Proc. of the 1997 IEEE Automated Software Engineering Conference, Nov., 1997, pp 219--226

A Formal Automated Approach for Reverse Engineering Programs with Pointers*

Gerald C. Gannod† and Betty H. C. Cheng‡

Department of Computer Science
Michigan State University
3115 Engineering Building
East Lansing, Michigan 48824
E-mail: {gannod,chengb}@cps.msu.edu

Abstract

Given a program S and a precondition Q, the strongest postcondition, denoted $sp(S, Q)$, is defined as the strongest condition that holds after the execution of S, given that S terminates. By defining the formal semantics of each of the constructs of a programming language, a formal specification of the behavior of a program written using the given programming language can be constructed. In this paper we address the formal semantics of pointers in order to handle a realistic model of programming languages that incorporate the use of pointers. In addition, we present a tool for supporting the construction of formal specifications of programs that include the use of pointers.

1. Introduction

As the demands placed on software continue to grow, there is an increasing recognition that software can be error prone. Moreover, the rising costs for software development impose the need to use a given piece of software for a longer period of time, for multiple purposes, and for increasingly larger customer bases. As a result, there is a need for more sophisticated and systematic approaches to maintain software.

Formal methods are techniques that incorporate the use of formal specification languages, where a formal specification language has a well-defined syntax and semantics. In addition, formal methods have associated calculation rules that can be used to analyze specifications in order to determine correctness and consistency. Since the notations have a formal mathematical basis, formal methods facilitate the use of automated processing [2]. Reverse engineering of program code is the process of examining components and component interrelationships in order to construct a more abstract representation of an implementation [3]. The primary focus of our research is to apply the use of formal methods to the reverse engineering of program code in order to support maintenance and evolutionary activities, where the formal approach facilitates automated processing.

Our previous investigations have focused on the use of the *strongest postcondition* for deriving as-built formal specifications from imperative program code [6]. Given a program S and a precondition Q, the strongest postcondition, denoted $sp(S, Q)$, is defined as the strongest condition that holds after the execution of S, given that S terminates. By defining the formal semantics of each of the constructs of a programming language, a formal specification of the behavior of a program written using the given programming language can be constructed. Using this approach, we have defined the formal semantics of a subset of the C programming language that excludes the use of pointers [7]. This paper describes the formal semantics of pointers as found in commonly used programming languages. In addition, we present a tool for supporting the construction of formal specifications of programs with pointers.

The remainder of this paper is organized as follows. Section 2 discusses background material in the areas of formal methods and reverse engineering. The formal semantics of pointers are presented in Section 3. Section 4 presents applications of a support tool to specific examples. Section 5 discusses related work. Finally, Section 6 draws conclusions and suggests further investigations.

2. Background

This section describes background material in the areas of pointers, software maintenance, and formal methods for software development.

2.1. Pointers

Using terminology of the C programming language, a *pointer* is a variable that contains the address of a variable.

*This work is supported in part by the NASA Graduate Student Research Fellowship NGT-70376 and the National Science Foundation grants CCR-9633391, CCR-9407318, CCR-9209873, and CDA-9312389.

†A portion of this research was performed while this author was at the NASA Jet Propulsion Laboratory.

‡Please address all correspondences to this author.

A common use of pointers is the creation of *aliases*, which refers to the condition when several names can be used to refer to a single data object. For instance, the statement "`x := @a`", where `x` is a pointer and `a` is some data variable, creates an alias, thus making operations involving `x` and `a` synonymous. The notation "`*p`" indicates a *dereference* of the pointer `p` in order to access the value of the referenced object. There are four different classes of alias detection: intraprocedural may-alias, interprocedural may-alias, intraprocedural must-alias, and interprocedural must-alias. The term *may-alias* refers to the condition when two variables, during *some execution* of a program, are aliases for one another. The term *must-alias* means that during *all executions* of the program, the variables will be aliases for one another. The terms *interprocedural* and *intraprocedural* indicate the context of the aliasing, where interprocedural is global and intraprocedural is local. Compile-time analysis of programs to detect aliasing has long been recognized as difficult. In fact, it has been proven that static analysis to detect aliases is undecidable [10]. In this paper we do not address may/must-aliasing problems directly. Instead, our intention in the development of the formal semantics for pointers is to provide a theoretically rich formalism that can be used to aid may/must-alias analysis through mathematical proof techniques.

In addition to having may/must-alias detection, alias detection techniques can be *flow-sensitive* or *flow-insensitive*. A technique is flow sensitive if control structures are factored into the detection algorithm. The techniques that we suggest are flow sensitive although, again, we do not directly address alias detection.

