SOFTWARE VERIFICATION TOOLS

PART III

PETER H. ROOSEN-RUNGE

DEPARTMENT OF COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, CANADA

© P. H. Roosen-Runge 1999-2001

7. USING PROPOSITIONS AS SPECIFICATIONS

7.1 Specifications and computational states

Propositions which contain non-logical terms give us a formal language describing the states of a computation, either the actual state of a computation or a state which a program is intended to achieve at some particular point in the computation. In the latter case, we call the description a *specification*. A specification can also include a description of the assumed initial state of a computation—a *pre-condition*. A specification of the final state of a computation is called a *post-condition*. As the names suggest, pre- and post-conditions are typically not descriptions of what actually happens in a computation but are requirements on what ought to happen. However, we will sometimes blur this point and consider post-conditions based, not on what we want the computation to do, but on what we think it actually does.

Specifications describe computational states, and computational states are determined by the values of variables. So specifications typically express requirements on some or all of the variables in a program, using mathematical functions and relations. Here are some examples:

- "x is a natural number that is at least one and at most ten"

Translating this into a mathematical proposition might yield:

natural(x) and 1 <= x <= 10.

- "x is either a prime or divisible by the prime y" recast into propositional notation:

prime(x) or (mod(x, y) = 0 and prime(y)).

Notice that this latter specification does not explicitly require that x is an integer, although if the specification is satisfied, then this would be implied by prime(x). Now suppose, in this example, that x = 2 and y = "string". These values satisfy the specification, since the variable y does not need to be an integer if x is prime.

- "If the first character of the string x is not "a" then x is of length 0.

not x(1) = "a" implies length(x) = 0.

Here, in order to formalize the specification, we need notations to express the concept of selecting a character from a string and measuring the length of a string. Such notations are not part of standard mathematics, but are easily invented. In this case, a representation has been chosen which treats a string mathematically as a function from integers to characters.

The example illustrates that specifications need not provide equivalent detail for every possible case; a specification may only specify what is true or required under some particular assumption or condition. If the assumption does not hold, the specification may be trivially true, as in this case, since the specification is equivalent to

x(1) = "a" or length(x) = 0.

If x(1) = "a", then the disjunction is true and the specification supplies no additional information about x.

Exercise 7.1: Convert each of the following specifications to a single proposition.

(a) m is the smallest square larger than or equal to n

(b) x, y, and z can be ordered into a sequence of three distinct values, using a binary Boolean relation ≤ as the ordering.

(c) the size of a stack s is one more than the size of pop(s) if s is not nil.

Often a condition in the specification of a computational state is expressed as a pre-condition on a prior state. For example, in order to compute pop(s), where s is a stack, s must be non-empty. Thus, the code

```
x = pop(s); y = push(a, x);
```

can only guarantee the post-condition

top(y) = a

if the pre-condition not (s = nil) is met, prior to the execution of the code.

To get some practice with simple specifications, we will consider examples in which the code is given and the task is to construct a post-condition. In principle, code should be constructed from specifications, not the other way around, but it usually is not, at least not from formal specifications, so in applying the concept of verification to code, we have to expect that the specifications may be constructed after the fact.

Consider the following C / Java statement[1]:

```
if (!lightOn) { wattage = 0; roomDark = true; }
else { wattage = 60; roomDark = false; }
```

To give a post-condition for this statement means to specify the values of the variables and the logical relationships between them after the statement is executed. A natural approach is to make a disjunction from the if and else cases:

(not lightOn and wattage = 0 and roomDark) or (lightOn and wattage=60 and not roomDark)

The Boolean variables `lightOn` and `roomDark` in the code can be used directly in the specification as logical variables; the C / Java negation `!` is changed to the corresponding not operator.

[1] This and some of the following examples are taken from Ritchey, T. , *Java!*, New Riders: 1995.

Instead of using disjunction, we can use a conjunction of implications which expresses the if-then structure more closely:

(not lightOn implies wattage = 0 and roomDark) and
(lightOn implies wattage=60 and not roomDark)

The description of the state of a computation may need additional variables that are not mentioned explicitly in the code. For example, the return statement constructs a value which is part of the program state at that point in the computation, but no variable is declared for it in C/Java. In such cases, we create additional *specification variables*[2] to be used in post-conditions. These are mathematical variables which are defined in terms of the values of program variables at a particular point in a computation; unlike program variables, specification variables never change their value, once defined.

In the following example, we create a specification variable return whose value is the value returned by the code, and we use it to express the post-condition following the return from the function call:

```
int i = lastIndexOf(o);

if (i >= 0) {
    return size() - i ;
}
return -1;
```

The post-condition can be expressed as:

lastIndexOf(o) >= 0 and return = size() - i or lastIndexOf(o) < 0 and return = -1

Strictly speaking, this post-condition is attached to a *state* of the computation—the state following the return from the function—not a specific place in the code, so it must be expressed as a general condition that obtains no matter which return statement is executed. If we want to show what condition holds after each return statement, we can include the condition as a comment immediately following the statement, with the understanding that it is specific to that exit, not a general description of the computational state following the function's return:

[2] Dahl(1992) refers to these as *mythical* variables. Backhouse (2003, p.107) terms them *ghost* variables.

```
   int i = lastIndexOf(o);

   if (i >= 0) {
       return size() - i;
//{ lastIndexOf(o) >= 0  and return = size() - i }
   }
   return -1;
//{ lastIndexOf(o) < 0  and return = -1 }
```

Exercise 7.2:

(a) Write a post-condition for the code:

```
   if (x > 0)
        if (x == y) a = z;
   else a = x;
```

(b) and for the code:

```
   if (x > 0) {
        if (x == y) a = z;
   }
   else a = x;
```

Exercise 7.3:

(a) Write a post-condition for the intended effect of the following `switch` statement, using a specification variable or variables whose values are character string that are printed.

```
   switch (temperature){
        case (0):
             System.out.println("Freezing water");
             break;
        case (37):
             System.out.println("Human body");
             break;
        case (100):
             System.out.println("Boiling water");
             break;
        default:
             System.out.println("some temperature");
   }
```

(b) Do the same for the following code:

```
switch (temperature){
      case (0):
            System.out.println("Freezing water");

      case (37):
            System.out.println("Human body");
            break;
      case (100):
            System.out.println("Boiling water");

      default:
            System.out.println("some temperature");
}
```

Exercise 7.4: Write three different but equivalent post-conditions for the following statement:

variable = (boolean ? value1 : value2) + addition;

- using a conditional (if) expression ,
- as a conjunction with no conditional expression,
- as a disjunction with no conditional expression.

Exercise 7.5: Write a specification for the state of the variables at the end of the following (Pascal) code:

```
IF (DlogItem = DoneDlogItem) OR
            (DlogItem = CancelDlogItem) THEN
      BEGIN
            anyChanges := TRUE;
            theSCSI := whichSCSI;
            ClockMhz := clockRate;
      END
ELSE
      anyChanges := FALSE
```

Here is a more challenging example of specification construction, in which the code is not given, but an informal description is provided, describing the interactions between an application and a Screen Manager system module in PowerTV's set-top box operating system:

The developer's guide[3] gives us the following information:

[3] Adapted from Sambar, S., and J. Becker, *PowerTV Operating System Overview*, PowerTV, Inc.: 1998.

"Applications conserve precious set-top memory by using a shared screen, rather than each application allocating its own purgeable screen. Purgeable screens save the time associated with redrawing the screen when the application becomes active, but at the expense of memory.

Purgeable screens allow Screen Manager to reclaim their memory resources if memory is limited. If a screen is purged, the application is notified the next time that it calls **scr_IsPurgeable()**. This notification comes in the form of a kEt_ScrPurged event. Once this event is delivered, two things can happen:

• If the system successfully resets the purged screen to use shared memory, it delivers to the application a kEt_ScrActivated event, indicating that the shared screen is ready for use. The application can then either request that memory for the purged screen be reallocated by calling **scr_ReallocatePurged()** or it can redraw the shared screen and abide by the drawing restrictions for shared screens.

• If there is not enough memory to reset the purged screen to use shared memory and bring the shared screen into focus, the system delivers a kEt_ScrUnavailable event to the application."

Suppose event notifications are obtained by the application by calling the function **getNextEvent()**, and that the application decides whether to retain the shared screen or reallocate a purgeable one based on whether **redrawTime** is greater than **minRedrawTime**, unless there is not enough memory for a purgeable screen**.** Assume that **scr_ReallocatePurged()** returns a value greater than 0 if there is enough memory for the reallocation, otherwise 0, and that **scr_ReallocatePurged()** is not called unless the application has received a kEt_ScrPurged notification.

We want to develop a post-condition which expresses the conditions under which the application successfully calls **scr_ReallocatePurged()** to created a purgeable screen, or uses a shared screen. To do this, we need some boolean specification variables to represent the events of interest such as a kEt_ScrPurged or kEt_ScrActivated event being received.

We define:

scrIsPurgeableCalled iff **scr_IsPurgeable()** was called;

scrIsPurgeable iff **getNextEvent()** was then called and returned a kEt_ScrPurged event;

sharedScreenReady iff the following **getNextEvent()** returned kEt_ScrActivated;

purgeableScreen iff a purgeable screen was created.

Then we can express the intended post-condition for an implementation of the above informal specifications as:

scrIsPurgeableCalled and scrIsPurgeable and (not sharedScreenReady or sharedScreenReady and (scr_ReallocatePurged() > 0 and redrawTime > minRedrawTime and purgeableScreen or not purgeableScreen and (scr_ReallocatePurged() = 0 or redrawTime ≤ minRedrawTime)))

Notice that this post-condition describes what happens if the kEt_ScrPurged event has been received: it requires that scrIsPurgeableis true. A more complete specification will cover the possibility that kEt_ScrUnavailable and that scrIsPurgeable is false.

The original specifications are quite difficult to unravel, partly because of the wording, and partly because of the complexities of the operation of the Screen Manager. The ambiguities in the wording and possible misunderstandings as to the intended logic could easily lead to an incorrect implementation. Given the formal post-condition and the definitions of its variables, the task of constructing a correct implementation becomes easier and verifiable. (See Exercise 8.11.)

