PROPER TECHNIQUE OF SOFTWARE INSPECTION USING GUARDED COMMAND LANGUAGE

Neeraj Kumar
Singhania University, Rajasthan, India, E-mail: neerajocp@gmail.com.

ABSTRACT

Michael Fagan introduced the software inspection process in the early 1970s. The Fagan inspection method, a formalization of review process is based on the application of process management techniques to software development. To generalize software inspections are performed with the intent of finding and correcting defects by identifying situations where a software element deviates from the requirement specification. In this paper we discuss techniques based on strongest post-condition predicates transformer (sp). We identify problems with other formal approaches for deriving semantics to support reasoning. We have described verification methods based on the derived semantics forms and conjecture that because the need to provide inference rules for language constructs has been removed, these techniques may be more amenable to automated and semi automated theorem proving than traditional approaches.

Keywords: Software Inspection, Guarded Command Language, Strongest Post-condition.

1. INTRODUCTION

One of the first practicable attempts in capturing mathematically the behaviour of a computer program has been made by E.W. Dijkstra in his famous work A discipline of programming. He formally defines syntax and semantics of his model programming language, the Guarded Command Language. One of the major benefits of his approach was that he could deal effectively with the problem of non-termination within his program semantics.

The meaning (or semantics) for each construct of his Guarded Command Language is given in terms of the weakest precondition to establish an arbitrary chosen post-condition Q. The degree of weakness is defined by the set of program states which a condition encompasses. A condition is considered to be weaker if it is satisfied by more program states.

Construct of the Guarded Command Language

\[ wp \left( \text{skip} \right) = df Q \]

The command that doesn’t change the program state is called \text{skip}. It always terminates and its weakest precondition is trivially equivalent to the post-condition Q.

\[ wp \left( \text{abort} \right) = df Q \]

The command that never ensures termination is called \text{abort}. It can generally lead to any behaviour but moreover, just refuse to terminate. In conclusion, It could never be guaranteed to establish any such Q we may ask for. Hence, its weakest precondition can never be satisfied by any program state.

\[ wp \left( \text{x} = E \right) = df Q <x \setminus E> \]

The assignment construct assigns to a variable \text{x} the value of an arbitrary expression \text{E}. Its weakest precondition is derived by substituting the variable \text{x} by the expression \text{E} in the post-condition Q.

\[ wp \left( \text{x}_1, \text{x}_2, \ldots, \text{x}_n : = E_1, E_2, E_3, \ldots, En \right) = df Q \setminus \text{x}_1, \text{x}_2, \ldots, \text{x}_n \setminus E_1, E_2, E_3, \ldots, En > \]

Instead of assigning just one variable at a time, the concurrent assignment allows to assign a list of variables simultaneously to a list of expressions. Analogously, this is translated into simultaneous substitution of all variables in the post-condition Q.

\[ wp \left( S; T \right) = df wp \left( S, wp \left( T, Q \right) \right) \]

The sequential composition command captures the idea of executing two commands in sequential order. The weakest precondition for the command sequence S; T to establish a post-condition Q is the weakest precondition for S to establish wp (T, Q), the weakest precondition for T to establish Q. For more we can refer A discipline of programming.

2. THE STRONGEST POST-CONDITION PREDICATE TRANSFORMER

In order to verify a program we must show that it satisfies its specification and terminates. That is, we must show that a program S, executed under some precondition Q terminates and satisfies a post-condition R. A program may be verified by either forward or backward reasoning by applying predicate transformers to the specifications and program [80].
Weakest Preconditions: The weakest precondition predicate transformer, \( wp (S, R) \), provides the weakest definition of the set of all initial states under which the program \( S \) can execute and terminate and guarantee that some post-condition \( R \) will hold. It is used in the “backwards reasoning” approach to program verification by showing that where we have \( \{Q\} S \{R\} \) \( wp (S, R) \). Dijkstra [2] provides rules for calculating the weakest precondition of a program constructed from assignment, selection and iteration constructs. Traditionally, weakest precondition calculations have provided a strong basis for the derivation of programs from a specification using a process of stepwise refinement, and have also been used as a basis for formal proof.

Strongest Post-conditions: The strongest post-condition transformer, \( sp (Q, S) \), gives the strongest assertion \( R \) that holds given that \( S \) executes and terminates under a precondition satisfying \( Q \). Strongest post-conditions, therefore, specify the strongest assertions that hold at each point in the execution of the program \( S \). A program satisfies it’s specification when \( sp (Q, S) \) \( R \). Dijkstra [2] provides rules for computing the strongest post-condition for a program consisting of assignment, selection and iteration constructs. This thesis redefines and uses these rules as a basis for paraphrasing code in terms of a complete, unambiguous first-order specification of its semantics.

