International Journal of Information Technology and Knowledge Management (ISSN: 0973-4414)
July-December 2012, Volume 5, No. 2, pp. 493-496

Novel approach for Software Inspection

Dinesh Kumar, Vijay Kumar
M.Tech. student, Department of Computer Science, Rajasthan Technical University, Kota
Research Scholar, Singhania University, Rajasthan, India
dinesh_matwa@yahoo.co.in, vijay_matwa@yahoo.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. 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

INTRODUCTION

First practicable attempts in capturing mathematically the behavior 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(**skip**,Q)=df Q

The command that doesn't change the program state is called **skip**. It always terminates and its weakest precondition is trivially equivalent to the post-condition Q.

wp(**abort**,Q)=df Q

The command that never ensures termination is called **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(x : = E, Q) = df Q<x\E>

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.

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}, Q) 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 iteration2 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.

International Journal of Information Technology and Knowledge Management (ISSN: 0973-4414)
July-December 2012, Volume 5, No. 2, pp. 493-496

Strongest Post-condition for Assignment

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

1. sp(Q, x := E) ≡ (x = E) Λ ∃x (Q), and
2. sp(Q, x := E(x)) ≡ 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 ∃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 := $3x^6 - 4x^2 + 2$)

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 := $3x^6 - 4x^2 + 2$) becomes

sp ((x = X) [t/x], x := $(3x^6 - 4x^2 + 2)[t/x]$) ≡ sp(t = X, x := $(3t^6 - 4t^2 + 2)$) ≡ (t = X Λ x = $3t^6 - 4t^2 + 2$)

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.

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: = x2 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 x0. 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.

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 Q1 V ... V Qn where the sub formulas Q1, . . . ,Qn contain no disjuncts.

sp(Q, x := E) = sp(Q1, x := E) V .. V sp(Qn, x := E), where for all i Є [1..n],

sp (Qi, x := E) is defined as:

1. If an xC-equality occurs in Qi, let the constant symbol occurring in the leftmost xC-equality of Qi be D, then sp(Qi, x := E) = Qi[D/x] Λ x =E[D/x].
2. If an xC-equality does not occur in Qi, and an xv-equality occurs in Qi, let the non-x variable occurring in the leftmost xv-equality of Qi be w, then sp(Qi, x := E) = Qi[w/x] Λ x = E[w/x].
3. If an xC-equality does not occur in Qi, and an xv-equality does not occur in Qi, and an xf-equality occurs in Qi, let the non-x term occurring in the leftmost xf-equality of Qi be u, then sp(Qi, x := E) = Qi[u/x] Λ 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] Λ 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 Λ z = y * x, x := y)
≡ (x = a Λ z = y * x)[a/x] Λ x = y[a/x]
≡ (a = a Λ z = y * a) Λ x = y
≡z = y * a Λ x = y
(ii) sp (x = X, x := $3x^6 - 4x^2 + 2$)
≡ (x = X)[X/x] Λ x = ($3x^6 - 4x^2 + 2$)[X/x]
≡ (X = X) Λ x = ($3X^6 - 4X^2 + 2$)
≡ 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 v0 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 v0 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 v0 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[($\frac{x-b}{a}$)/x].

Proof For the assignment x: = ax + b to occur correctly in an imperative program, Q must be of the form (x = t) Λ 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) Λ P, x := ax + b)
= ((x = t) Λ P)[t/x] Λ x = (ax + b)[t/x]
= ((t = t) Λ P[t/x]) Λ x = (at + b)
≡ P[t/x]) Λ x = (at + b)
RHS = Q[($\frac{x-b}{a}$)/x]
= ((x = t) Λ P) [($\frac{x-b}{a}$)/x].
= ((($\frac{x-b}{a}$) = t) Λ P[($\frac{x-b}{a}$)/x])
= ((($\frac{x-b}{a}$) = t) Λ P[t/x])
=(x = at + b) Λ P[t/x]
≡ P[t/x] Λ (x = at + b)
≡ LHS

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 definitions 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

[1] Software Inspection: An effective verification process - A. Ackerman, L. Buchwald and F. Len ski, 1989.
[2] Predicate Calculus and Program Semantics Springer - Verlag E. Dijkstra and C. Scholten,1978.
[3] Understanding and Documenting Programs. IEEE Transactions on Software Engineering ,V. Basili and H. Mills,1982
[4] Program Construction and Verification, Prentice Hall - R. Backhonse, 1986
[5] Software Quality Improvement, Specification, Derivation and Measurement using Formal Methods. PhD thesis, School of Computing andInformation Technology, S. Pan. Griffth University, 1994.
[6] A Calculus of Refinements for Program Derivation, Acta Informatica - R. Back, 1988
[7] Powerful Techniques for the Automatic Generation of Invariants - S. Bensalem, Y. L akhnech and H. Saidi,1996
[8] Automatic Generation of Invariants and Intermediate Assertions - N. Bjorner, A. Browne and Z. Manna, 1997
[9] A Computational Logic Academic Press - R.Doyer and J. Moore, 1979
[10] R. Britcher: Using Inspections to Investigate Program Correctness, 1988
[11] Cleanroom Review Techniques for Application Development. In Proc. 6th International Conference on Software Quality M. Deck, 1996
[12] Guarded Commands for Formal Derivation of Programs E. Dijkstra,1975
[13] A Discipline of Programming, Prentice Hall - E . Dijkstra, 1976
[14] A Method of Progmmminq. Addison-Wesky - E. Dijkstra and W. Feijien,1988
[15] How to solve it by Computer. Addison-Wesle - R.Dromey, 1982