Specified Blocks

position paper for
IFIP Working Conference on
Verified Software: Theories, Tools, Experiments
Zurich, Switzerland
2005 October 10 to 14

Eric C.R. Hehner

Department of Computer Science, University of Toronto
Toronto ON, M5S 2E4, Canada
hehner@cs.utoronto.ca

Abstract. This paper argues that specified blocks have every advantage over the combination of assertions, preconditions, postconditions, invariants, and variants, both for verifying programs, and for program development. They are simpler, more general, easier to write, and they make proofs easier.

0 Introduction

At the Workshop on Verified Software in Menlo Park in February of this year, various people spoke about assertions, invariants, variants, preconditions, and postconditions, as useful annotations for the verification of software. I shall refer to all of these as “assertion”, by which I mean something that is intended to be true whenever execution passes the point in the program where it is located. This position paper argues that assertions are superseded by the more general, easier to use, idea of specification.

The kind of specification I am advocating is not new; for a full account see [0]; for a shorter account see [2]. I hope that the version and presentation in this paper will make the idea more palatable to those who still cling to assertions.

All I ask of a programming language (for my present purpose) is that there be some way to make a block of code, such as the { } brackets in C and Java, and a way to label a statement or block. I will use ordinary ( ) parentheses to make a block, and an identifier followed by a colon for a label, but I make no case for any particular syntax.

1 First Example

To illustrate the idea, I will start with a standard programming pattern: do something to every item of a list. To make the example specific, given a natural number n, and a variable L of type lists of length n of integers, add one to every item of L. The program fragment is easily written.

$$i{:=}\ 0;\ \ \textbf{while}\ i\neq n\ \textbf{do}\ (L[i]{:=}\ L[i]+1;\ \ i{:=}\ i+1)$$

A specification describes the purpose of a block of code. Any block of code may have a specification. The only code that needs to have a specification is loops and procedures (methods, functions). For this example, I specify the whole, and also the loop within it.

```
add_1_to_every_item_of_$L$:
    (   $i$:= 0;
        add_1_to_remaining_items_of_$L$: **while** $i \ne n$ **do** ($L[i]$:= $L[i]+1$;  $i$:= $i+1$)
    )
```

We must also tell the verifier the formal specifications. So we define

add_1_to_every_item_of_L $=$ $(\forall j\colon 0,..n\cdot L'[j] = L[j] + 1)$
add_1_to_remaining_items_of_L $=$ $(\forall j\colon 0,..i\cdot L'[j] = L[j]) \wedge (\forall j\colon i,..n\cdot L'[j] = L[j] + 1)$

I am using an unprimed variable to stand for a variable's value at the start of the block to which the specification is attached, and a primed variable to stand for a variable's value at the end of the block to which the specification is attached. (The asymmetric notation $a,..b$ starts with a and includes $b\text{–}a$ integers.) The programmer is now finished, unless the automated verifier needs help. The verifier has to make one proof per label. It must prove

add_1_to_every_item_of_L $\Leftarrow$ (i:= 0; add_1_to_remaining_items_of_L)
add_1_to_remaining_items_of_L $\Leftarrow$ **while** $i \ne n$ **do** ($L[i]$:= $L[i]+1$; i:= $i+1$)

For the first of these, notice that the loop is represented only by its specification. In general, any specified block is represented in a proof only by its specification. Starting with the right side,

(i:= 0; add_1_to_remaining_items_of_L) replace informal spec with formal spec
$=$ (i:= 0; $(\forall j\colon 0,..i\cdot L'[j] = L[j]) \wedge (\forall j\colon i,..n\cdot L'[j] = L[j] + 1)$)
replace i by 0 in what follows
$=$ $(\forall j\colon 0,..0\cdot L'[j] = L[j]) \wedge (\forall j\colon 0,..n\cdot L'[j] = L[j] + 1)$ first $\forall$ has *null* domain
$=$ $(\forall j\colon 0,..n\cdot L'[j] = L[j] + 1)$
$=$ add_1_to_every_item_of_L

For the second, the verifier uses the refinement semantics of loops [1]. That means proving

add_1_to_remaining_items_of_L
$\Leftarrow$ **if** $i \ne n$ **then** ($L[i]$:= $L[i]+1$; i:= $i+1$; add_1_to_remaining_items_of_L) **else** *ok*

For the proof, the verifier needs to know a little theory of programming. It needs to know
if b **then** P **else** Q $=$ $b\wedge P \vee \neg b\wedge Q$
and it needs to know that if the variables are L and i, then
ok $=$ $L'=L \wedge i'=i$
Proofs, in general, are substitutions and simplifications, with little or no inventive steps. For lack of space in this paper, we do not pursue this proof further.