2.2. Software Maintenance

Figure 1 contains a graphical depiction of a process model for reverse and re-engineering [1]. The process model is captured by two sectioned triangles, where each section in a triangle represents a different level of abstraction. The higher levels in the model are *concepts* and *requirements*. The lower levels include *designs* and *implementations*. Entry into this re-engineering process model begins with system *A*, where *Abstraction* (or reverse engineering) is performed to an appropriate level of detail. The next step is *Alteration*, where the system is configured into a new form at a different level of abstraction. Finally, *Refinement* of the new form into an implementation can be performed to create system *B*.

This paper describes an approach to reverse engineering that is applicable to the *implementation* and *design* levels. In Figure 1, the context for this paper is represented by the dashed arrow. That is, we address the construction of formal low-level or "*as-built*" design specifications. The motivation for operating in such an implementation-bound level of abstraction is that it provides a means of traceability between the program source code and the formal specifications constructed using the techniques described in this paper. This traceability is necessary in order to facilitate technology transfer of formal methods [4]. That is, current development teams must be able to understand the relationship between the source code and the specifications before taking advantage of the benefits offered by formal methods.

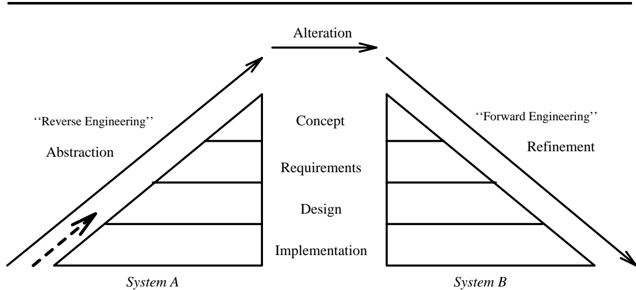

Figure 1. Reverse Engineering Process Model

2.3. Strongest Postcondition

Strongest postcondition, denoted $sp(S, Q)$ is defined as follows: given that Q holds, execution of S results in $sp(S, Q)$ true, if S terminates [5]. As such, $sp(S, Q)$ assumes partial correctness. The context for our investigations is that we are reverse engineering systems that have desirable properties or functionality that should be preserved or extended. Therefore, the partial correctness model is sufficient for these purposes since the termination properties of these systems are known *a priori*.

The *weakest precondition* predicate transformer $wp(S, R)$ is defined as the set of all states in which the statement S can begin execution and terminate with postcondition R true. Given a Hoare triple $Q \{ S \} R$, we note that wp is a "backward" rule, in that a derivation of a specification begins with postcondition R, and produces a predicate $wp(S, R)$. The predicate transformer wp assumes a total correctness model of computation, meaning that given S and R, if the computation of S begins in state $wp(S, R)$, then the program S *will* halt with condition R true. We contrast this model with the sp model, a "forward" derivation rule. That is, given a precondition Q and a program S, sp derives a predicate $sp(S, Q)$. The motivation for using sp instead of wp is that sp derives a postcondition, which is typically the objective of reverse engineering. That is, to determine the purpose of a given code segment. In constrast, wp requires a postcondition in order to derive a precondition.

In this paper we use a variation of the Dijkstra guarded command language as the target programming language with the intent of constructing analogous formalizations for languages such as C and Pascal. We find this approach to be reasonable given our initial experiences in formalizing the C programming language [7].

Table 1 summarizes the strongest postcondition semantics of the Dijkstra guarded command language as given in [5], where the notation P^y_c means that every free occurrence of y in P is replaced by c, "::" indicates that the range of the quantified variable is not explicitly referenced in the current context. In addition, `IF` represents the n alternative conditional statement `if B`$_1$ $\rightarrow$ `S`$_1$`; ...` $\|$ `B`$_n$ $\rightarrow$ `S`$_n$`; fi`. `B`$_i$ $\rightarrow$ `S`$_i$ represents a guarded command such that `S`$_i$ is only executed if logical expression (guard) `B`$_i$ is true. `DO` represents the loop statement "`do B` $\rightarrow$ `S od`" where `S` is executed iteratively until guard `B` is false.

Construct		*sp* Semantics
$sp(\texttt{x := e}, Q)$	$\equiv$	$(\exists v :: Q^x_v \wedge x = e^x_v)$
$sp(\texttt{IF}, Q)$	$\equiv$	$(\exists i : 1 \leq i \leq n : sp(S_i, B_i \wedge Q))$
$sp(\texttt{DO}, Q)$	$\equiv$	$\neg B \wedge (\exists i : 0 \leq i : sp(\texttt{IF}^i, Q))$
$sp(S_1; S_2, Q)$	$\equiv$	$sp(S_2, sp(S_1, Q))$

Table 1. Strongest Postcondition Semantics for the Dijkstra Guarded Command Language

In the table, the semantics for $sp(\texttt{x := e}, Q)$ states that after the execution of "`x := e`" there exists some value v such that every free occurrence of x in Q is replaced with v and $x = e^x_v$. The semantics for $sp(\texttt{IF}, Q)$ states that after execution of the `if-fi` statement, at least one of $sp(S_i, B_i \wedge Q)$ is true. In the case of iteration, denoted $sp(\texttt{DO}, Q)$, the semantics are that after execution of the loop, the loop guard is false ($\neg B$), and a disjunctive expression describing the effects of iterating the loop some number of times (approximated by the notation $\texttt{IF}^k$), where $k \geq 0$. Finally, for sequences, $sp(S_1; S_2, Q)$ means that the postcondition for statement `S`$_1$ is the precondition for some subsequent statement `S`$_2$.