7.2 Interfaces

In object-oriented programming, object classes can be described by *interfaces*, as in the following example:

```
public interface Set extends Container
  {
  /**
   * Return the first object that matches the given object, or
null if no match exists.
   * @param object The object to match against.
   * @see Set#put
   */
  public Object get(Object object);

  /**
   * If the object doesn't exist, add the object and return
       null, otherwise replace the
   * first object that matches and return the old object.
   * @param object The object to add.
   * @see Set#get
   */
  public Object put(Object object);

  /**
   * Remove all objects that match the given object.
   * @param object The object to match for removals
   * @return the object removed, or null if the object was
       not found.
   */
  public Object remove(Object object);
  }
```

In this example, a post-condition is definable for each method given in the interface. Since the interface is for objects of type Set, we can use set operations such as ε, , and - to express the post-conditions. We need a specification variable to represent the set to which the methods are applied; in fact, for put and remove, we need two such variables: we'll use OldS to represent the initial state of the set, and S to represent the final state; we also need an order relation ≤ on the Set objects to represent the concept of

the "first" matching object[4]. Matching will be represented by a Boolean relation match on set objects.

For the get operation, we have the post-condition

> get(Object) ε S and match(get(Object) , Object) and
> (X ε S and match(X, Object) implies get(Object) ≤ X) or
> X ε S implies not match(X, Object) and get(Object) = null.

The remove operation has the post-condition:

> Object ε OldS and S = OldS - Object and remove(Object) = Object or
> not (Object ε OldS) and remove(Object) = null.

In this interface, there are no pre-conditions for achieving the post-conditions, or putting it another way, we can take the pre-condition for each method to be the trivial condition true.

Exercise 7.6: Specify a post-condition for the `put` method in the interface defined above.

Exercise 7.7: Suppose the description of the `put` method in the interface read

```
 /**
* Replace the first object that matches the object and
     return the old object.
* @param object The object to add.
* @see Set#get
*/
```

Write an appropriate pre-condition for this method.

7.3 Refining an interface specification

In the evolution of a program, the capabilities of a class may be extended by replacing it with one with more capabilities. Very often, it is important that this "upgrade" not require the rewriting of *existing* code which uses the services of the class. This requires that the new class retain the names and signatures[5] of the public methods in the original class, so that the original code is still syntactically correct; but more important, nothing should change in the expected semantics of the new methods from the perspective of calls to those methods which rely on the specifications given in the previous class interface. If we think of the pre- and post-conditions in the interface as a *contract* which promises a certain performance if certain conditions are met, then the

[4] Strictly speaking, sets are unordered, so no ordering should be definable for the Set class. But in practice, Sets are often implemented as ordered collections.

[5] The signature of a method is the sequence of types of the arguments in the method's argument list.

requirement we want for reuse of the new class in the existing code is that the contract still holds. This means that if the pre-conditions in the original contract are met, then each post-condition Post in the original interface must be implied by its corresponding post-condition $Post_{new}$ in the new interface. In this case, we say that the new interface **refines** the original.

> Failure of new code to properly refine old code is one of the most common causes of system failure. When we read that a major bank was unable to record deposits or payments for two days, we are not surprised to learn that "attempts by the bank to improve one of their computer systems caused problems with another system, throwing everything out of whack."[6]

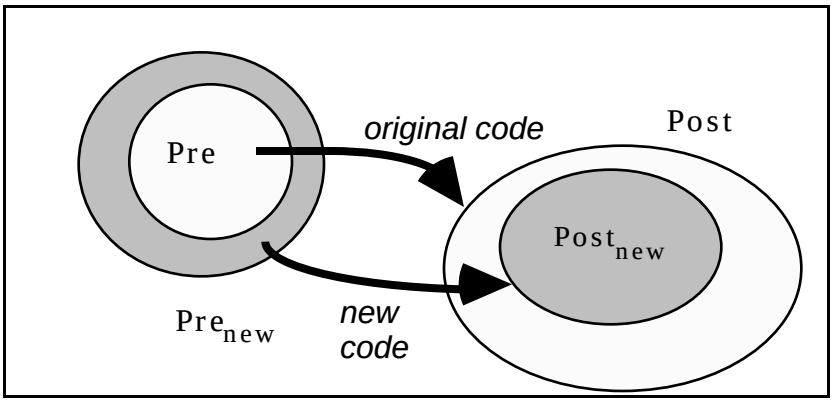
An easy case of refinement is one in which the original pre-conditions imply the pre-conditions in the new interface, and the new post-conditions are special cases of the original post-conditions:

Pre implies Pre_{new}

and

$Post_{new}$ implies $Post$.

We can use a Venn diagram to help visualize the relationships :



As an example, suppose we extend the Set methods in Section 7.2, by adding a function count(S, Object), so that after each method is executed, the variable S.count = count(S, Object) = the number of objects in S which now match the argument Object. Thus, after calling remove(Object), the post-condition is

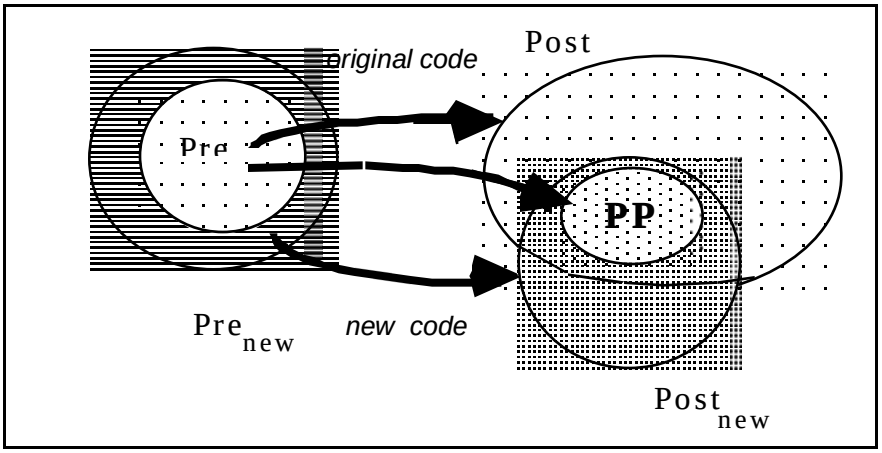
Object ε Old S and S = OldS - Object and remove(Object) = Object
and S.count = count(OldS, Object) - 1
or
not (Object ε Old S) and remove(Object) = null
and S.count = count(OldS, Object).

No pre-condition is required and so, as in the original interface, we take the pre-condition to be true. It then follows trivially that the original pre-condition implies the new one, and the new post-condition implies the original one; so existing code (which, of course, makes no use of the new count method) can safely use the new Set class—if it

[6] Globe and Mail, May 2, 1998.

met the conditions of the original contract and the new class meets the post-conditions specified in its interface, then the requirements of the original contract will be met.

But not all refinements are so easily established. Suppose the relationships between the original and the new conditions are as shown in the following diagram:



Here, while the original pre-condition guarantees that the new one is satisfied, the new post-condition no longer implies the original post-condition Nonetheless, the new interface satisfies the original contract—why? Because, *when the new code is used*, the original pre-condition achieves something more restricted than the stated post-condition $Post_{new}$; it achieves a specification PP which happens to imply the original post-condition.

To illustrate this, we consider a more complex interface, based on an example given by [Szyperski, 1997, p. 74]:

The original class interface, called TextModel, involves a method `write` which inserts a character into a character array. The pre-condition requires that the position at which the character is inserted lies between 0 (the first position) and the length of the array len (the last position), inclusive. The post-condition ensures that the text contains the original characters and the inserted character at the right positions.

```
interface TextModel {

void write (int pos, char ch);
// insert char ch at position pos within the existing text.

// pre-condition:
//{ len = 'this.length'(nil) and txt = ‘this.text’(nil)
    and (0<= i < len implies 'this.read'(i) = array(txt,i))
    and len < 'this.max'(nil)'
    and 0 <= pos <= len }

// post-condition:
//{ 'this.length'(nil) = len + 1
    and (0<= i < pos implies 'this.read'(i) = array(txt,i))
    and 'this.read'(pos) = ch
    and pos < i < 'this.length'(nil) implies 'this.read'(i)
```

```
       = array(txt, i-1) }
```

In this interface specification, methods with no arguments are written as having the argument `nil` so that they can be treated as functions in the mathematical sense. The variables `len`, `txt`, and `i` are specification variables, not program variables; `len` is the value returned initially by the class's `length` method. `pos` is a program variable, but since it is a parameter, it, like `len`, doesn't change as a result of executing the `write` method. `txt` is the text array returned initially by the class's `text` method; again, as a specification variable, its value does not change as a result of the `write`, although the value returned by `this.text()` does. Finally, `this.read(i)` is a class method which returns the character at position `i` in `this.text()`.

The reason for the single quotes around the function names is to allow us to input propositions of this sort into tools such as `prover`; otherwise, the dot in a qualified name would cause a syntax error.

Now suppose we want to replace the `write` method with one which allows a character to be inserted beyond the end of the original array, with the intervening positions filled up with blanks. To distinguish this from the previous case, we will call the new interface BetterTextModel. The name of the method is still `write` and so we need to ensure that the original code which calls `write` will still function correctly.

```
interface BetterTextModel {

void write (int pos, char ch);
// insert char ch anywhere--if after the end of the text,
   pad with blanks.

// pre-condition:
//{ len = 'this.length'(nil) and txt = 'this.text'(nil)
    and (0<= i < len implies 'this.read'(i) = array(txt,i))
    and len < 'this.max'(nil)
    and 0 <= pos <= 'this.max'(nil) }

// post-condition:
//{ 'this.length'(nil) = max(len, pos) + 1
    and (0 <= i < min(pos, len) implies array(txt, i) =
        'this.read'(i))
    and 'this.read'(pos) =  ch
    and (pos < i < 'this.length'(nil) implies
        'this.read'(i) = array(txt, i-1))
    and (len < i < pos implies 'this.read'(i) = " ") }
```

It is easy to see that TextModel's pre-condition implies that of BetterTextModel; the conjuncts are identical except for

0 <= pos <= 'this.max'(nil)

which should be implied by 0 <= pos <= len, and is, since

len < 'this.max'(nil).