3. STRONGEST POST-CONDITION FOR ASSIGNMENT

Dijkstra [2] defines the strongest post-condition for assignment as:

1. \( sp (Q, x := E) \equiv (x = E) \land 3 x (Q), \) and
2. \( sp (Q, x := E (x)) \equiv Q [E^{-1}(x)/x]. \)

The first definition, used when \( x \) is assigned an expression which is not a function of \( x \), makes direct calculation difficult due to the term \( 3 x (Q) \) only indicating that \( x \) is bound by \( Q \) Pan [5]. The second definition may make calculation impossible in many situations as the inverse function may not be defined or may be difficult to capture \( (x := 3x^2 - 4x^2 + 2). \)

Pan [5] suggested the introduction of fresh variables to the problem in order to remove the requirement to calculate inverse functions, by effectively removing any assignments of the second form. This approach highlights quality defects in code by identifying situations where a single variable is used for more than one purpose. By Pan’s approach, introducing the new variable \( t \), the calculation \( sp (x = X, x := 3x^2 - 4x^2 + 2) \) becomes

\[ sp (x = X, t := 3x^2 - 4x^2 + 2) \equiv sp (t = X, x := 3x^2 - 4x^2 + 2). \]

This allows us to describe the semantics of the assignment statement at the cost of introducing variables that are not part of the original program variable set, and which can not be removed without applying the inverse function that we are trying to avoid. Because it is the aim of this thesis to present a semantic description of a unit of code to be used to assist in the inspection of the code, we feel it may be confusing to introduce variables where-ever a self-assignment is made. To this end, we provide an alternate definition of the strongest post-condition for assignment which is equivalent to the two calculations defined by Dijkstra. This definition removes the necessity to calculate inverse functions, is repeatable, and does not rely on the introduction of new variables to the problem.

4. THE MODIFIED STRONGEST POST-CONDITION CALCULATION

We start with the conjecture that in any program written in an imperative language that allows self-assignment, a variable must have been initialised with a value before being used on the right hand side of an assignment. If a variable has not been initialised, prior to it being used on the right hand side of an assignment, then we can say that the assignment in question is non-deterministic.

For example, if we consider the assignment \( x := x^2 \) under the precondition true, we can make no reasonable guess at what \( x \) might be in the post-condition. The value of \( x \) is determined either by the value that happens to be stored at the particular memory location allocated for \( x \), or the language default for variables of the type of \( x \). We denote this non-determinable state of \( x \) by \( x^0 \). The definition for the strongest post-condition for assignment provided here uses the conjecture that all variables must be initialized with a value, either explicitly by the program or implicitly by the environment, prior to their use in an expression.

5. ALGORITHM 1 SP FOR ASSIGNMENT

Consider the assignment \( x := E \), under some precondition \( Q \) where \( x \) is a variable, \( E \) is a term, and \( Q \) is a formula of the form \( Q_1 V ... V Q_n \) where the sub-formulas \( Q_1, ..., Q_n \) contain no disjuncts.

\( sp (Q, x := E) = sp (Q_1, x := E) V ... V sp (Q_n, x := E) \),

where for all \( i \in \{1..n\} \),

\( sp (Q_i, x := E) \) is defined as:

1. If an \( xC \)-equality occurs in \( Q_i \), let the constant symbol occurring in the leftmost \( xC \)-equality of \( Q_i \) be \( D \), then \( sp (Q_i, x := E) = Q_i [D/x] \land x = E [D/x] \).
2. If an \( xC \)-equality does not occur in \( Q_i \), and an \( xv \)-equality occurs in \( Q_i \), let the non-\( x \) variable occurring in the leftmost \( xv \)-equality of \( Q_i \) be \( w \), then \( sp (Q_i, x := E) = Q_i [w/x] \land x = E [w/x] \).
3. If an \( xC \)-equality does not occur in \( Q_i \), and an \( xv \)-equality does not occur in \( Q_i \), and an \( xf \)-equality occurs in \( Q_i \), let the non-\( x \) term occurring in the leftmost \( xf \)-equality of \( Q_i \) be \( u \), then \( sp (Q_i, x := E) = Q_i [u/x] \land x = E [u/x] \).
4. Otherwise, \( x \) is not defined in the scope of the assignment prior to use, and \( sp \) (\( Qi; x := E \)) = \( Qi \) \[ x0/x] \( \Lambda x = E \) \[ x0/x].

Within a program we may establish a number of equalities between a variable and other variables, constants or functions of constants or variables. In order to make the strongest post-condition calculation deterministic, we distinguish between the following types of variable equalities:

**Definition 1:** \( xc \)-equality: An \( xc \) equality is any atom of the form \( x = C \) or \( C = x \) where \( x \) is a variable and \( C \) is a constant symbol.