3. Pointer Semantics

A *pointer* is a variable that contains the address of some data object. Pointers can be assigned in a number of different ways including heap allocation and direct addressing of a variable. For instance, the C-like command "`p := @k`" assigns the address of the variable `k` to pointer variable `p`. As such, the pointer variable `p` *points-to* variable `k`. This section describes the strongest postcondition semantics of pointers.

3.1. Conventional Assignment Semantics

The strongest postcondition semantics of the assignment statement is as follows. Given a statement "`x := e`" and a precondition Q:

$$sp(\texttt{x := e}, Q) \equiv (\exists v :: Q^x_v \wedge x = e^x_v). \qquad (1)$$

Two lemmas for eliminating the existential quantification in Expression (1) have been described [8]. Due to space constraints they are not repeated here. However, below we describe an outline of the lemmas. In the first lemma, if the precondition Q is of the form $C \wedge (x = u)$, where C is a logical expression, then after the textual substitution of variable x with v in Q, Expression (1) reads as $(\exists v :: C^x_v \wedge (v = u) \wedge x = e^x_v)$. Since $(v = u)$, the expression $(\exists v :: C^x_v \wedge (v = u) \wedge x = e^x_v)$ is logically equivalent to $C^x_u \wedge (u = u) \wedge x = e^x_u$. In the second lemma, if x does not appear as a free variable in either the logical expression Q or the expression e, then $(\exists v :: Q^x_v \wedge x = e^x_v)$ is logically equivalent to $Q \wedge x = e$.

In a naive treatment of pointers, we can attempt to apply the semantics of the assignment statement to pointer variables. However, doing so causes various problems. Consider the example in Figure 2 where `p` and `q` are pointer variables, `d` is a typed variable, and `e` is a constant. Given the precondition $\{*q = Y\}$, where $*q$ is a dereference of an object and Y is a constant, the strongest postcondition of the statement sequence "`p := q; d := *p; *p := e`" is $\{d = v \wedge p = q \wedge *q = Y \wedge *p = e\}$ when the conventional semantics for assignment is used for pointer assignments. This specification, derived using the two lemmas described in [8], states that after execution of the sequence, d has value v, p and q point to the same object, and the value of $*q = Y$ and $*p = e$. The problem with this specification is that while $p = q$ (i.e., pointers `p` and `q` refer to the same object), there is a contradiction in the conjuncts $*q = Y$ and $*p = e$.

{ $\quad *q = Y \quad$ }
`p` $\quad$ `:= q;`
{ $\quad (\exists v :: (*q = Y)^p_v \wedge p = q)$
$\quad\equiv (p = q \wedge *q = Y) \quad$ }
`d` $\quad$ `:= *p;`
{ $\quad (\exists v :: (p = q \wedge *q = Y)^d_v \wedge d = *p)$
$\quad\equiv (d = *p \wedge p = q \wedge *q = Y) \quad$ }
`*p` $\quad$ `:= e;`
{ $\quad (\exists v :: (d = *p \wedge p = q \wedge *q = Y)^{*p}_v \wedge *p = e)$
$\quad\equiv (d = v \wedge p = q \wedge *q = Y \wedge *p = e) \quad$ }

Figure 2. A simple pointer example

The remainder of this section presents a model for describing the formal semantics of pointer operations that overcome the problems that occur when using conventional assignment semantics.

3.2. Memory Model

In the C programming language, variables can be allocated from heap storage, registers, or address stack. The model used in this paper for representing memory is cell based, where the memory consists of a large number of storage cells. Each cell is named and contains a value. A diagram of this model is shown in Figure 3, where the entries in the column labeled *N* indicate the names of the cells, and

the entries in the column labeled *V* indicate the values. In the diagram, data objects *x*, *y*, and *z* have values *a*, *b*, and *c*, respectively. As a convention we use "$n.V$", where n is a cell name, to denote the value of the data object n. In our example, $x.V = a$, $y.V = b$, and $z.V = c$.

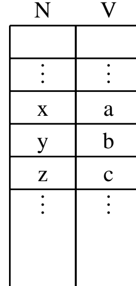

Figure 3. Cell Memory Model

3.3. Extending the Model for Pointers

A pointer can be assigned by heap allocation, pointer assignment, or alias assignment. Different alternatives for representing the use of pointers within the context of the cell memory model are available including the use of indirection where the value of the pointer is the name of the variable being pointed to.