But BetterTextModel's post-condition does not imply that of TextModel. The problem lies with the conditions in BetterTextModel which do not occur in the TextModel interface. BetterTextModel has

'this.length'(nil) = max(len, pos) + 1 and
(0 <= i < min(pos, len) implies array(txt, i) = 'this.read'(i))

whereas the corresponding condition in TextModel reads

'this.length'(nil) = len + 1 and
(0<= i < pos implies 'this.read'(i) = array(txt,i)).

Is there any more information which we can add to the post-condition stated for BetterTextModel which would allow us to prove the TextModel conditions? Yes, there is, for the pre-condition for TextModel states that

0 <= pos <= len

(which, note, is *not* required for BetterTextModel), and—here is the key point—the variable pos is an argument to the write method and hence can not be altered by the method, and len is a specification variable, with the fixed value 'this.length'(nil) as determined prior to executing the method; its value does not change either. So the inequality can be added to the post-condition achieved by the BetterTextModel **if** the pre-condition for using the original code is met. Thus, under the original contract,

min(pos, len) = pos and max(len, pos) = len

holds following the execution of the method, and that, together with BetterTextModel's post-condition, ensures that TextModel's contract is satisfied.

Exercise 7.8: Identify the condition PP in the diagram above, for the case of the TextModel, and write it out as a proposition.

7.4 The Well-Behaved Expression Assumption

The specifications in the previous examples can only work under a general assumption which needs to be made explicit: that the built-in operations and functions computed by the code are correctly implemented by the compiler or interpreter to agree with the mathematical definitions assumed to hold for the specifications. In other words if a + occurs in the code, we assume it means the same as a + in a specification, and for the latter, we usually take the normal mathematical meaning or we rely on some set of axioms for an abstract datatype to tell us what + means (in the context of character strings, for example.)

But the assumption the operations in the code have the expected mathematical meaning doesn't always hold, at least not without qualification. Consider the following Java code fragment with post-condition :

```
int n1 = Integer.MAX_VALUE;
int n2;
```

```
n2 = n1 + 1;
//{ n2 = n1 + 1}
```

The code cannot meet the specification, for if `n1` is equal to the largest `int` value `Integer.MAX_VALUE`, then `n1 + n1` will not return the value `n1 + n1`.[7]

So we could only verify this code relative to an assumption about the operations and functions in the code, an assumption which we might express informally as follows:

> *the operations or functions referenced in the code are assumed to be correctly implemented to agree with their mathematical counterparts for some very large range of expected variable values— in this case, for nearly all* ***`int`*** *values.*

We will call this assumption the *Well-Behaved Expression Assumption***.** Relative to this assumption, the code fragment obviously meets its specification.

Exercise 7.9 Write a post-condition for what the following code achieves:

```
if (count > 2) flag = adjust(total - 1);
else flag = adjust(total - 1);
```

Discuss the specific ways in which the post-condition depends on the Well-Behaved Expression Assumption.

The problem with the Well-Behaved Expression Assumption is that a mathematical function may itself not be "well-behaved" on quite ordinary values. This is a different problem than in the preceding—rather than assuming that an operation in the code correctly implements a corresponding mathematical function, here we have to recognize in the specification that the mathematical function itself is only defined under specific conditions.

So for the code:

```
y = (a *n ) / n;
     //{ y = a }
```

the case of `n = 0` cannot be taken to be covered under the Well-Behaved Expression Assumption, by excluding `0` from the assumed range of `n`. The problem is not with the implementation of `/` but with its mathematical definition. In order for `y = a` to be true following the statement, `n <> 0` must also be true, but expressing this as an implication

`n <> 0 implies y = a`

[7] Under some Java run-time interpreters, the code causes a system crash, a very undesirable outcome. This underscores how important it is to recognize clearly the exact conditions under which even the most innocent looking code such as a simple sum statement can be expected to execute correctly.

doesn't quite capture the situation. For the implication allows n = 0, and in that case, there isn't any computational state to describe following the division by n. *Nothing should be true* about the computation following division by 0. If we were to allow n = 0 to be true after the division statement (by, for example, skipping the division operation), then it would be possible for a result to be produced even though the mathematical function being computed is not defined for n = 0 and therefore does not have this value—a distinctly undesirable situation. As one author puts it:

> "It is of fundamental importance for our confidence in a program that *it should never produce wrong results which could be mistaken for correct ones*. Instead one should insist on some easily recognizable abnormal behavior, such as program abortion, whenever correct results cannot be computed. " [Dahl, 1992, p. 60]

If, instead of an implication, we were to specify

n <> 0 and y = a

then indeed, if n is initially 0, the post-condition is false. So the only way it can be achieved is through a *pre-condition* that n is not 0.

But it is not always possible to meet such pre-conditions, and users find the penalty of a "program abort" too drastic.[8] Some languages, such as Java, provide therefore a weaker substitute: an **exception mechanism**, as illustrated in the following example:

```
public class TrivialApplication {

   public static void main(String args[]) {
        int x, n;
   . . . .
        x=100/n;
        System.out.println( "Hello World!" );
   }

}
```

If n is 0 when "x=100/n;" is executed, the following output is produced (on some Java Virtual Machines):

```
Executing: javai -working test
                 -classpath TrivialApplication
java.lang.ArithmeticException: / by zero
   at TrivialApplication.main(TrivialApplication.java:9)
Completed(0)
```

[8] Alan Cooper reports that in 1997, the US guided-missile cruiser Yorktown was completely disabled due to the accidental entry of a zero as a divisor into a calibration being carried out on an Intel Pentium II PC running Windows, "which resulted in a complete crash of the entire shipboard control system. Without the computers, the engines stopped and the ship sat wallowing in the swells for two hours and forty-five minutes until it could be towed into port." [*The Inmates are Running the Asylum*, 1999, p. 13]

In this case, there *is* something true of the computational state, following the division by zero—an exception has been "thrown" (and in this case, "caught" by the system handler `java.lang.ArithmeticException.`) We can represent this in a post-condition as we did with the `return` statement, by introducing as a specification variable, a Boolean variable which is true if an exception has been raised, along with a variable for the character string printed (as in Exercise 7.3):

(n <> 0 and x = 100/n and printed = "Hello World!") or (n = 0 and exceptionRaised)

Exercise 7.10 Construct a post-condition to describe the computational state after the following code is executed:

```
    /**
     * Peeks at the top of the stack.
     * @exception EmptyStackException
     * if the stack is empty.
     */

   public Object peek() {

   int   len = size();

   if (len == 0) throw new EmptyStackException();
   return elementAt(len - 1);
   }
```

In general, expressions within code fragments are "well-behaved" to the extent that they have the same meaning in the code that they have in the mathematical descriptions of the computation. One implication of being well-behaved is that distinct function calls with the same arguments should, as they do in mathematical expressions, return the *same value*. But this is not always the case, for example, when a function operates on an object external to the code whose own state is changing independently. So in the earlier example of the settop box, the function getNextEvent() may return different event constants to the application code, depending on the current state of the operating system. This makes it difficult to connect the code with a mathematical description.

Are specifications "correct"?

To conclude this chapter, we consider the question of "correct specifications". By constructing specifications for code in terms of mathematical and logical concepts, the question of verification becomes a formal problem of determining whether or not a computational state satisfies a particular description, in which case the code can be said to be correct for that specification. But the issue of *code correctness*. should not be confused with the issue of the 'correctness' of a specification. We can verify pieces of code with respect to specifications, but we cannot prove whether the specifications themselves are what they should be—that they are what the programmer or user intended. As W. Maurer [1979] has observed: "you cannot state mathematically the property of users being satisfied." A specification may be reasonable or unreasonable, useful or useless—but it is not provably correct or incorrect.

Thus the argument that verification gives you a false sense of security,, because you don't know whether what you specified is what you want, or whether you left out something important, misses the point. What verification gives you is a method for comparing what a program does, as expressed by its computational states, and its stated requirements or goals. There is no more security in a verified piece of code than there is in a text that has passed a spelling and grammar checker.

8. VERIFICATION THROUGH SYMBOLIC EXECUTION

8.1 Sequences of assignment statements

As an application of the algorithms incorporated in the `prover` program, we consider the problem of verifying a fragment of Java code consisting of a sequence of simple assignment statements.

If we are given an initial condition in the form of a set of initial values for some of the variables in the code, then we can propagate this condition forward through the sequence of assignment statements, updating the condition after each statement to show the effects of assigning a new value to one of the variables.

To simplify the updating, the condition on the variables is maintained as a table[9], which, expressed as a proposition, has the form of a list of equalities:

$v_1 = e_1$ and $v_2 = e_2$ and $v_3 = e_3$ . . , etc.

In order to avoid circular descriptions, we will sharply distinguish between the **variables** in a program text and their mathematical **values** — we will require that the right side of each equality be a pure value containing no reference to any of the variables. (Variables change their value during execution. By eliminating variables from the expressions on the right-hand side of the equalities, we ensure that the value remains a fixed mathematical quantity.)

The effect of assigning a value to v_i by the statement $v_i = e_{new}$ can be recorded in the table by replacing the old value e_i in the table by the expression e_{new} *after eliminating all references to the* v_j *in* e_{new} by substituting the corresponding e_j. No circularity can arise in this process since the values in the table contain no references to program variables.

Suppose at some point in the sequence of statements, an assertion is claimed to hold among the statements. For example, given the initial condition:

$x = a + 2$ and $y = 7$,

then after the execution of the statements

```
x = x - y ;  y = x ;
```

it can be asserted that

$y = a - 5$

holds.