**Definition 2:** \( xv \)-equality: An \( xv \) equality is any atom of the form \( x = v \) or \( v = x \) where \( x \) and \( v \) are variables.

**Definition 3:** \( xf \)-equality: An \( xf \) equality is any atom of the form \( x = f(t1, t2, ..., tn) \) or \( f(t1, t2, ..., tn) = x \) where \( x \) is a variable and \( f \) is an \( n \)-ary function symbol and \( t1, t2, ..., tn \) are terms.

The new sp calculation for assignment is described by Algorithm (1).

**Example:** Calculating sp for assignment

(i) \( sp \) (\( x = a \) \( \Lambda z = y \) \( x := y \))

\[ \equiv (x = a \Lambda z = y \Lambda x = y) \Lambda x = y \Lambda [a/x] \]

\[ \equiv (a = a \Lambda z = y \Lambda x = z \Lambda x = y) \]

\[ \equiv z = y \Lambda x = y \]

(ii) \( sp \) (\( x = X \) \( x := 3x^2 - 4x^2 + 2 \))

\[ \equiv (x = X) \Lambda x = (3x^6 - 4x^2 + 2) \Lambda [X/x] \]

\[ \equiv (X = X) \Lambda x = (3X^6 - 4X^2 + 2) \]

\[ \equiv x = (3X^6 - 4X^2 + 2) \]

As a side effect of the calculation process we may achieve automatic recognition of defects that relate to the use of an unutilized variable. If the result of the sp calculation involves any references to \( v \) \( v \) for a variable \( v \) then \( v \) has been used prior to being initialized in the scope of the program segment being analyzed.

There are a number of reasons why this may occur:

- If \( v \) is declared locally, within the scope of the program segment being analyzed, then \( v \) \( v \) indicates that \( v \) has not been initialized prior to use.

This is a defect that affects portability, maintainability and reusability.

- If \( v \) is not declared locally then \( v \) \( v \) indicates that \( v \) may be either a global variable, or a class variable (in oo programs).

The identification and reporting of such semantics is useful to assist code readers in identifying defects related to variable initialization.

It should be noted that due to the treatment given to sub-programs in Chapter 8, the precondition prior to the execution of the body of a procedure or function includes an \( xv \)-equality of the form \( v = v \)? for all formal parameters \( v \). This ensures that formal parameters are not considered uninitialized.

**Lemma:** For a simple assignment of the form \( x := ax + b \) under some precondition \( Q \), where \( x \) is a variable, and \( a \) and \( b \) are constants \( a \neq 0 \), \( sp \) (\( Q, x := ax + b \)) = \( Q \left[ \frac{x - b}{a} \right] / x \).

**Proof:** For the assignment \( x := ax + b \) to occur correctly in an imperative program, \( Q \) must be of the form \( (x = t) \Lambda P \), where \( t \) is a term, otherwise \( x \) is uninitialised prior to its use in the assignment, and the assignment itself is defective.

\[ LHS = sp \) (\( Q, x := ax + b \)) \]

\[ = sp \) ((\( x = t) \Lambda P, x := ax + b \)) \]

\[ = ((x = t) \Lambda P) \left[t/x \right] \Lambda x = (ax + b) \left[t/x \right] \]

\[ = ((t = t) \Lambda P \left[t/x \right]) \Lambda x = (at + b) \]

\[ = P \left[t/x \right] \Lambda x = (at + b) \]

\[ RHS = Q \left[ \frac{x - b}{a} \right] / x \]

\[ = ((x = t) \Lambda P \left[t/x \right]) \Lambda x = (at + b) \]

\[ = P \left[t/x \right] \Lambda x = (at + b) \]

\[ = LHS \]

**6. sp FOR SEQUENCE ELEMENT ASSIGNMENT**

For an assignment of the form \( a [exp] := E \), executing under the precondition \( Q \), where \( exp \) and \( E \) are expressions and \( a \) is a sequence data type, the above definition of strongest post-condition for assignment and the substitution function can be extended to describe the changes to \( a \) as well as the fact that the remainder of \( a \) is unchanged by the assignment. We use the notation \( a = A \), where \( a \) and \( A \) are sequence variables, to denote the relationship \( \forall i \in [0..\#a - 1] (a [i] = A [i]) \). That is, writing \( a = A \) is equivalent to \( [a [0] = A [0] \Lambda a [1] = A [1] \Lambda \ldots \Lambda a [\#a - 1] = A [\#a - 1]) \).

The semantics of the substitution \( F [exp/a [e]] \), where \( F \) is a term or formula, \( a \) is a sequence variable and \( e \) and \( exp \) are expressions, can be obtained by application of the following rules:
1. Evaluate $e$ and denote the result as $e_0$.
2. Make the substitution $F[exp\ [e_0]/a\ [e_0]]$.

