should have their original links.*

We showed in ($i3_e$) that the location stack at the end of the iteration was the same as the location stack at the beginning of the iteration. In the code, the only change we make to the target variable involves *p* which is on the current stack. Therefore, all locations not on the stack will have their original links. In other words:

if not In_String(x, Loc_Stack(p)) **then**

Target(x, RIGHT) = #Target(x, RIGHT) **and** Target(x, LEFT) = #Target(x, LEFT)

(i6e) *Locations on the stack should be able to have their original links restored.*

This occurs if *Stack_OK* is true for its parameters at the end of the iteration. The location stack at the end of the iteration is the same as the location stack at the beginning of the iteration. Since we are not changing the value of the *Target* variable for any location on the location stack other than *p*, we know that *Stack_OK* will still be true if we prove that we can get the original links for *p*.

We know that $Control\_Is\_Set_b$(p) = false. Hence, from $i6_b$, we get

#Target(p ,LEFT) = t_b **and** #Target(p, RIGHT) = $Target_b$(p, RIGHT)

To prove that the original links can be restored, we need to prove

#Target(p, LEFT) = $Target_e$(p, LEFT) **and** #Target(p, RIGHT) = t_b.

(* Since $CB\_Is\_Set_e$(p) = true *)

From the facts about the program state we know, $Target_b$(p, RIGHT) = $Target_e$(p, LEFT) **and** $Target_e$(p, LEFT) = tb. Substituting these in the first assertion, we get

#Target(p, LEFT) = $Target_e$(p, LEFT) **and** #Target(p, RIGHT) = t_b

This is the same as the second assertion and hence we can say that the original links can be restored.

5. Conclusion and Future Work

This thesis presented a formal specification of the Schorr-Waite algorithm in Resolve. Custom definitions such as Is_Path and Is_Reachable were given in the specification. These definitions assisted in the reasoning of the code. We presented two different implementations of the Schorr-Waite algorithm. The first was a more traditional implementation that was based on the description of the algorithm in [Mehta] and [Marche]. Custom definitions such as Loc_Stack and Stack_OK were used in the Realization (implementation) to help describe the loop invariant and to help prove the correctness of the implementation. Both the formal specification and the traditional implementation were based on the pointer component developed in [tech report].

Using traditional pointer operations, assertions about reachability were difficult to prove. To simplify reasoning, we introduced the accessibility-preserving operations. These operations maintain the accessibility of the system after they are executed. Hence, proofs that involve maintaining reachability of nodes are made easier. An added advantage to this approach was significantly shorter code for the Schorr-Waite algorithm and avoidance of a temporary variable used in Schorr-Waite coding examples from other research studies.

Earlier proofs of the Schorr-Waite algorithm have required explicit reasoning of the heap. The idea for the proof in this thesis was to present a modular approach of reasoning to pointers where we can reason about pointers just as we can for any other component. The proof of the Schorr-Waite algorithm is given for the implementation using the accessibility-preserving operations. The proof centered on the correctness of the loop invariant, which was broken down into three phases: initialization, termination and maintenance. The proof for maintenance was broken down into three separated blocks of code that could be executed for any given iteration of the loop: the push, pop and swing blocks.

5.1 Future Work

To help generate the verification conditions to prove the correctness of a program, the Resolve verifying compiler [x] is currently in development. The goal for the verifying

compiler is to automate as much of the verification process as possible. Currently, if the compiler is given a specification, a proposed implementation, and the specifications of the components it uses, the verifying compiler is able to generate verification conditions that can be fed into an automatic theorem prover such Isabelle. As the pointer component is quite complex, there are some verification conditions that the compiler cannot yet generate. We are working closely with researchers in Clemson University to fix these problems.

Isabelle is theorem prover with ML-like syntax. It can prove theorems with some user assistance. However, for the proof, one needs to code all the facts as well as definitions used by the code in Isabelle. It can directly use verification conditions such as those generated by the Resolve verifying compiler. Programmers can then attempt to prove these verification conditions using Isabelle's library of mathematical theories. The theories may be built-in or custom-made.

An important future goal of this research is not only to demonstrate how to prove component-based pointer programs, but also to show the extent to which such proofs can be automated. We are currently working on slight modifications to the specification of the Schorr-Waite algorithm so that the Resolve verifying compiler can automatically generate the required verification conditions for the proof of correctness. These verification conditions will then be discharged using the Isabelle theorem prover.

Bibliography

[1] Bornat, R.. Proving pointer programs in Hoare logic. In Mathematics of Program Construction, pages 102–126, 2000.

[2] Bornat, R. and Sufrin, B. Animating formal proof at the surface: The Jape proof calculator. The Computer Journal, 42(3):177–192, 1999. http://jape.org.uk.

[3] Burstall, R. Some techniques for proving correctness of programs which alter data structures. Machine Intelligence, 7:23–50, 1972.

[4] Hubert, T. and Marche, C. A case study of C source code verification: the Schorr-Waite algorithm**.** In Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods, 2005

[5] Kulczycki, G., Sitaraman, M., Ogden, W. F. and Hollingsworth, J. E., "Component Technology for Pointers: Why and How," Technical Report RSRG-03-03, Clemson University, 2003.

[6] Kulczycki, G., Sitaraman, M., Weide, B. W. and Rountev, A. "A Specification-Based Approach to Reasoning about Pointers," in Proceedings ESEC/FSE SAVCBS '05 Workshop, ACM Software Engineering Notes, Vol. 31, No. 2, 2005.

[7] Kulczycki, G., Sitaraman, M., Keown, H. and Weide, B. W."Abstracting Pointers for a Verifying Compiler," in Proceedings 31st Annual Software Engineering Workshop, Baltimore, MD. March, 2007.

[8] Luckham, D. C. and Suzuki, N. 1979. Verification of array, record, and pointer operations in pascal. ACM Transactions on Programming Languages and Systems (TOPLAS) 1, 2, 226–244.

[9] Mehta, F. and Nipkow, T. Proving pointer programs in higherorder logic. In F. Baader, editor, 19th Conference on Automated Deduction, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[10] Morris, J. M.. A proof of the Schorr-Waite algorithm. In M. Broy and G. Schmidt, editors, Theoretical Foundations of Programming Methodology, pages 25–51. Reidel, 1982. Proceedings of the 1981 Marktoberdorf summer school.

[11] Singh, A., and Kulczycki, G., Accessibility Preserving Operations for Pointers, Proceedings of the Resolve 2007 Workshop.

[12] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E. 2000. Reasoning about software-component behavior. In Procs. Sixth Int. Conf. on Software Reuse. Springer-Verlag, 266–283.

[13] Schorr, H. and Waite, W. M. An efficient machineindependent procedure for garbage collection in various list structures. Commun. ACM, 10:501–506, 1967.

[14] Wing, J. M. 1990. A specifier's introduction to formal methods. IEEE Computer 23, 9, 8–24.