This can be checked by a program which *symbolically executes* the assignment statements, using and updating the current table of values assigned to the variables at each step[10]. If

[9] Sometimes called a *trace table*. See [Stavely, 1999, 3.5-3.7]

we include the conditions on the variables as comments in the code, we can represent the symbolic execution as:

```
//{ x = a + 2 and y = 7 }
   x = x - y ;
//{ x = a + 2 - 7 and y = 7 }
   y = x ;
//{ x = a + 2 - 7 and y = a + 2 - 7 }
```

(As in Chapter 7, we distinguish conditions and assertions from other comments in the code, by wrapping them in { }.)

The last condition, given the appropriate simplification rules, will simplify to

x = a - 5 and y = a - 5

which obviously implies that

y = a - 5.

So the program statements are verified with respect to the given assertion.

8.2 Initial values

In the first code example, the program variables x and y were explicitly assigned initial values expressed in terms of constants; however, in the final assertion, there was no reference to these initial values. This is not always what's needed. In many cases, we will wish to verify an assertion explicitly involving the initial values for which we then need some names. We will adopt a convention employed in the programming language Eiffel [Meyer, 1988] and use the special value 'old *V* ' as the name of the initial value of the variable *V*.

To see how this convention is used, consider a program which increases x by 1 and sets y equal to (the original value of) x - 1. The final assertion is then

x = 'old x' + 1 and y = 'old x' -1

and symbolic execution of the following code

```
y = x - 1;
x = x + 1;
```

with the implicit pre-condition

---

[10] See [Dannenberg, 1982] for an early computer implementation of symbolic execution and a formal treatment of the underlying algorithm. It is briefly discussed in [Dahl, 1992, p. 76-77]. [Gannon, 1994, Ch. 3] applies symbolic execution to the verification of functional specifications.

Symbolic execution is more generally used in analyzing the properties of programs, under the labels *non-standard execution* and *abstract interpretation* "which amounts to performing the program's computations using *value descriptions* or *abstract values* in place of the actual computed values." [Jones, 1994]

x = 'old x' and y = 'old y'

will show that the final assertion is correct.

If the statements in the code were reversed, the assertion x = 'old x' + 1 and y = 'old x' -1 would not hold. But in logic, the ordering of a set of conjuncts does not affect the truth-value—a and b says the same thing as b and a. Again, there is a gap between the code and its re-formulation in logic. Is there a way of writing the desired assignments which avoids the dependence on the order of the assignments and removes the gap? The following notation

```
x, y = x + 1, x - 1;
```

while not valid C/Java syntax, expresses the idea that the effect of the assignments is as if they were performed in parallel or simultaneously. The meaning of the parallel assignment notation can be specified precisely using the previously introduced notation for textual substitution:

$x_1, x_2, \ldots, x_k = e_1, e_2, \ldots, e_k$;
{ $x_1 = e_1$[old x_1 /x_1] and $x_2 = e_1$[old x_2 /x_2] and . . $x_k = e_k$[old x_k /x_k] }

Introducing parallel assignment into standard programming languages would be a useful way to eliminating the chance of errors resulting from a faulty ordering of assignment statements.

8.3 The symbex program

The program `symbex` symbolically executes a sequence of assignment or `if`-statements in C/Java syntax and, if an assertion is given as a comment beginning on a new line in the form

`//{` *assertion* `}`,

`symbex` attempts to verify that the condition computed as holding after the preceding statement implies the assertion.

Here is a sample data file:

```
temp = x ; x = y  ; y = temp ;
//{ x='old y' and y ='old x' }
```

This produced the following output:

```
//{ true }
        temp  =  x  ;
        x  =  y   ;
        y  =  temp  ;
        // assert: x=old y and y=old x
        //  -- assertion is verified.
//{ y=old x and x=old y and temp=old x }
```

Like `prover`, `symbex` will search the working directory, and the directories specified in `prolog.ini` for `arithmetic.simp`, `equality.simp`, and `logic.simp`. It also loads a file `simplification.rules` if it exists in the working directory. This file can either contain simplification rules or directives to load specific theory files, written in the form

`theory('`*file specification*`').`

or just

`theory(`*name*`).`

if the file *name*`.simp` exists in the working directory.

The simplification rules are used by `symbex` to verify an assertion by checking that the proposition

condition implies *assertion*

is a tautology, where *condition* is a condition on the program variables which is supposed to hold just prior to the *assertion*, as calculated by symbolic execution.

A pre-condition can be asserted as an initial assertion preceding the Java code. It is assumed to be true at the beginning of the symbolic execution and is combined with the effects of execution.

For example,

```
|: //{ x = a + 2 and y = 7}
|: x = x - y ;  y = x ;
|: //{ y = a-5 }
|: ^D                (^D terminates the input)
//{ x=a+2 and y=7 }
        x =  x-y
        y =  x  ;
        // assert: y=a-5
        //  -- assertion is verified.
//{ a+ -5=y and a+ -5=x }
```

The `symbex` tool also checks whether an assertion is inconsistent with the current state of computation, as in the following example:

```
|: x = x + y; y = x - y; x = x - y;
|: //{ not (y = 'old x') }
|: ^D
//{ true }
        x  =  x + y ;
        y  =  x - y ;
        x  =  x - y ;
        // assert: not y=old x
        // -- assertion is impossible!
        // -- current state is y=old x and x=old y
//{ y=old x and x=old y }
```

This allows us to generate a proposition describing the state at any point in a computation, by inserting the assertion `//{false}` at the point of interest, as in the following example:

```
|: temp =  x ; x =  x*cos(t) + y*sin(t) ;
|: //{false}
|:  y =  temp*sin(t) - y*cos(t) ;
|:^D

//{ true }
        temp  =   x  ;
        x  =   x*cos(t) + y*sin(t)  ;
        // assert: false
        // -- assertion is impossible!
        // -- current state is sin(t)*y+cos(t)*old x=x
and temp=old x
        y  =   temp*sin(t) - y*cos(t)  ;
//{ sin(t)*old x-cos(t)*old y=y and sin(t)*old y+
    cos(t)*old x=x and temp=old x }
```

Exercise 8.1: What test can `symbex` apply to an assertion and a condition which will allow it to determine whether or not the assertion is impossible, given the condition?

Exercise 8.2: Write Java code fragments to compute the following parallel assignments (with as few temporary variables as possible) and use `symbex` to verify the code[11].

(a) huey, dewey, louie = dewey * huey, louie - dewey, huey + dewey;

(b) a, b, c = a + c, b + c, 2 * c;

(c) x, y = x* cos(t) + y* sin(t), x* sin(t) - y* cos(t);

[11] (b) and (c) are taken from [Kubiak, p. 137].

Exercise 8.3: Figure out what expression should be substituted for `'??'` in the following code (cf. [Gries, p. 124]), and use `symbex` to verify that it is correct.

```
//{ c = z + a*b }
   z = z + b; a = '??';
//{ c = z + a*b }
```

(Do not change the pre- or post-conditions.)

Exercise 8.4: If not B implies V = E then the statement

```
if (B) V = E;
```

can be replaced by `V = E;` (assuming everything is well-behaved.)

Use `symbex` to justify this claim.

8.4 Verifying register arithmetic

Consider a computer with 8-bit registers `A, B, C, D`, and the following instructions:

```
L     register1, register2     ; copies register2 into register1.
L     register, memory-location; copies memory location into register.
L     register, constant       ; set register to constant.
A     register                 ; sets register A to (A) plus (register) .
SI    constant                 ; sets register A to (A) minus constant.
```

where `(X)` = contents of `X`. Note that `A` serves as an accumulator for the arithmetic operations.

Each instruction assigns a value to a register, and so can be translated directly into an assignment statement, using variables to represent registers and memory locations. In this way, we can verify a piece of assembler code by translating it into a C/Java assignment statement (for a large program, this could be done automatically by an appropriate tool) and then use symbolic execution. For example, if we let the variable `a` represent the accumulator `A`, the assembler code

```
L A, X
A A
A A
SI 10
```

translates to

```
a = x;  a += a; a += a; a += -10;
```

which can be verified by symbex to be correct with respect to the post-condition

a = 4*x -10 .

Exercise 8.5: The following assembler code is intended to convert a two-digit decimal number stored as two ASCII digits in locations D1 and D2 (high-order digit in D1) to a byte value in register A:

```
L     A, D2
SI    48
A     A
A     A
L     C, A
L     D1, A
SI    48
A     C
```

Translate the code into assignment statements, and use symbex to debug it with respect to the post-condition:

a = 10*(d1 - 48) + (d2 - 48)

Exercise 8.6: The following code is intended to leave the value 100*D1 + 10*D2 in register A. The left-shift operation SH p is used to multiply A by a power of 2 (between 0 and 7).

```
L     A, D1
A     A
L     D, A
L     B, D2
A     B
A     A
L     C, A
SH    2
A     C
L     C, A
L     A, D
SH    3
A     D
SH    3
A     C
```

Use symbex to verify this code, or a debugged version, by translating it into assignment statements. You can represent the effect of a left-shift using C's shift-left operator <<:

a << p = a shifted left p places

8.5 Executing conditional statements symbolically

Executing conditional statements symbolically produces much more complex output, since the state of the program must now be described by a *disjunction*, each of whose disjuncts represents one execution path through the code. Each disjunct contains a condition on the initial variable values which holds in the current program state, and a table of the current variable values.