Specifications do not have to be complete. We might be interested only in some property of the computation. Suppose we are interested in its execution time. Then, if you will allow me a freer syntax of labels, we can specify

```
$t'=t+n$:
  (  $i$:= 0;
     $t'=t+n-i$: **while** $i \ne n$ **do** ($L[i]$:= $L[i]+1$;  $i$:= $i+1$;  $t$:= $t+1$)
  )
```

The specification $t'=t+n$ says that the final time t' is the initial time t plus n, measuring time as iteration count. The specification $t'=t+n-i$ says that the time remaining at the start and end of each iteration of the loop is $n-i$.

So why is this better than assertions? First, there is the well-known problem that an assertion (abstraction of a single state) cannot relate the output to the input. And there is the usual work-around: introduce some specification variables (constants?) of uncertain status (they ought to be quantified to make them local to the assertions that use them). In our example, we need two of them: one to capture the initial state of the list, and the other to capture the initial state of the variant. In the usual style, here is the annotated program.

```
$\{L = M\}$
$i := 0;$
$\{(\forall j: 0,..i\cdot L[j] = M[j] + 1) \wedge (\forall j: i,..n\cdot L[j] = M[j]) \wedge 0 \le n-i\}$
while $i \ne n$ do
        $\{(\forall j: 0,..i\cdot L[j] = M[j] + 1) \wedge (\forall j: i,..n\cdot L[j] = M[j]) \wedge 0 < n-i = V \wedge i \ne n\}$
        $(L[i] := L[i]+1;\ \ i := i+1)$
        $\{(\forall j: 0,..i\cdot L[j] = M[j] + 1) \wedge (\forall j: i,..n\cdot L[j] = M[j]) \wedge 0 \le n-i < V\}$
$\{(\forall j: 0,..i\cdot L[j] = M[j] + 1) \wedge (\forall j: i,..n\cdot L[j] = M[j]) \wedge i=n\}$
$\{(\forall j: 0,..n\cdot L[j] = M[j] + 1)\}$
```

Second, look at the number of assertions (six) needed, versus the number of specifications (two) needed. The loop rule we are using here

```
        if      $\{I \wedge b \wedge 0<v=V\}\ P\ \{I \wedge 0\le v<V\}$
        then    $\{I \wedge 0\le v\}$ while $b$ do $P$ $\{I \wedge \neg b\}$
```

is standard, and does indeed require all the assertions we have used.

Third, and more importantly, I think it makes more sense to programmers to say what a block of code is intended to do than to try to say what is true at strategic points in the code. In other words, it's easier to write specifications than preconditions and postconditions and invariants and variants. This point is elaborated in Section 2.

Fourth, and most importantly, specifications are not limited to talking about initial and final values of variables, nor are they limited to talking about terminating computations. They can talk about intermediate values of variables, and about communication sequences. They can talk about space usage. They work for loops with intermediate exits, and loops with deep exits, which are problematic for assertions. They work for general recursion, and for parallel composition, and a great many other things. This point is elaborated in Section 3.

2 Binary Search

Here is an example to illustrate the difference between writing assertions and writing specifications. Given a nonempty sorted list L, assign natural variable h to indicate a position where x occurs, if any. Adding natural variables i and j, here is a solution.

```
$h := 0;$  $j := \#L;$  while $j-h>1$ do $(i := (h+j)/2;$  if $L[i]\le x$ then $h := i$ else $j := i)$
```

We need a specification for the whole thing, and one for the loop. For the moment, I use meaningless labels A and B because the choice of meaning is the point of the example.

```
$A$:  (  $h$:= 0;  $j$:= #$L$;
         $B$:  while $j–h>1$ do
               (  $i$:= $(h+j)/2$;
                  if $L[i]≤x$ then $h$:= $i$ else $j$:= $i$
               )
      )
```

The proof obligations are

$$\begin{aligned} A &\Leftarrow\ h:=0;\ \ j:=\#L;\ \ B \\ B &\Leftarrow\ \textbf{while}\ j–h>1\ \textbf{do}\ (i:=(h+j)/2;\ \ \textbf{if}\ L[i]\le x\ \textbf{then}\ h:=i\ \textbf{else}\ j:=i) \end{aligned}$$

The first proof (after defining A and B) is two substitutions: replace j by $\#L$ and then h by 0 in B. The second proof is

$$\begin{aligned} B \ \Leftarrow\ \ & \quad j–h>1 \land (i:=(h+j)/2;\ \ L[i]\le x \land (h:=i;\ \ B) \\ & \qquad\qquad\qquad\qquad\qquad \lor\ \ L[i]>x \land (j:=i;\ \ B)) \\ & \lor\ \ j–h\le 1 \land h'=h \land i'=i \land j'=j \end{aligned}$$