Consider the set of data objects N and the set of pointers M that are currently allocated at some step during the execution of a particular program. Assuming that all the pointers in M point to data objects (not necessarily distinct) in N, using the equivalence relation "=" where pointer $p = q$ if and only if p and q point to the same object, we can partition M such that each partition is an equivalence class. As such, any operation on a member of a particular equivalence class is behaviorally equivalent to performing an operation on any other member of the same equivalence class. For instance, suppose we have variables `x` and `y` and pointers `p`, `q`, `r`, `s`, and `t`. Let pointers `p`, `q`, and `r` point to variable `x`, and pointers `s` and `t` point to variable `y`. Since `p`, `q`, and `r` point to the same variable `x`, they form one equivalence class, and since `s` and `t` point to the other variable, they form another equivalence class.

The equivalence classes within the set of pointers M can be considered to be *dynamic* since the execution of a programming statement can possibly rearrange the members of each set as is the case when pointer variables are either reused in heap allocation or a pointer assignment. Figure 4 depicts the extension of the memory model where there is an associated equivalence class in M for each memory cell. For consistency sake we assume that a data object can reference itself and, as such, is a member of the associated equivalence class. For example, the data object `y` with pointers `s` and `t` has an associated equivalence class from M with members { `y`, `s`, `t` }. We refer to this equivalence class as $M[y]$.

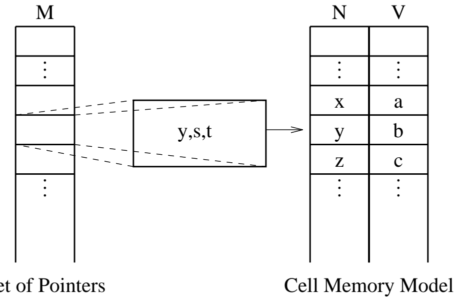

Figure 4. Extensions to the Memory Model

3.4. Points-to and Coset

This section defines the semantics of the *points-to* relation and the *coset* function, both of which are used to formally describe the behavior of pointer operations.

Let M be the set of pointers, N be the set of allocated data objects, and $\mathcal{B}$ be the Boolean type. Figure 5 defines the $\gtrdot$ (pronounced "*points-to*") relation. The primary use of the *points-to* relation is for making assertions about pointers and their relation to specific data objects. That is, it asserts that a pointer is in the equivalence class associated to a particular data object. Informally, the *points-to* relation is a heterogeneous relation on $\{M \cup N\} \times N$. The first axiom states that when data objects $o1$ and $o2$ are both in the set N that $o1 \gtrdot o2$ is true if and only if $o1 = o2$, where "$o1 = o2$" when $o1$ and $o2$ are the same data object. As such, a data object can only reference itself and never references another data object. The second axiom states that for a pointer $p \in M$ and a data object $o \in N$, $p \gtrdot o$ if and only if $p \in M[o]$. That is, p points-to o if and only if pointer p is an element of the equivalence class of o.

$$_ \gtrdot _ : \quad \{M \cup N\} \times N \mapsto \mathcal{B}$$

$Axioms:$

$$(\forall o1, o2 : o1, o2 \in N : o1 \gtrdot o2 \Leftrightarrow o1 = o2)$$
$$(\forall p, o : p \in M \wedge o \in E : p \gtrdot o \Leftrightarrow p \in M[o])$$

Figure 5. The *points-to* relation

The *coset* function is defined in Figure 6. The primary use of the *coset* function is to identify a dereferenced object. Informally, the *coset* function maps pointers to data objects, where a pointer p maps to a data object o if $p \in M[o]$. If a pointer does not belong to any equivalence class then the function is undefined.

$$coset : M \mapsto \{N \cup \{undefined\}\}$$
$$coset(p) = \begin{cases} o & \textit{if and only if } \; p \gtrdot o, \\ undefined & otherwise. \end{cases}$$

Figure 6. The *coset* function

3.5. Assignment Revisited

Given the definition of the *points-to* operator and the semantics of the equivalence class model, we must redefine the *sp* semantics of the assignment statement for simple (non-pointer) variables to be consistent with the model. Given a statement "`x := e`" and a precondition Q where `x` is a non-pointer variable and `e` is an expression, the strongest postcondition for assignment statements is:

$$sp(\texttt{x := e}, Q) \equiv (\exists v :: Q_v^{x.V} \wedge x.V = \tilde{e}_v^{x.V}),$$

which states that after the execution of "`x := e`" there exists some value v such that every free occurrence of $x.V$ in Q is replaced with v and $x.V = \tilde{e}_v^x$. The notation $\tilde{e}$ indicates that the expression *e* is transformed so that every simple variable `u` that is a term in *e* is replaced by $u.V$, and every pointer dereference `*p` that is a term in *e* is replaced by $coset(p).V$, where $coset(p).V$ refers to the value of the object identified by $coset(p)$. This formalization ensures that there is a consistent notation for referring to the values of data objects.