As an example, if we use symbex to execute (with simplification) the code

```
//{ x = x0 }
   y = x + 2 ;
   if  (y < 0)  y  = x ;
   x = y - x ;
```

we obtain the following result:

```
//{ x = x0  }
        y  =  x + 2  ;
        if (y < 0) //{ x0+2=y and x0=x and x0< -2 }
                y   =  x  ;
        x  =  y - x  ;
//{ x0+2=y and x=2 and not x0< -2 or y=x0 and x=0 and
    x0< -2 }
```

The symbex tool also reports on the state of the computation on each branch of a conditional, and whether the branch can ever be taken, as shown in the following example, in which we verify that the result of the conditional statement is only x = no, regardless of the value of i:

```
|: if ( i != 4 || i != 5 ) x=no ; else x = yes;
|: //{ x = no }
|: ^D
//{ true }
        if ( i != 4 || i != 5 ) //{ not i=5 or not i=4 }
        // This branch is always taken.
                x = no  ;
        else //{ i=5 and i=4 }
        // This branch is never taken.
                x  =  yes ;
        // assert: x=no
        // -- assertion is verified.


//{ yes=x and i=5 and i=4 or x=no and not i=5 or x=no and not
i=4 }
```

In the following example, the question is—does the code compute the maximum of x, y, and z?

```
|: if (x > y) if (x > z) max = x; else if (y > z) max = y;
else max = z;
|: ^D
//{ true }
        if (x > y) //{ y<x }
                if (x > z) //{ z<x and y<x }
                        max  =  x ;
                else //{ y<x and not z<x }
                        if (y > z) //{ false }
                        // This branch is never taken.
                                max  =  y ;
                        else //{ y<x and not z<y and
                                 not z<x }
                                max  =  z ;
//{ z=max and y<x and not z<y and not z<x or x=max and z<x and
y<x or not y<x }
```

Symbolic execution shows that the code needs fixing: one branch of an if-statement is never taken and as a result, the post-condition contains no case in which `max` has the value `y`.

A syntactic hack:

An assertion `//{ . . }` must *precede* a statement (or be at the end of the code fragment.) So you can't insert an assertion at the end of an then-branch or else-branch—the following won't work:

```
if (x > y) x = a; //{ a > y } else x = b;
```

since `else x = b;` is only part of a statement, not a complete statement.

To get around this, put the assertion in front of a null statement as in the following example:

```
if (x > y) {x = a; //{ a > y } ;} else x = b;
```

verify C/Java code fragment using symbolic execution
context: the code fragment consists of simple assignment or `if`-statements, followed by a goal assertion as a comment `//{` *goal* `}`, and possibly preceded by an optional precondition.
method: use `symbex` to compute a post-condition for the fragment. and check whether the post-condition implies the goal assertion.
proof obligations: check whether the post-condition implies the goal assertion.
background: theory files, `simplification.rules`, and the Well-Behaved Expression assumption

Exercise 8.7: Write a C / Java code fragment to compute

x = max(x, min(y,100)) ;

using conditionals and assignment statements instead of the functions max and min.

Write definitions for

max(x, y) and min(x, y)

using < and = . Use these definitions to construct an appropriate assertion whose truth is equivalent to the correctness of the code.

Add this assertion to the code and use `symbex` to verify that the code is correct.

Exercise 8.8: Verify that the following Java code is correct with respect to the specification { y = abs('old y') }:

```
x = x + y; y = y - x;
if (x+y > 0) y = x+y; else y = -x - y;
```

(Cf. [Zelkowitz, 1990, p. 33], [Gannon, 1994, p. 94])

Exercise 8.9: Beginning with version 1.4, Java has included a runtime assertion service, as illustrated by the following example:

```
 /*********************************************
Computes the largest of the three passed distinct
integers.

 @param a, b, c the three integers
 @return their largest
//*********************************************
public static int largest(int a, int b, int c)

// Pre-condition
assert a != b && a != c && b != c;
 int big;
 if (a > b && a > c)
     big = a;
     else if (b > a && b > c) big = b;
          else
// Assertion
        { assert c > a && c > b;
          big = c;

}

//Post-condition
   assert big >= a && big >= b && big >= c;
          return big;
```

Change the assertions into comments in the form used by `symbex`, and input the body of the `largest` function to `symbex` to verify that the post-condition holds for the returned value.

8.6 Procedures with contracts

In C syntax, *procedures* are functions which do not return a value—they are declared as returning the empty type **`void`**. Because no value is returned, procedures are typically used to carry out a computation with *side effects* either on the procedure arguments, or more generally, on the system state. A typical C example: `bzero(&bytes, n)` zeros out `n` bytes, starting with the byte pointed to by the address expression `&bytes`.

We can think of procedures as a sort of generalization of assignment statements, in which the variables which are referenced via the address operator & may be assigned new values by the procedure. (In the example, the `bytes` array gets a new value, in general, as the result of the zeroing.)

Procedures can be incorporated into the symbolic execution of code by giving them a specification in the form of a pre-condition, and a post-condition on the variables that

may be altered by the procedure. The pair of conditions we will call a *contract*, as in Section 7.2.

Consider the procedure

```
void swap(&X, &Y);
```

which is intended to exchange the values of X and Y. A contract for such a procedure is given by:

pre-condition: X = 'old X'
post-condition: X = 'old Y' and Y = 'old X'.

The contract for a procedure defines its effect and allows us to execute the procedure call symbolically without having to execute its internal code. In symbex, this is accomplished by specifying procedure contracts at the beginning of the code. A contract is specified in the following form:

```
//{ pre-condition }
void procedure(parameters);
//{ post-condition }
```

The middle line is a *prototype* for a procedure call. In the prototype, the parameters are represented by ***upper-case*** *variables*, which are used much as upper-case variables are used in simplification rules, to match against the arguments in a procedure call. For the purposes of verification, the prototype, together with the pre- and post- conditions, can be considered as a comment or annotation in the code—we could write the contract all on one line:

```
//{ pre-condition } void procedure(parameters); //{ post-condition }
```

The contract for a procedure must have a pre-condition; if no specific condition is required, use `//{ true }`.

Since arguments to C functions are always call-by-value, the only way for a function to have a side-effect on a variable is if the address of the variable is passed to the function; otherwise, the function cannot alter the variable. So we know that, in the previous example, the bzero procedure cannot alter the second argument n. To identify which arguments are passed as addresses, symbex requires that such arguments are passed using the address operator &

In arguments to Java functions, the address operator is not used since all object arguments (including arrays) are referenced, i. e. the argument is passed as an address, and all other values can only be passed as call-by-value. But as symbex doesn't know the type of the function arguments, it cannot symbolically execute a function call in Java unless the address operator is used in the input to symbex to indicate which arguments are referenced.

Note that in the contract, the call-by-address parameters are not given; the contract is specified in terms of derefenced parameters, using upper-case variables. For example,

```
//{ true } void bzero(B, N);
//{ 0 <= i < N implies array(B, i) = 0 }

 bzero(&id, 3);

//{  array(id,  3) = 0 }
```

specifies a contract for `bzero` in terms of the dereferenced parameter `B` and the call-by-value parameter `N`, and asserts a post-condition for the `bzero` procedure call in terms of the corresponding arguments `id,` and `3.`

In the following example, `symbex` verifies that if the length of a text array is initially within the allocated amount, then the code safely adds a character at the end and preserves the condition length <= allocated:

The input file `addchar.test` is:

```
//{ Length < allocated} void addchar(Text, Length, C);
//{ Length = 'old Length' + 1 and array(Text, Length) = C }

//{ length <= allocated }
if (length == allocated) allocated *= 2;
addchar(&text, &length, c);
//{ length <= allocated and array(text, length) = c }
```

and the output from `%symbex <addchar.test` is:

```
void addchar(Text, Length, C);
//{  Length = 'old Length' + 1 and
     array(Text, Length) = C  }


//{ length <= allocated }
        if (length == allocated) //{ length=allocated }
           allocated  =  allocated * 2 ;
        addchar(&text, &length, c);
        //assert: array(text,length)=c and length=allocated
                  or array(text,length)=c and
                  length<allocated
        // -- assertion is verified.

//{ array(text,length)=c and old length+1=length and
    old allocated*2=allocated and old allocated=length or
    array(text,length)=c and old length+1=length and
    length<allocated }
```

(The simplification rules used to simplify the post-condition are left as an exercise for the reader.)

What happens if symbex cannot show that a pre-condition for a procedure holds? As seen in the following example, symbex continues with the symbolic execution under the assumption that the post-condition holds. This allows it to test further conditions, such as an implication between a post-condition and a following assertion:

```
//{  online(Printer)  }
void setup(Printer);
//{  ready(Printer)  }


//{ true }
        setup(laser3);
        ... cannot show online(laser3) for setup(laser3).
        // assert: ready(laser3)
        // -- assertion is verified.

//{ ready(laser3) }
```

Although symbex reports that the assertion is verified, this only means that the post-condition of setup implies it; in the absence of being able to assert the pre-condition that the printer is online, the assertion that the printer is ready may be false.

verify C/Java code fragment using symbolic execution and procedure contracts
context: the code fragment consists of simple assignment statements, if-statements or procedure calls, followed by a goal assertion as a comment //{ *goal* }. The code is preceded by contracts in the form //{ *pre-condition* } void *procedure*(*parameters*); //{ *post-condition* } for each procedure called in the code, and an optional precondition. method: for each procedure call, symbex checks whether the pre-condition in the procedure's contract is satisfied, and instantiates the contract's post-condition, which becomes the pre-condition for the subsequent code. A post-condition for the fragment is computed. proof obligations: The fragment is verified if all the pre-conditions to procedure calls are satisfied and the post-condition for the fragment implies the goal assertion. background: theory files, simplification.rules, the Well-Behaved Expression assumption, and the assumption that no procedure call alters any variables other than those passed in as call-by-address (&*V*) arguments.

Exercise 8.10:

(a) The code in the example above must have been verified with rules that assumed that the value of allocation was positive. Use `symbex` to show that if the initial condition for the code is length <= allocated and allocated = 0 then the pre-condition for the `addchar` procedure cannot be verified.

(b) Suppose we keep the specification for the `addchar` procedure the same as in 8.6. Consider a code fragment which tests if allocated <= size, and if it is, increments the allocation by the amount size. Use `symbex` to show that if this is followed by the procedure call `addchar(&string, &size, ch)`, the procedure's pre-condition cannot be verified, without further conditions.

(c) Add an initial condition to the code in (b) which allows you to verify the post-condition: size <= allocated and array(string, size) = ch .