Specification A is, informally, to find x in L. Formally, it is

$$(\exists i\colon 0,..\#L\cdot L[i]=x)\ \ =\ \ L[h']=x$$

If x occurs in L anywhere from the beginning to the end, then the final value of h will be a position of x; and if x does not occur anywhere in L, then the final value of h will be a position of some other value. (Obviously our search will be followed immediately by the test $L[h]=x$ to determine whether x was found.) For specification B we have a choice. The sensible choice is

$$h<j\ \Rightarrow\ ((\exists i\colon h,..j\cdot L[i]=x)\ \ =\ \ L[h']=x)$$

which says: given that the list segment from (incl.) h to (excl.) j is nonempty, search in there. It describes what remains to be done at the start and end of each iteration. There is an alternative choice for B which says: given what's true at the start and end of each iteration, fulfill the original task. Formally,

$$\begin{aligned} & \quad h<j \land \neg(\exists i\colon 0,..h\cdot L[i]=x) \land \neg(\exists i\colon j,..\#L\cdot L[i]=x) \\ & \Rightarrow\ ((\exists i\colon 0,..\#L\cdot L[i]=x)\ \ =\ \ L[h']=x) \end{aligned}$$

The antecedent (first line) is an invariant (as was the antecedent of the “sensible” choice). This example illustrates that we can, if we choose to, encode invariant assertions within specifications. It also illustrates how much simpler and more direct it is to say what's left to be done, rather than to formulate an invariant.

I have been teaching about specifications and proofs for many years. But my formulation of specifications has been affected (or infected) by my earlier education: I habitually looked for the invariant. I am embarrassed, and also proud, to say that my students have taught me to stop looking for the invariant. More generally, they write better specifications by thinking about “what remains to be done”, rather than about “what's true now” (assertions).

3 Product of Power Series

Here is an example to illustrate some of the general applicability of specified blocks. Write a procedure to read from channel a an infinite sequence of coefficients $a_0\ a_1\ a_2\ a_3$ ... of a power series $A = a_0 + a_1{\times}x + a_2{\times}x^2 + a_3{\times}x^3 + ...$ and in parallel to read from channel b an infinite sequence of coefficients $b_0\ b_1\ b_2\ b_3$ ... of a power series $B = b_0 + b_1{\times}x + b_2{\times}x^2 + b_3{\times}x^3 + ...$ and in parallel to write on channel c the infinite sequence of coefficients $c_0\ c_1\ c_2\ c_3$ ... of the power series $C = c_0 + c_1{\times}x + c_2{\times}x^2 + c_3{\times}x^3 + ...$ equal to the product of the two input series.

```
procedure PowerSeriesMultply (chan c)
(   var a0, a1, aa, b0, b1, bb, dd: real;  chan d: real;
    (read a0 from a || read b0 from b);  write a0×b0 to c;
    (  PowerSeriesMultply(d)
    || (   (read a1 from a || read b1 from b);  write a0×b1 + a1×b0 to c;
           write_remaining_coefficients_on_c_reading_from_a_b_and_d:
              while true do
              (  (read aa from a || read bb from b || read dd from d);
                 write a0×bb + dd + aa×b0 to c
              )
       )
    )
)
```

The procedure has a channel parameter; the channel supplied as argument will get the output. It also has a local channel declaration for use within the procedure. This procedure is nonterminating, reading and writing infinite sequences of coefficients; it has dynamic process generation (because a parallel composition occurs within each recursive call); it has synchronization (otherwise known as sequential composition); it has dynamic storage allocaton (declarations occur within each recursive call). A parallel composition is just a conjunction of its operands. The same input can be read by different processes, each at its own speed.

Just two specifications are needed, the procedure name and the loop name, and they are already present. The verifier requires formal definitions, as follows.

PowerSeriesMultiply(c) $=$ $C = A{\times}B$
write_remaining_coefficients_on_*c*_reading_from_*a*_*b*_and_*d* $=$ $C = a0{\times}B + D + A{\times}b0$

where D is the power series $d_0 + d_1{\times}x + d_2{\times}x^2 + d_3{\times}x^3 + ...$ formed from the messages $d_0\ d_1\ d_2\ d_3$ ... written to and read from local channel d . It would be nice to allow procedures and loops and other blocks to be named with a formal specification, rather than just an identifier, particularly when, as here, the formal specification is shorter.

The prover needs to be told what it means to multiply power series. We tell it that $C = A{\times}B$ means $c_n = \Sigma i\colon 0,..n+1\cdot\ a_i{\times}b_{n-i}$. We also tell the prover what it means to multiply a power series by a scalar, as in $a0{\times}B$, and what it means to add power series. After that, the proof is straightforward, well within reach of an automated verifier; it can be found in complete detail in [0].

4 Conclusion

The proposal in this paper is applicable to initial program development, to program modification, and to verification of completed programs. For program development, proof can be made when a specification is formalized, even before its block has been written. For program modification, specifications are the information needed to make the modification. For verification, the specifications need to be invented if they are not already present.

