Verification Using Weakest Precondition Strategy

Nikolaj Popov

Research Institute for Symbolic Computation, 4232 Hagenberg, Austria
popov@risc.uni-linz.ac.at

Abstract

We describe the *weakest precondition strategy* for verifying programs. This is a method which takes a specification and an annotated program and generates so-called *verification conditions*: mathematical lemmata which have to be proved in order to obtain a formal correctness proof for the program with respect to its specification. There are rules for generating the *intermediate* pre- and post- conditions algorithmically. Based on these rules, we have developed a package for generating *verification conditions*.

1 Weakest Preconditions and Predicate Transformation

We have seen [Kovacs03] a verification method using inference rules of Hoare Logic. However, there exists another strategy for program verification using the so-called "weakest precondition predicate transformer" (*wp*) developed by E. W. Dijkstra [Dijkstra76]. This approach is also based on Hoare Logic.

Let us assume we want to verify a program where we know the postcondition but *not* the precondition:

$$\{?\}S\{R\}$$

In general, there could be arbitrarily many preconditions Q which are valid for the program S and a postcondition R. However, there is precisely one precondition describing the *maximal* set of possible initial states such that the execution of S leads to a state satisfying R. This Q is called the *weakest precondition*. (A condition Q is *weaker* than P iff $P \Rightarrow Q$.)

2 How It Works

We are given a program $s_1; s_2; \ldots; s_n$ with precondition $\{P\}$ and postcondition $\{R\}$. We want to verify this program by the *weakest precondition strategy*. Starting from s_n and $\{R\}$, we produce (algorithmically) $\{P_{n-1}\}$ and accumulate some lemmata, the so-called *verification conditions*. Here $\{P_{n-1}\}$ is the *weakest precondition* for the statement s_n, and $\{P_{n-1}\}$ now becomes the *postcondition* for s_{n-1}. Then we take s_{n-1} and $\{P_{n-1}\}$, and

$$\{P\}s_1; s_2; \ldots; s_n\{R\}$$

$$\{P\}$$

$$\{P_0\}$$

$$s_1;$$

$$\{P_1\}$$

$$s_2;$$

$$\ldots$$

$$s_{n-1};$$

$$\{P_{n-1}\}$$

$$s_n$$

$$\{R\}$$

Finally we produce $\{P_0\}$, and we have accumulated a list of lemmata. What remains is to prove the lemmata and also $P \Rightarrow P_0$. If we succeed to prove this, we can be sure that the program $s_1; s_2; \ldots; s_n$ *meets* its specification.

2.1 Simple Example

Given:

A program $S:\ y := x * x$

A postcondition $R:\ y \geq 4$

Find:

The weakest precondition Q.

Solution:

$Q:\ (x \leq -2) \vee (x \geq 2)$.

The precondition $x \geq 2$ would also guarantee that R is valid after execution. Even stronger preconditions like $x \geq 3$, $x = 3$, etc. would be valid preconditions as well. However, the weakest precondition is $Q:\ (x \leq -2) \vee (x \geq 2)$.

The weakest precondition of a program S and a postcondition R is denoted by

$$wp(S, R)$$

and is a predicate which describes the set of all initial states that will guarantee termination of S in a state satisfying R. This can also be expressed as a Hoare Triple:

$$\{wp(S, R)\}S(R)$$

So if one wants to verify $\{Q\}S\{R\}$ using wp, one can first determine $wp(S, R)$ and then prove

$$Q \Rightarrow wp(S, R).$$

For a fixed statement S, the function wp can be viewed as a function taking only a predicate (the postcondition) and returning another predicate (the weakest precondition). Therefore wp is also called a *predicate transformer.* More on predicate transformation can be found in [Dijkstra76].

For certain statements (e.g. *WHILE*), one also needs to generate *verification conditions.* These are mathematical lemmata, which have to be proved additionally. Moreover, fully automatic verification of *WHILE* statements is very difficult in general [Futschek89]. Therefore, in practice we need to supply a *loop invariant* and a *termination term* (see the example below).

2.2 While Example

Given:

A program S: $WHILE[\, y \leq r,$

$$r := r - y;\ q := q + 1,$$

$$Invariant \rightarrow (x = r + y * q)$$

$$TerminationTerm \rightarrow (r - y)]$$

A postcondition $R: \{x = r + y * q \wedge r < y\}$

Find:

The weakest precondition Q.

Solution:

$Q: \ (x = r + y * q)$

Accumulated Verification Conditions:

$(x = r + y * q) \wedge \neg(y \leq r) \Rightarrow (x = r + y * q) \wedge r < y$

$(x = r + y * q) \wedge y \leq r \wedge (r = T1) \Rightarrow$

$\quad (x = (r - y) + y * (q + 1)) \wedge (r - y) < T1$

$(x = r + y * q) \wedge y \leq r \Rightarrow r - y \geq 0$

One can see that the above formulae are provable in the theory of integers, under the condition: $y > 0$.

3 What We Have Available

In our system *Theorema*, we have developed a package for generating *verification conditions* [Kirchner99]. They are generated in such a way that they are syntactically acceptable for the available provers.

The Extended Predicate Transformer *EPT* is a function which takes a statement and a postcondition. It returns two "data structures": the precondition (the transformed postcondition) and a list of verification conditions.

$$EPT :< stat, post > \mapsto < pre, vcList >$$

The Verification Condition Generator *VCG* is a function which takes a program and its specification and returns a list of verification conditions.

$$VCG :< program, specification > \mapsto \{lemmata\}$$

The list of verification conditions is transformed into a *Theorema* formula list. Parameter variables (from the interface definition) and local variables occur as free variables in these formulae. In the last step of processing the formula list is turned into a *Theorema* `Lemma` where these variables are universally quantified. This job is done by a function called Verification Condition Generator VCG, which takes an annotated program with pre- and postcondition and returns a list of verification conditions.

$$\text{Annotated Program} \stackrel{\text{EPT}}{\longrightarrow} \text{Verification Conditions} \stackrel{\text{VCG}}{\longrightarrow} \text{Lemmata} \stackrel{\text{Prover}}{\longrightarrow} \text{Proof}$$

4 References

[**Dijkstra76**] E.W Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[**Kirchner99**] M. Kirchner. Program Verification with the Mathematical Software System Theorema. Diploma Thesis, Fahhochschule Hagenberg, 1999.

[**Futschek89**] Gerald Futschek. *Programmentwicklung und Verifikation.* Springer Verlag, Wien, New York, 1989.

[**Kovacs03**] L. Kovacs, Program Verification using Hoare Logic. Talk at *CaVIS* 2003, Timisoara, Romania, 2003.