Exercise 8.11: Implement a code fragment for a settop box application that tries to reallocate a purgeable screen, as described in Section 7.1, and use `symbex` to verify that it achieves the intended post-condition. Suggestion: define a contract for **scr_IsPurgeable()** with the post-condition scrIsPurgeableCalled; and use simplification rules to map conditions involving the event codes returned by **getNextEvent()** into the Boolean variables used in the specification. Since the description of the settop box application does not give an action for the application to execute if a purgeable screen is allocated, represent the allocation by including a variable in the code whose value is set to 1 if the allocation is performed, and 0 otherwise. This allows you to map the allocation action to the specification condition purgeable required in the post-condition. Notice also that the assertion to be proved to verify the code must be attached to the right *place* in the code, unless it includes the case of no purgeable screen being available.

9. COMPUTING WEAKEST PRECONDITIONS

In Chapter 8, we explored the symbolic execution of loop-free code, propagating a pre-condition in the forward direction through the program statements to compute a post-condition. There are major weaknesses in this approach:

- symbolic execution requires separate tracking of logical values and variable values, complicating the handling of other statements than assignment and simple conditionals. Loop constructs cannot be treated directly, but have to be translated into another form such as a recursive function (cf. [Gannon, 1994, Ch. 3.]);

- sequences of conditional statements generate post-conditions which grow exponentially with the number of the conditions, because they record the state of execution for every possible path through the code;

- the computed post-condition typically contains a great deal of information about the final state of the program which is irrelevant to the goal which the code is supposed to achieve. The verification of the code therefore involves a further step of showing that the post-condition implies the goal—a task which may be difficult due to the length and complexity of the post-condition.

It turns out that the reverse approach—the propagation of a desired goal **backwards** through the code to compute a pre-condition which ensures that the code achieves the goal assertion—produces simpler and more useful results.

Our focus in this chapter is on the specific problem of computing a pre-condition P for a code fragment S which achieves a post-condition Q, which we write as follows:

{P} S {Q}.

Note that an expression of this form is a proposition; it is true if P implies that following the execution of S, Q holds. In this case, we can say that S is *correct* with respect to the specification or "contract" {P} _ {Q}.

9.1 The strength of pre-conditions

If P implies X, and X is not equivalent to P, we say that P is **stronger** than X. One way to think about the concept of 'strength' in the context of verifying a program fragment is to identify the pre-condition and post-condition in {P} S {Q} as *sets* of program states (determined by the values of variables) such that whenever the program is in a state in P (i. e., P is true), then after executing S, the program state is in the set Q, i. e., Q is true[12]. So if P implies X, this means that whenever P is true of a state, then X is true of that state, so the set of states corresponding to P are *contained* in the set X.

[12] In our use of {P} S {Q}, S is required to *terminate*, so that program segment actually ends in a state in Q. S is then *unconditionally* correct with respect to the conditions P and Q. This deviates from [Backhouse, 1986] who treats termination separately, so that {P} S {Q} only means that S is correct *conditionally*, the condition being that it terminates. Backhouse's *conditional correctness* (see Sec. 11.6) is the same as the original notion of *partial* correctness in [Hoare, 1969], although Hoare put the braces around the statement rather than the conditions. The concept and notation used here is that of [Gries, 1981] who refers to it as *total* correctness.

Thus the stronger condition (P) is associated with the 'smaller', more restricted set of states; the weaker condition (X) is associated with the 'larger', less restricted or more general set of states [13]. The weakest possible proposition is the constant true since it is implied by any proposition; the strongest is the constant false.

Programmers and clients of programmer have opposite interests with respect to logical strength; a lazy programmer would prefer that the pre-condition of a program be as strong as possible, since it is very easy to write a program for any goal (post-condition) if the pre-condition can be made sufficiently strong. If the pre-condition is the goal itself, the program can be the null statement ";" since there is no need to compute anything, and if the pre-condition is set to be the absolutely strongest proposition false[14],the programmer can submit any program she likes—the pre-condition can never be met, so whatever the program does, it will meet the specification.

Notice that the situation is reversed for goals: clients prefer strong goals which specify exactly what they want the program to do in every detail. Programmers prefer weak goals which are easy to achieve. For example, the goal true is achieved by any program that halts, in particular the null program ";". The worst thing a client could do to a programmer would be to specify a goal equivalent to false, for now no matter what the programmer does, she can never achieve the goal—nothing can make false true!

Clients, in contrast to programmers, prefer weak pre-conditions, since if the pre-condition is weak, the program is more general; if it is correct, it will produce the desired output for a wider range of initial states. In fact, in many cases, the client will prefer the weakest possible pre-condition true, so that no special setup or system initialization is required for the program to function correctly.[15] For this reason, we will recast the verification problem into a new form: that of calculating for a program segment S and a goal Q, the most general, that is, the *weakest* pre-condition for S {Q}, which we will call

wp(S, Q).

It follows by definition that if the proposition wp(S, Q) holds as an initial condition, then S is necessarily correct with respect to the goal Q.

Any other condition C can be easily checked to see if it is a pre-condition for S {Q} by checking whether it implies wp(S, Q). If it does, then clearly C is a pre-condition, since

---

[13] Some authors define a condition as being stronger if 'fewer' states satisfy it (cf. [Backhouse, 1986, p. 88], [Gries, 1981, p. 16], [Dromey, 1989, p. 14].) Strictly speaking, this is a mistake, at least if 'fewer' is interpreted in terms of number or cardinality. One set may be properly contained in another and still have the same number (in terms of cardinality) of elements as the containing set if the sets are infinite. It is better to interpret 'strength' strictly in terms of logical implication or set containment, rather than in terms of 'size' of the sets corresponding to the conditions.

[14] For false implies every proposition; it corresponds to the empty set of states which is contained in any set.

[15] (For more on the differing logical interests of programmers and clients, see [Morgan, 1992] and [Brood, 1994, pp. 27-29.]) If we return to the concept of refinement described in Section 7.3, we find in the simple case, where the new pre-condition is weaker than the old one, and the new post-condition is stronger than the new one, that refinement agrees—as we might expect—with the interests of the client. Clients will prefer the refined program to the original since it delivers more with weaker initial assumptions.

when it is true, wp(S, Q) is true, and by definition, wp(S, Q) guarantees that Q holds after executing S. If C does not imply wp(S, Q), then there are some program states in which C is true but wp(S, Q) is false, which implies that in those states the goal Q is not achieved by executing S, so C cannot be a pre-condition.

By calculating the weakest pre-condition, we reverse the procedure implemented in `symbex`; here we take the post-condition or goal of a program fragment as given, and work out the most general condition which would have to be true initially for that program to be correct with respect to the given goal.

9.2 Computing wp(S,Q) for loop-free code

For some simple statement types, the weakest pre-condition wp(S, Q) can be calculated relatively easily using rules first worked out by C. A. R. Hoare [Hoare, 1969]:

null statement

We will represent the null or empty statement by "**;**". Then

wp("**;**", Q) ≡ Q

since the null statement does not change the program state.[16]

sequence of statements

wp("S_1 S_2 . . S_n", Q) ≡ wp("S_1 S_2 . . S_{n-1}", wp("S_n", Q))

Each statement up to the last has as its goal the weakest pre-condition which will ensure that the following statement achieves *its* goal, and the goal for the last statement is the goal for the entire sequence.

conditional statements

wp("**if (**B**)** S_1 **else** S_2", Q) ≡
B and wp(S_1, Q) or not B and wp(S_2, Q)

wp("**if (**B**)** S_1", Q) ≡ B and wp(S_1, Q) or not B and Q

wp("**switch (**C**) {**
case L_1**:**S_1**; break;**
case L_2**:**S_2**; break;**
. . .
case L_n**:**S_n**}**", Q)
≡
C=L_1 and wp(S_1, Q) or
C=L_2 and wp(S_2, Q) or

[16] The ≡ here is the same as iff; we use it to indicate that we are *defining* the value of the wp predicate for specific arguments.

. . .
C=L_n and wp(S_n, Q)

For each of these equations to be valid, we need to make the same assumption as in Chapter 7, that the expressions in the conditions (B and C) are 'well-behaved'—in particular, that their evaluation terminates without side-effects.

Exercise 9.1: Use wang to show that

wp("**if (**B**)** S_1", Q) iff B implies wp(S_1, Q) and not B implies Q

Exercise 9.2: Write an equivalence parallel to the one given above for the n-case switch statement, for a switch statement with a default case:

wp("**switch (**C**){**
case L_1**:**S_1**;** break;
case L_2**:**S_2**;** break;
. . .
case L_n**:**S_n**;** break;
default S**}**", Q) ≡ ?

assignment

wp("R **=** Exp**;**", Q) ≡ Q[Exp / R]

For example,

wp("**index = index - 2;**" , a = (1+index) / index) ≡
a = (index - 1) / (index - 2).

For this equation to be valid, R must be a simple variable (not an array or record variable), and the expression Exp must be well-behaved.

Exercise 9.3: Show that applying the equation for the assignment statement to the specification

a[i+3] = 7; {a[4] = x}

will give an erroneous 'weakest pre-condition'.

assignment to an array element

Despite Exercise 9.3, we can treat assignment to array elements as a special case of the equation for the assignment statement, if we interpret the assignment to an element as the assignment to the array variable of a new *array value*. That is, we treat

A[Index] = Exp; (where Index and Exp are well-behaved expressions)
as if it were written:

A = change(A, Index, Exp) ;

in which the variable A is assigned a new array value given by the function change(A, Index, Exp);. The new value is the original array A with the one element A[Index] altered to the value Exp. Under this interpretation, we can use the previous rule for simple assignment statements to calculate the weakest pre-condition of an assignment to an array element:

wp("A[Index] = Exp;", Q) ≡ Q[change(A, Index, Exp) / A]

9.3 The wp program

The program wp is a tool for calculating weakest pre-conditions of C/Java code fragments. The program can be invoked interactively, as shown in the examples below[17].

Initial conditions and post-conditions are added to the code fragment in the form of comments //{ *condition* }. A post-condition is required; an initial condition is optional.