From the fact that my examples are small, some people draw the conclusion that the method is only applicable to small programs. That inference is ridiculous; that conclusion is wrong. There is no upper limit on the size of blocks that can be specified. Scaling up works because a specified block is represented in a proof only by its specification.

A specification says what a block of code is intended to do; an assertion says what is true at a strategic point in the code. The former is often easier to write and briefer than the latter. Furthermore, specifications are not limited to talking about initial and final values of variables, nor are they limited to talking about terminating computations. They can talk about intermediate values of variables, communication sequences, time usage, and space usage. Without inventing new proof rules, they work for loops with intermediate and deep exits, for general recursion, for parallel composition, and a great many other things.

When there are two scientific theories, each having a merit over the other, it makes sense to keep them both. For example, Einstein's theory of motion is more accurate than Newton's, applying to a broader range of motion; but Newton's theory is simpler than Einstein's, so we keep it and use it whenever it is accurate enough for our purpose. But sometimes one theory has all the merits. For example, Galileo's theory of elliptical orbits around a stationary sun is both more accurate and simpler than Ptolemy's theory of cycles within cycles around a stationary earth. So it would be scientifically irresponsible to continue to use the worse theory when a better one is available.

There is an interesting variety of ways of approaching verification. Some of them have at least one merit not shared by the others, and so they deserve continued attention. Assertions (invariants, preconditions, postconditions), however, are completely subsumed by specifications in the form of a single boolean expression; the latter are both simpler and more widely applicable than the former. It would be scientifically irresponsible to continue to use the worse theory.

5 Progress and Further Work

We can easily handle pointers that are attached to a specific data structure, but general pointers are harder. And current work [3] is investigating dynamic dispatch, and other features of object-oriented programming.

We have a working prover [4], an adaptation of HOL, that knows the necessary theory of programming, and keeps track of frames. It is aimed entirely at program development, and includes a syntax-directed editor that confines the user to a limited selection of programming constructs. Further work is needed to broaden the applicability and usability of the tool.

6 Comments on the Grand Challenge

Tony Hoare has made clear that the purpose of our grand challenge is scientific; we want to advance our long-term scientific understanding of program verification. That is what attracts me to the project. No-one denies the urgent needs of industry, nor the value in meeting those needs, but that is not the goal of this project.

It seems to me that the problem of coping with large amounts of badly written legacy code falls on the side of urgent industry need. It has been proposed that, due to the large quantity of legacy programs, we treat them as though they were natural objects, and worthy of study. To me, the quantity seems irrelevant, unless one is proposing to do a statistical study of them. I doubt that anything of long-term scientific interest will come from a study of badly written legacy code.

Some legacy code is well written, and can serve as an experimental testbed. But we must not treat it as sacred and unmodifiable. If it has any bugs, we will have to modify it to verify it. We may as well modify it any time we think a structural change will help us to verify it.

Just as I do not want to be constrained by badly written legacy code, I also do not want to be constrained by badly designed existing languages. In the Menlo Park meeting there was some discussion about the programming language we should be directing our verification to, and some commented that the future is Java. Our timeframe is 15 years, and if we look back 15 years, any prediction then about our working language now would have been ridiculous (Java was not yet invented). We should build on whatever we consider to be good, and we should not build on, or toward, any language or language features that we consider to be bad for verification.

Finally, I do not want our project to be constrained by any assumption about the abilities of existing programmers. I agree with Bertrand Meyer that such assumptions are probably wrong; programmers can and do learn what they perceive to be of help to them. Many of our current formal methods and verification tools are more trouble than they are worth. The scientific challenge is to find the simplest theories and methods that are adequate.

7 Acknowledgements

I thank Doug McIlroy for the Power Series example. I thank Albert Lai and Yannis Kassios for discussion concerning specification style.

8 References

[0] E.C.R.Hehner: *a Practical Theory of Programming*, Springer, New York, 1993; current edition available free at www.cs.utoronto.ca/~hehner/aPToP

[1] E.C.R.Hehner, A.M.Gravell: “Refinement Semantics and Loop Rules”, *FM'99 World Congress on Formal Methods*, Toulouse France, 1999 September 20-24, LNCS 1709 p.1497-1510, and www.cs.utoronto.ca/~hehner/RSLR.pdf

[2] E.C.R.Hehner: “Specifications, Programs, and Total Correctness”, *Science of Computer Programming* v.34 p.191-205, Elsevier, 1999, and www.cs.utoronto.ca/~hehner/SPTC.pdf

[3] I.T.Kassios: *a Decoupled Theory for Object-Oriented Refinement*, Ph.D. thesis, University of Toronto, in progress

[4] A.Y.C.Lai: *a Tool for a Formal Refinement Method*, MSc thesis, University of Toronto, 2000