3.6. Heap Allocation

When a pointer is assigned a "value", that pointer is placed into an equivalence class such that all the members of the equivalence class point to the same data object. One method for assigning a value to a pointer is through heap memory allocation. Heap memory allocation has the form "`p := new T`" where `p` is a pointer and `T` is a data type. Informally, upon allocation of heap memory, a new data object of type `T` is created and the pointer `p` is used to reference the object. In our model, this action is represented by introducing a new entry o in N with an undefined value in V, and adding p to the equivalence class $M[o]$. In addition, if p was previously in some equivalence class $M[k]$, it is removed from that set. Formally we can state this condition as follows:

$$sp(\texttt{p := new T}, Q) \equiv \quad (\exists c : c \in N : Q_c^p) \wedge \quad p \gtrdot o \wedge o.V = \mathit{undefined}, \qquad (2)$$

where o is a new data object. The textual substitution of every free occurrence of p in Q with the term $c \in N$ ensures that p is removed from any equivalence class that it may have previously been associated, and the assertion $p \gtrdot o$ places p into the equivalence class $M[o]$. Finally, the term $o.V = \mathit{undefined}$ asserts that the value of the new object o is undefined. As an example, let precondition Q be $\{q \gtrdot o1\}$ and statement S be "`q := new T`". Then

$sp(S, Q) \equiv sp(\texttt{q := new T}, q \gtrdot o1)$

⟨Expression (2)⟩

$\equiv (\exists c : c \in N : (q \gtrdot o1)_c^q) \wedge q \gtrdot o2 \wedge o2.V = \mathit{undefined}$

⟨Textual substitution of q with c⟩

$\equiv (\exists c : c \in N : c \gtrdot o1) \wedge q \gtrdot o2 \wedge o2.V = \mathit{undefined}$

⟨Points-to axiom (Figure 5) applied to $c \gtrdot o1$⟩

$\equiv (\exists c : c \in N : c = o1) \wedge q \gtrdot o2 \wedge o2.V = \mathit{undefined}$

⟨Trading [5].⟩

$\equiv (\exists c : c = o1 : c \in N) \wedge q \gtrdot o2 \wedge o2.V = \mathit{undefined}$

⟨One-point rule [5] with $c = o1$, $o1 \in N \equiv \mathit{true}$⟩

$\equiv q \gtrdot o2 \wedge o2.V = \mathit{undefined}$

As such, after the execution of the statement "`q := new T`", the pointer q points to some new object $o2$.

3.7. Pointer Assignment

Another way of assigning a value to a pointer is via direct aliasing as in the C-like command "`p := @x`". In terms of the equivalence class model, the pointer alias assignment adds the pointer `p` to the equivalence class $M[x]$. The formal semantics of this command is similar to the heap allocation case. Formally the semantics is as follows:

$$sp(\texttt{p := @x}, Q) \equiv (\exists c : c \in N : Q_c^p) \wedge p \gtrdot x. \qquad (3)$$

Expression (3) states that after executing the statement `p := @x` that $p \gtrdot x$ and that every free occurrence of p in Q is replaced with c. This relationship ensures that the pointer p is placed into the equivalence class associated to the variable x.

The final way that a pointer can be assigned a value occurs when a statement of the form "`p := q`" is executed, where `p` and `q` are pointers. In terms of the equivalence class model, the pointer assignment adds the pointer `p` to the class that contains pointer `q`. In this case the formal semantics is expressed as

$$sp(\texttt{p := q}, Q) \equiv (\exists c : c \in N : Q_c^p) \wedge p \gtrdot coset(q). \qquad (4)$$

Expression (4) states that after execution of the pointer assignment, p points to the object $coset(q)$. For example, let statement S be "`p := q`" and precondition Q be "$\{q \gtrdot o1 \wedge p \gtrdot o2\}$". Then

$sp(S, Q) \equiv sp(\texttt{p := q}, q \gtrdot o1 \wedge p \gtrdot o2)$

⟨Expression (4)⟩

$\equiv (\exists c : c \in N : (q \gtrdot o1 \wedge p \gtrdot o2)_c^p) \wedge p \gtrdot coset(q)$

⟨Textual substitution of p with c⟩

$\equiv (\exists c : c \in N : (q \gtrdot o1 \wedge c \gtrdot o2)) \wedge p \gtrdot coset(q)$

⟨Points-to axiom applied to $c \gtrdot o2$)⟩

$\equiv (\exists c : c \in N : (q \gtrdot o1 \wedge c = o2)) \wedge p \gtrdot coset(q)$

⟨Trading⟩

$\equiv (\exists c : c = o2 : c \in N \wedge q \gtrdot o1) \wedge p \gtrdot coset(q)$

⟨One-point rule with $c = o2$, $o2 \in N \equiv \mathit{true}$ ⟩

$\equiv q \gtrdot o1 \wedge p \gtrdot coset(q)$

⟨Definition of coset⟩

$\equiv q \gtrdot o1 \wedge p \gtrdot o1$

As such, after the execution of "`p := q`", the pointers p and q reference the same data object.