If all the statements in the code are of the simple types given above, then the weakest pre-condition is calculated and labelled **PRE** in the output.

```
% wp
Version 1.5.5, March 14, 1999
|: x = x+1; y = y+1;
|: //{ x = y }
|: ^D        typing ^D terminates the program fragment.

// PRE: y+1=x+1


|: x = x * y;
|: //{ x*y = c }
|: ^D
// PRE: y*y*x=c


|: x = 2*y + 3;
|: // { x = 13 }
|: ^D
// PRE: y*2=10
```

[17] adapted from [Gries, 1981, p. 120] and [Backhouse, 1986, p. 94].

```
|: x = (x-y)*(x+y);
|: //{x + y**2 <> 0}
|: ^D

// PRE: x**2<0 or 0<x**2

|: b[i] = i;
|: //{ b[b[i]] = i }
|: ^D
// PRE: true
```

Each fragment is terminated by an end-of-file ^D; only the first ^D is shown is shown explicitly in these examples, and some of the output from the Prolog interpreter is omitted.

(A second ^D exits from the input loop.)

Exercise 9.4: Explain how the weakest pre-condition for the last code fragment was obtained, in terms of the rule for assignment to an array element, and simplification rules for arrays.

As its first step, wp checks that the goal following a code fragment cannot be shown to be a contradiction, for if it were, there is no pre-condition which could achieve it. For example:

```
|: ;            null statement
|: //{ 1 = 2 }
|:
!! ** Goal is a contradiction.  There is no pre-condition.
```

The weakest pre-condition for the goal is calculated using the rules for each statement type, and the resulting proposition is then simplified. The simplification rules are loaded from arithmetic.simp, equality.simp, array.simp, and logic.simp, found either in the user's working directory or in the default directory specified in the initialization file prolog.ini, in the same way as in the symbex tool. In addition, a file called simplification.rules is loaded if it exists in the working directory.

One-dimensional C/Java array expressions are allowed on the left side of an assignment statement, and in expressions.

Allowable non-looping statement types for the current version of wp are:

V **=** Expression **;**
if (B**)** statement
if (B**)** statement1 **else** statement2
{ sequence of statements **}**

The allowable forms for Expression are rather limited in the current version: simple variables, array expressions, function terms, and arithmetic expressions.

The specification for a program fragment can include a pre-condition as well as a post-condition or goal. In that case, the fragment S is verified with respect to the specification {P} S {Q} by a proof that P implies wp(S, Q). wp checks this and if it is able to establish that the implication is a tautology, it reports that the initial condition will achieve the goal as shown in the following example:

```
|: //{ x = 10 }
|: y = x - 1;
|: //{ y > 0 }
|:
// PRE: 1<x
Initial condition achieves the goal.
```

If P implies wp(S, Q) cannot be proved, this does not mean that the code is incorrect — it may be that the simplification rules used are not strong enough to establish the implication. So in this case, wp just reports that the initial condition may not be compatible with the goal —as in the example below, where the simplification rules which have been loaded are insufficient to show that x + y = 0 implies a - y = x + a is a tautology.

```
|: //{x + y = 0 }
|: x = x + a;
|: y = y - a;
|: //{ x + y = 0 }
|:
// PRE: a-y=x+a
Initial condition may not be compatible with the goal.
Cannot prove -y=x implies a-y=x+a.
```

Exercise 9.5: Use wp to verify the following specifications (from [Gries, 1981, p. 124] and [Backhouse, 1986, p. 100, 103]):

(a) { true} "**b = a + 2; a = a + 1;**" { b = a +1 }

(b) {i = j} **"j = j+ 1; i = i + 1;"** {i = j}

(c) {i = j+1} **"if (i > j) j = j + 1; else i = i + 1;"**
{i >= j}

(d) {m = i*j + k + 1}
"if (j == k + 1) {i = i + 1; k = 0; }
else k = k + 1;"
{m = i*j + k}

Exercise 9.6: Rework Exercise 8.8 using wp instead of symbex.

Exercise 9.7: Use wp and appropriate simplification rules for the max function to show that the following code is correct:

```
if(m < y) m = y; //{ m = max(m, y)}
```

Exercise 9.8: Use wp to determine whether the following code fragment is correct or not. (Explain your answer.)

```
|: x = x+y; y=x-y; x=x-y;
|: //{y='old x' and x='old y'}
```

Using wp to solve for unknown expressions

As Edward Cohen ([Cohen, 1990]) has pointed out, a weakest pre-condition calculation can be used to solve for unknown expressions in assignment statements. A simple example:

```
|:  a = a + c; q = '??'; //{ q = a*c }
|:

// PRE: c*c+c*a= ??
```

Here is a more complex example. Consider the calculation of the real roots of the quadratic equation

$$ax^2 + bx + c = 0$$

with $a \neq 0$ and $b^2 - 4ac \geq 0$, so that the discriminate $d = sqrt(b^2 - 4ac)$ is real.

The naive calculation

```
x1 =  (-b - d)/(2*a); x2 = (-b + d) / (2*a);
```

runs into accuracy problems which are avoided if the larger root in absolute value is computed first:

```
if (b >= 0) x1 = -(b + d)/(2*a);
else x1 = (d - b)/(2*a);
```

and the second root is then computed from it:

```
x2 = '??'/x1;
```

We have left the numerator as an unknown quantity to be determined from the requirement that the set {x1 , x2} consists of the two roots, which gives us the post-condition:

x1 = -(b + d)/(2*a) and x2 = (d - b)/(2 * a)
  or

x1 = (d - b)/(2 * a) and x2 = -(b + d)/(2*a).

With simplification, we obtain the following pre-condition using wp:

```
// PRE: (d-b)/ (a*2)= ?? / ((d-b)/ (a*2))and d=0 and b<0 or (-
(d+b))/ (a*2)= ??
 / ((- (d+b))/ (a*2))and d=0 and 0<=b or
(d-b)/ (a*2)= ?? / ((- (d+b))/ (a*2))and 0<=b or
(- (d+b))/ (a*2)= ?? / ((d-b)/ (a*2))and b<0
```

This reduces to

$(-b/(2a))^2 = ??$ and $d = 0$ or $((b - d)(b + d)/(2a))^2 = ??$

which is equivalent to

$?? = ((b - d)(b + d)/(2a))^2 = (b^2 - d^2)/(4a^2) = 4ac/(4a^2) = a/c.$

In effect, we have calculated and verified *Vieta's formula*:

x2 = c/(a * x1).

Exercise 9.9 (a) Use wp to solve for ?? in the following (adapted from [Cohen, 1990, p. 112]):

{ r = n*n } n= n+1; r= '??'; { r = n*n }

Eliminate the reference to '??' by substituting the solution into the code, and show that the code only works correctly if n is initially -3/2.

(b) Rework (a) with the assignment statements reversed. What can you now conclude?

9.4 Reasoning about actions and their consequences[18]

As a brief digression, we consider an application of pre-condition calculation outside the domain of software verification. Researchers in artificial intelligence (AI) have long been interested in how programs could be equipped to reason about the effects of actions applied to objects. Typically, a special language is created to represent the state of the system, the actions that can occur, and their results. For example, in one scenario[19] which became the subject of numerous articles in the AI literature, we are asked to imagine the initial circumstance of an unloaded gun and a live turkey, followed by the action of waiting for the turkey to appear, aiming the gun and shooting it at the turkey. Using common-sense reasoning, we might then conclude that the turkey is dead. The problem for artificial intelligence is to work out computational processes for making such inferences.

Actions in such scenarios can be presented by procedures which alter the collective state of the objects in the scenario—for this example, let's assume three procedures *load*, *wait*, and *fire* are defined. In order to describe the states of the scenario, we will need some predicates—say, turkeyAlive and gunLoaded. A scenario can then be presented by a pre-condition and a series of procedure calls:

[18] This section is based on [Lukaszewicz, 1994] but the details are somewhat different.

[19] Hanks, S., and D. McDermott, "Nonmonotonic Logic and Temporal Projection", *Artificial Intelligence* **33**, 1987, pp. 379-412.

```
//{  turkeyAlive and not gunLoaded }
load; wait; fire;
```

What else is needed to infer the post-condition not turkeyAlive? If we give a specification for each of the procedures in terms of pre- and post-conditions, then we can see if the weakest pre-condition for not turkeyAlive with respect to the scenario implies the stated pre-condition; if so the inference is valid (relative to the specifications.)

So suppose we specify load, wait, and fire as follows:

wp("load;", gunLoaded) ≡ true.

wp("wait;", X) ≡ X.

wp("fire;", not turkeyAlive) ≡ gunLoaded.

wp("load; wait; fire;", not turkeyAlive)
≡ wp("load; wait;", gunLoaded)
≡ wp("load;", gunLoaded)
≡ true.

Exercise 9.10 The reasoning about actions in the preceding example is relative to a scenario which provides a *simulation* of real actions; the scenario is not the real thing. It could happen in real life that the gun is loaded; the hunter waits until the turkey appears, fires the gun and—the turkey squawks, flaps its wings and hurriedly takes itself off. In real life, we would then infer that something happened not covered in the specifications. For example, someone may have unloaded the gun while the hunter was waiting.

Modify the specifications for one or more of the actions in the above example to allow for some of the possibilities, by adding more cases to the definitions of wp. For example,

wp("fire;", turkeyAlive) ≡ gunNotFunctioning..

Add conditions which take into account the possibility that the state of the scenario may not be the same after a wait as it was at the beginning.

With your changes to the scenario, under what pre-condition can one still infer that the turkey is dead after the gun is fired?

Exercise 9.11: The Deaf Turkey Scenario. Modify the turkey scenario so that when the gun is loaded, and the turkey is present, it hides (unless it is deaf). The action `fire` now only kills the turkey if the gun is loaded and the turkey isn't hidden.

Replace the procedures load and fire with assignment and conditional statements which set the values of appropriate variables, e. g. "`fired = true;`" to record that the gun was fired, and construct a code fragment which simulates the modified scenario. Use wp to verify that the weakest precondition for the turkey's survival is that it be not deaf.