This substitution function, when applied to Algorithm 1, yields the following definition for the strongest post-condition for assignment to a sequence element.

Let $Q = (P \land a = A) \ (A \text{ may be } a\ 0\text{ if } a\text{ is uninitialized})$ and let $exp\ 0$ denote the evaluation of the expression $exp$.

Then

$$sp\ (Q, a\ \{exp\} := E) = P\ [A\ [exp]/a\ [exp]] \land (a = A) \ [A\ [exp]/a\ [exp]] \land a\ [exp] = E\ [A\ [exp]/a\ [exp]]$$

For example, the strongest post-condition semantics of the assignment $a\ [i] := x$ under the precondition $(a = a; \ \Lambda i = i; \ \Lambda v = v?)$, where $a$ and $a0$ are sequences and $i$ is a natural number, is calculated by (3.1) as follows:

$$sp\ ((a = a; \ \Lambda i = i; \ \Lambda v = v?), a\ [i] := v) = P\ [A\ [exp]/a\ [exp]] \land (i = i) \ \Lambda i = i; \ \Lambda v = v?; \ \Lambda a\ [i] = v\ [a?\ [i] / a\ [i]])$$

$...\ (3.1)$

7. STRONGEST POST-CONDITION FOR SELECTION

Dijkstra [2] defines the strongest post-condition for a selection statement, if $C_1 \rightarrow S_1 \ [\ldots] C_n \rightarrow S_n \ fi_1$ under the precondition $Q$, where each $Ci$ is a boolean expression and each $Si$ is a statement as:

$$sp\ (Q, C_1 \rightarrow S_1 \ [\ldots] C_n \rightarrow S_n \ fi_1) = sp\ (Q \land C_1, S_1) \ V \ldots \ V \ sp\ (Q \land C_n, S_n)$$

Example: Calculating $sp$ for selection

$$sp\ (N \geq 0 \land a = x, a \land N \rightarrow a := a + 1 \ [a < N \rightarrow a := a + a\ [a + 1]])$$

$= sp\ (N \geq 0 \land a = x \land a \geq N, a := a + 1) = sp\ (N \geq 0 \land a = x \land a < N, a := a + a)$

$= (N \geq 0 \land a = x + 1 \land x \geq N) \ V (N \geq 0 \land a = 2x \land x < N)$

8. STRONGEST POST-CONDITION FOR SEQUENCE

Dijkstra [2] defines the strongest post-condition for sequential statements, $S_1; S_2$, under the precondition $Q$ as: $sp\ (Q, S_1; S_2) = sp\ (sp\ (Q, S_1), S_2)$.

Example: Calculating $sp$ for sequence $sp\ (True, b := 1; c := 2 * b)$

$= sp\ (sp\ (True, b := 1), c := 2 * b)$

$= sp\ (True \land b = 1, c := 2 * b)$

$= (True \land b = 1, c = 2b)$

$= (b = 1 \land c = 2b)$

9. STRONGEST POST-CONDITION FOR ITERATION

Back [6] and Dijkstra [2] define the strongest post-condition for the iteration do $G \rightarrow S \ od$, under the precondition $Q$, where $G$ is a boolean expression and $S$ is a statement as:

$$sp\ (Q, do G \rightarrow S \ od) = \neg G \land (Q \land P_1 \land \ldots \land P_m)$$

Where $m$ is a positive integer representing the number of iterations, and $P_1 = sp\ (Q \land G, S)$ and $P_i + 1 = sp\ (P_i \land G, S)$ for $0 < i \leq m$.

The application of this definition is difficult, if not impossible, in practice, because, although the calculation depth, $m$, is bound, it is not fixed. This stops us from deriving the strongest post-condition for a loop unless we know in advance how many times the loop will iterate. The net effect of this is that we can not use this definition of strongest post-condition calculation to derive a specification of what a loop is doing. This impedes our verification efforts. Pan [5] introduces a process to obtain the Strongest Accessible Conjunctive Post-condition, in which the calculation depth depends only on the loop body and the loop precondition, and is therefore fixed. This method is limited in its application to only a small subset of simple loops.

10. CONCLUSION

In this paper we discuss definitions for the strongest post-condition semantics of imperative programming language constructs.

This paper makes contribution in that it describes an algorithm for constructing the strongest post-condition specification from an assignment statement, without relying on the calculation of inverse functions or on the introduction of new variables to the problem, as is the case with previous definitions. We also discuss the problems with existing definition of strongest post-condition for iterative constructs and procedure calls.

This paper also introduces a notational aid called the iterative-form notation. This notation is the foundation for the future work in which we present an operational definition for calculating an iterative-form invariant for iterative program constructs.
REFERENCES