3.8. Value Assignment

In the C programming language, the value of the data object that a pointer references is accessed using the notation "`*p`", where `p` is a pointer variable. Using the same notation convention, an assignment to the data object is achieved using a command of the form "`*p := e`", where `e` is an expression. In terms of the equivalence class model, the assignment of `*p` sets the value of the data object $coset(p)$ to `e`. Formally, the semantics of assignment to a dereferenced data object is as follows:

$$sp(\texttt{*p := e}, Q) \equiv \quad (\exists v : v \in T : Q_v^{coset(p).V} \wedge coset(p).V = \tilde{e}_v^{coset(p).V}) \tag{5}$$

where T is the type of the data object, and v is a value of that type. For instance, if T is the type *integer*, then v is some integer. The variable v represents the value of the data object dereferenced by `*p` prior to the execution of the statement "`*p := e`". Informally, the semantics states that after execution of the statement "`*p := e`", `*p` will have the value $\tilde{e}_v^{coset(p).V}$. Additionally, $Q_v^{coset(p).V}$ will be true.

3.9. Value Dereference

The command for observing the value of a pointer dereference has the form "`x := *p`", where `x` is a variable and `p` is a pointer. In terms of the equivalence class model, the value dereference `*p` refers to the value of the data object associated to the equivalence class containing `p`. That is, `*p` refers to $coset(\mathtt{p}).V$. The formal semantics of a value dereference is as follows:

$$sp(\texttt{x := *p}, Q) \equiv (\exists v : v \in T : Q_v^{x.V} \wedge x.V = coset(p).V) \tag{6}$$

where T is the type of the data object, and v is a value of that type. Informally, Expression (6) states that after the execution of a statement with `*p` on the right hand side of an assignment, the left hand side of the assignment takes on the value of the object that has been dereferenced. The term $Q_v^{x.V}$ states that every free occurrence of $x.V$ in Q is replaced with the value of x previous to executing the statement "`x := *p`". As an example, let statement S be "`x := p`" and let precondition Q be $\{p \gtrdot o1 \wedge o1.V = n \wedge x.V = y\}$. The *sp* derivation proceeds as follows.

$sp(S, Q) \equiv sp(\texttt{x := *p}, p \gtrdot o1 \wedge o1.V = n \wedge x.V = y)$

⟨Expression (6)⟩

$\equiv (\exists v : v \in T : (p \gtrdot o1 \wedge o1.V = n \wedge x.V = y)_v^{x.V} \wedge x.V = coset(p).V)$

⟨Textual substitution of $x.V$ with v⟩

$\equiv (\exists v : v \in T : p \gtrdot o1 \wedge o1.V = n \wedge v = y \wedge x.V = coset(p).V)$

⟨Trading⟩

$\equiv (\exists v : v \in T \wedge v = y : p \gtrdot o1 \wedge o1.V = n \wedge v = y \wedge x.V = coset(p).V)$

⟨One-point rule with $v = y$⟩

$\equiv p \gtrdot o1 \wedge o1.V = n \wedge x.V = coset(p).V)$

⟨Definition of coset⟩

$\equiv p \gtrdot o1 \wedge o1.V = n \wedge x.V = o1.V)$

This states that the new value of $x.V$ is equivalent to the value of the data object $o1$.

4. Pointer Examples

AUTOSPEC is a tool that has been developed to support the use of strongest postcondition in the construction of formal specifications from existing program code [6]. In this section we describe how AUTOSPEC has been used to facilitate the analysis of programs.

AUTOSPEC accepts programs as input and using semantic based rules, such as the ones found in Figure 1, derives a formal specification of the input program. Our current investigations include extending the AUTOSPEC tool to support the formal strongest postcondition semantics of the C programming language as described in [7]. For statements such as assignments and conditionals, AUTOSPEC is fully automated. When processing loops, AUTOSPEC allows a user to provide appropriate preconditions and postconditions. In addition to supporting user guidance during the specification process, AUTOSPEC supports syntactic and semantic verification of the input supplied by users by using an integrated syntax checker and theorem prover.

Figure 7 contains two programs for illustrating the pointer semantics described in this paper as well as for showing the use of an automated tool for analyzing programs with pointers. Figure 7(a) is a program for demonstrating how aliases are resolved using the pointer semantics, and Figure 7(b) shows a program with a conditional statement and how the conditional statement impacts pointer resolution. The specifications in this section were all automatically generated by the AUTOSPEC tool.

4.1. `manyvars`

The `manyvars` program is shown in Figure 7(a). This program demonstrates the difficulty in understanding programs that use a high degree of aliasing. The program uses two integer variables and two pointer variables, where the pointers `r` and `q` are used to create aliases of variables `u` and `z`, respectively. In addition, a number of value assignments are made to the primary variables (e.g., `z := 0;`) and aliases (e.g., `*r := 1;`). Figure 8 contains the specification of the `manyvars` program as generated by the AUTOSPEC system. The first statement of the program at line `11` (`r := @u`) creates an alias of variable `u` and is specified by the conjunct `(r .> u)` in the expression:

```
program manyvars()
decl
   int z; int u; int *r;
   int *q;
lced
begin
   r := @u;
   z := 0;
   q := @z;
   *r := 1;
   *q := *r;
end
```

(a)

```
program maxThresh(
   inputs: int e; int x; int y;
   outputs: int *z;
)

begin

   if ( x > y ) -> z := @x;
   || ( x <= y ) -> z := @y;
   fi;
   *z := *z + e;

end
```

(b)

Figure 7. Two Sample Programs: (a) `manyvars` **(b)** `maxThresh`

```
(((u.V = u_0) & (z.V = z_0)) & (r .> u))
```

at line `12`. The fourth statement in the program at line `19` (`*r := 1`) assigns the value "1" to the data object identified by *coset*(`r`) which, due to the conjunct `(r .> u)`, is the data variable `u`. Hence, the conjunct `(u.V = 1)` appears in the specification at lines `20-22`.

```
1    program manyvars (
2    )
3    decl
4       int z;
5       int u;
6       int *r;
7       int *q;
8    lced
9    begin
10      { ((u.V = u_0) & (z.V = z_0)) }
11      r := @u;
12      { (((u.V = u_0) & (z.V = z_0)) & (r .> u)) }
13      z := 0;
14      { ((((u.V = u_0) & (_cnst2 = z_0)) & (r .> u)) &
15         (z.V = 0)) }
16      q := @z;
17      { (((((u.V = u_0) & (_cnst2 = z_0)) & (r .> u))
18         & (z.V = 0)) & (q .> z)) }
19      *r := 1;
20      { ((((((_cnst5 = u_0) & (_cnst2 = z_0)) &
21         (r .> u)) & (z.V = 0)) & (q .> z)) &
22          (u.V = 1)) }
23      *q := *r;
24      { (((((((_cnst5 = u_0) & (_cnst2 = z_0)) &
25         (r .> u)) & (_cnst7 = 0)) & (q .> z)) &
26          (u.V = 1)) & (z.V = coset( r ).V)) }
27   end
```

Figure 8. AUTOSPEC applied to `manyvars`

The final specification of the `manyvars` program (lines `24-26`) is the following:

```
(((((((_cnst5 = u_0) & (_cnst2 = z_0)) & (r
.> u)) & (_cnst7 = 0)) & (q .> z)) & (u.V =
1)) & (z.V = coset( r ).V))
```

which states that after the execution of the program, `z.V` has a value equivalent to that of `coset(r).V`. Since *coset*(`r`) = `u`, and `(u.V = 1)`, the value of variable `z`, denoted `z.V`, is `1`.

4.2. `maxThresh`

The `maxThresh` program is shown in Figure 7(b). The purpose of this program is to demonstrate the use of AUTOSPEC for specifying the cases where pointers may reference many objects rather than just a single object. The `maxThresh` program sets pointer `z` to alias the maximum of two input variables `x` and `y`. After determining the maximum, the program adds a threshold `e` to the maximum.

Figure 9 contains the specification of the `maxThresh` program as generated by the AUTOSPEC system. After the execution of the `if-fi` statement (lines `11-22`), the following (as shown in lines `23-26`) is true:

```
((((x.V > y.V) & ((y.V = y_0) & ((x.V = x_0) &
(e.V = e_0)))) & (z .> x)) | (((x.V <= y.V) &
((y.V = y_0) & ((x.V = x_0) & (e.V = e_0)))) & (z
.> y)))
```

which states that the value of variable `x` is greater than the value of variable `y` and the pointer `z` points to the variable `x`, or the value of variable `y` is greater or equal to the value of variable `x` and and the pointer `z` points to the variable `y`.

```
1    program maxThresh (
2    inputs :
3       int e;
4       int x;
5       int y;
6    outputs :
7       int *z;
8    )
9    begin
10      { ((y.V = y_0) & ((x.V = x_0) & (e.V = e_0))) }
11      if
12         (x > y) ->
13         z := @x;
14         { (((x.V > y.V) & ((y.V = y_0) &
15            ((x.V = x_0) & (e.V = e_0)))) &
16             (z .> x)) }
17      || (x <= y) ->
18         z := @y;
19         { (((x.V <= y.V) & ((y.V = y_0) &
20            ((x.V = x_0) & (e.V = e_0)))) &
21             (z .> y)) }
22      fi;
23      { ((((x.V > y.V) & ((y.V = y_0) & ((x.V = x_0)
24         & (e.V = e_0)))) & (z .> x)) |
25          (((x.V <= y.V) & ((y.V = y_0) & ((x.V = x_0)
26           & (e.V = e_0)))) & (z .> y))) }
27      *z := (*z + e);
28      { (((((((x.V > y.V) & ((_cnst4 = y_0) &
29         ((_cnst5 = x_0) & (e.V = e_0)))) &
30          (z .> x)) |
31           (((x.V <= y.V) & ((_cnst4 = y_0) &
32            ((_cnst5 = x_0) & (e.V = e_0)))) &
33             (z .> y))) & ((_cnst4 = CV3) |
34              (_cnst5 = CV3))) & (CV3 = _o6)) &
35               (coset( z ).V = (_o6 + e.V))) }
36   end
```

Figure 9. AUTOSPEC applied to `maxThresh`

The final specification of the `maxThresh` program is shown in lines `28-35` of Figure 9. The specification states that the value of `coset(z).V` is equivalent to the expression `(_o6 + e.V)` where `_o6 = CV3` and `((_cnst4 = CV3) | (_cnst5 = CV3))`. Since the pointer `z`

can point to either variable x or variable y, the value (*z + e) in the statement at line 27 is dependent on the result of the if-fi statement. Hence, the specification states that CV3 can take the value _cnst4 or _cnst5.

5. Related Work

Many techniques have been suggested for addressing reverse engineering and program understanding including plan abstraction [11], formal program transformation [14], and structural abstraction [13]. Although these approaches support automated program understanding, formal treatment of pointers is either not addressed or is addressed at a level of granularity that may not reflect the actual behavior of the program.

The model of analysis involving the dynamic equivalence classes and the points-to relation used in this paper is similar to the points-to analysis models of [12] in that we capture the relationship between a pointer and a data object. With respect to the alias detection techniques, our approach is more straightforward since we focus on program semantics and not alias detection. That is, much of the alias detection research to date focuses on the development of algorithms for detecting aliases and, subsequently, use more complex structures for analyzing pointer constructs. Our approach addresses the issues related to program semantics and the development of algorithms for deriving the behavior of programs.

6. Conclusion

Formal methods provide many benefits in the development of software. Automating the process of abstracting formal specifications from program code is sought but, unfortunately, not completely realizable. However, by providing the tools that support the reverse engineering of software, much can be learned about the functionality of a system. In this paper we described the strongest postcondition semantics of pointer operations using a dynamic equivalence class model. Landi proved that may-alias pointer detection is undecidable and must-alias pointer detection is uncomputable [10]. Our intentions are not to provide a means for automatically deriving specifications of programs with loops and pointers but rather to provide tool support for users to derive those specifications.

Previously we investigated the application of the strongest postcondition to define the semantics of a subset of the C programming language. We are currently extending AUTOSPEC to support the semantic definition of C. This includes extending the tool to support the use of the pointer semantics described in this paper. In addition, we are investigating techniques for deriving abstractions of formal specifications that have been constructed using strongest postcondition. Finally, we have applied the use of our techniques to a mission control system for controlling robotic spacecraft at the NASA Jet Propulsion Laboratory [7].

References

[1] E. Byrne. A Conceptual Foundation for Software Reengineering. In *Proceedings for the Conference on Software Maintenance*, pages 226–235. IEEE, 1992.

[2] B. H. C. Cheng. Applying formal methods in automated software engineering. *Journal of Computer and Software Engineering*, 2(2):137–164, 1994.

[3] E. J. Chikofsky and J. H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[4] R. Covington, editor. *Formal Methods Specification and Verification Guidebook for Software and Computer Systems; Volume 1: Planning and Technology Insertion*, volume NASA-GB-002-95. National Aeronautics and Space Administration, July 1995.

[5] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[6] G. C. Gannod and B. H. C. Cheng. Strongest Postcondition as the Formal Basis for Reverse Engineering. *Journal of Automated Software Engineering*, 3(1/2):139–164, June 1996.

[7] G. C. Gannod and B. H. C. Cheng. Using Informal and Formal Methods for the Reverse Engineering of C Programs. In *Proceedings of the 1996 Internation Conference on Software Maintenance*, pages 265–274. IEEE, 1996.

[8] G. C. Gannod and B. H. C. Cheng. A Formal Automated Approach for Reverse Engineering Programs with Pointers. Technical Report MSU-CPS-97-19, Michigan State University, June 1997.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[10] W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[11] A. Quilici. A Memory-Based Approach to Recognizing Program Plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[12] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN Symposium on the Principles of Progamming Languages*, 1996.

[13] S. R. Tilley, K. Wong, M.-A. Storey, and H. A. Müller. Programmable Reverse Engineering. *The International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[14] M. Ward, F. Calliss, and M. Munro. The Maintainer's Assistant. In *Proceedings for the Conference on Software Maintenance*. IEEE, 1989.

[15] J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.