9.5 Properties of {P} S {Q}

In 9.1, we interpreted the pre-condition and post-condition in {P} S {Q} as *sets* of program states. The sets are related by a state-transition relation— the relation induced by the code which causes a state in P to transit to a state in Q. Since starting in a state in P guarantees that the final state is in Q, if {P} S {Q} is true, then Q considered as a set must contain *all* the states which are reached from states in P by executing the code.

Conversely, wp(S, Q) corresponds to the set of all states which transit to a state in Q when S is executed.

the impossible pre-condition

The contract {false}_{Q} is satisfied by any post-condition Q, for, as no state is in the empty set, every state in the empty state transits to Q.

the termination post-condition

The contract {P}_{true} is satisfied by any pre-condition P for a statement (or sequence of statements) which terminates—that is, which, starting in P, makes a transition to *some* state.

Thus, wp(S, true) corresponds to exactly those initial states in which S terminates. Conversely, if wp(S, true) ≡ false, then S "hangs" or "goes into an infinite loop" on all initial states.

and-distributivity

Another property of wp which is quite intuitive is *and-distributivity*:

wp(S, Q and R) ≡ wp(S, Q) and wp(S, R).

For if, when S is executed, an initial state transits to a state for which Q and R are true, then that initial state is a state which transits to a state for which Q and for which R is true—and conversely.

Exercise 9.12: Why is wp(S, not Q) not equivalent to not wp(S, Q) ?
Give a counter-example.

Exercise 9.13: The formula

wp(S, false) ≡ false

has been called the "Law of the Excluded Miracle". [Djikstra, 1976].
Discuss what properties of wp might inspire this characterization.

Exercise 9.14: Give an argument to show that if A implies B, then

wp(S, A) implies wp(S, B)

9.6 Pre-conditions for `while`-statements

Statements which loop, such as the `while` and `repeat` statements, are the most difficult to verify.

A formula can be given for

wp("**`while (`**B**`) do`** S", Q).

in the form

there exists n >= 0 such that P_n,

where P_0 ≡ (not B) and Q, and P_n ≡ B and wp(S, P_{n-1}). But this form is not very practical; just to determine whether the pre-condition is not false (that is, whether the statement halts), we would have to check a possibly unbounded number of P_n.

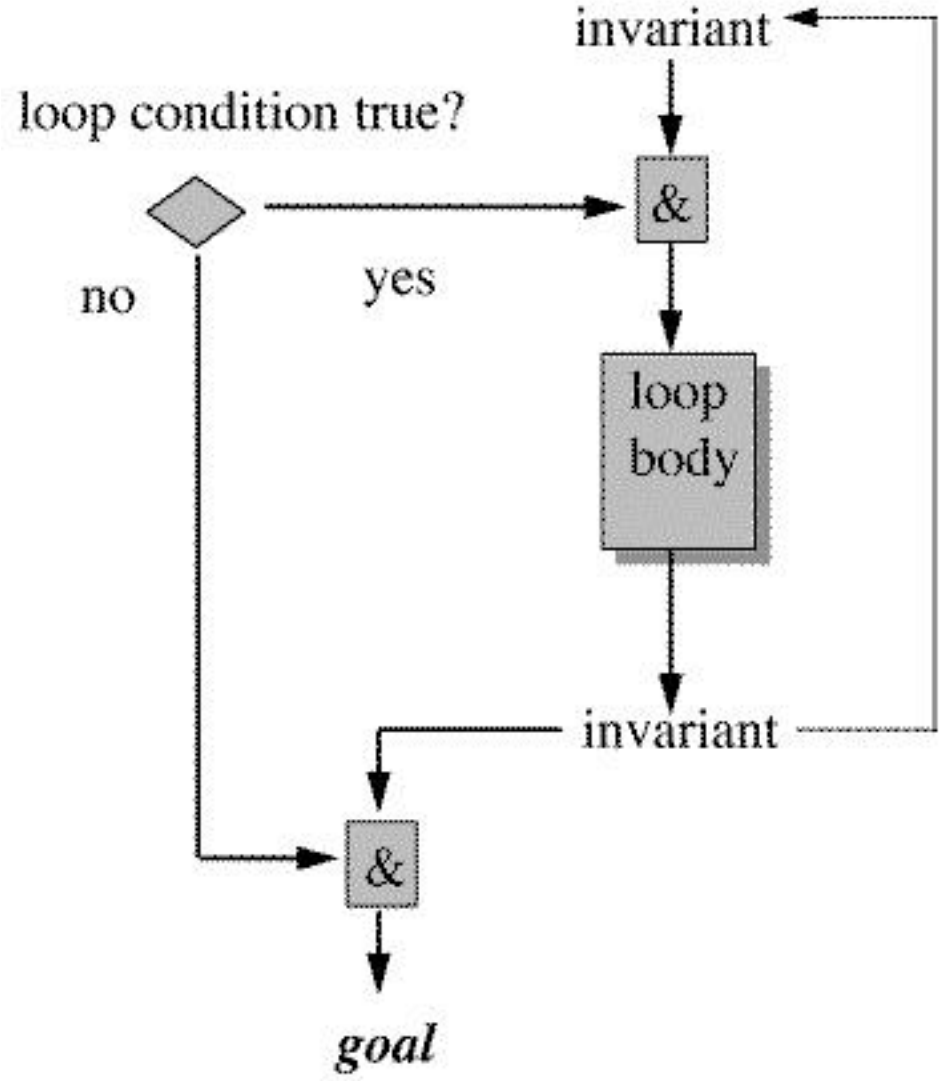
While we may not be able to calculate directly a usable *weakest* pre-condition for a `while` statement, we can, if given a *loop-invariant* I, calculate a useful pre-condition, using the following rule, known as the *Fundamental Invariance Theorem* (a proof is given in the Appendix):

Let W = "**`while (`**B**`)`** S". If I and B implies wp(S, I) and I and not B implies Q, then I and wp(W, true) implies wp(W, Q), so

{I and wp("**`while (`**B**`)`** S", true) } **`while (`**B**`)`** S {Q}.

In words, if I is a pre-condition for itself with respect to S, and implies the goal Q on loop exit (or if the loop is not entered), then I, as a pre-condition for the loop, ensures that the loop achieves Q if it terminates.

This is perhaps more easily understood in terms of a diagram which illustrates how loop-condition, invariant, and goal fit together like pieces in a puzzle to establish the invariant as a pre-condition:



Since in this case, I is only a pre-condition for Q, not necessarily the *weakest* pre-condition, we cannot use it in the rules given earlier for calculating the wp of compound statements. So, to verify a program fragment containing loops, we define a new function pre which can be used to compute a pre-condition for compound statements with or without loops.

The function pre is required to satisfy the following conditions:

pre(S, Q) = wp(S, Q) if S contains no loop statement;

pre(S, Q) = I if S = "**while (**B **) //{** I **}** S_1" and
{I and wp(S, true) } S {Q}.

We can now compute

pre("S_1; S_2 . ; S_n", Q) ≡ pre("S_1; S_2 . . ; S_{n-1}", pre(S_n, Q)).

In other words, we just replace wp by pre in the previous rules for loop-free fragments.

Exercise 9.15 Give a pre-condition for

```
do S while (B) ;
//{Goal}
```

in terms of B, and pre("`while (B) //{I} S`", Goal).

Note that the value of pre(S, Q) depends on the invariant specified for the loop. Any invariant will do to define pre; but some invariants may work better than others in allowing us to easily establish a meaningful pre-condition for the entire code segment. Unfortunately, there is no complete mechanical procedure available to compute invariants. The invariant has to be invented either by the programmer when the code is being developed, or subsequently by a person acting as verifier analyzing, the code. A useful heuristic is to look for an invariant which seems to capture the *intended purpose* of the loop, by leading to the goal when the loop condition is false. Good discussions of how to develop invariants are found in [Backhouse, 1996, Ch. 4] and [Gries, 1981, Ch. 16].

The program `wp` requires that an invariant be supplied for each `while`-statement following the loop condition. Given a sequence of statements of the form

```
statements1
while (B)
//{ I }
  S
statements2
//{Q}
```

`wp` calculates G = pre(statements$_2$, Q) as the goal for the `while`-statement and attempts to prove that I and not B implies G. It also attempts to prove that I is an invariant—that I and B implies wp(S, I). Assuming that these checks succeed, it takes I as the goal for statements$_1$ and reports the pre-condition pre(statements$_1$, I).

In the following example, the simplification rules supplied were not sufficiently strong to establish the proposed invariant, but under the assumption that it is indeed an invariant, `wp` displays the pre-condition which is sufficient to ensure the post-condition and verifies that it is implied by the initial condition:

```
|://{ y >= 0 and y= a and x = b }
|:
|: z = 0;
|: while (y != 0)
|: //{ x*y + z = a * b }
|: { z = z + x; y = y - 1;}
|:
|: //{ z = a*b }
y*x+z=b*a may not be an invariant.
Cannot verify y*x+z=b*a and y<0 or y*x+z=b*a and 0<y implies
(y-1)*x+z+x=b*a
// PRE: y*x=b*a
Initial condition implies the pre-condition.
```

It is possible for the code to be correct relative to a given initial condition, even though for the specified invariant, the "boundary condition" I and not B implies Q is not valid (in other words, the verification failure is not just a matter of stronger simplification rules). What is required in this case is an initial condition C, such that

C and I and not B implies Q.

For example, suppose we strengthen the loop condition in the above example to y > 0. Now we get the following output from wp (using the same simplification rules):

```
x*y+z=a*b may not be an invariant.
Cannot verify y*x+z=b*a and 0<y implies (y-1)*x+z+x=b*a
x*y+z=a*b may not be a precondition for the WHILE-loop goal.
Cannot verify y*x+z=b*a and y<=0 implies b*a=z.

// PRE: y*x=b*a
Initial condition achieves the goal...
```

The failure to verify the invariant is, as in the previous example, a matter of insufficient rules, but the reason that y*x+z=b*a and y<=0 implies b*a=z cannot be verified is that it is, in fact, not a theorem of arithmetic without further assumptions. However, as the final message indicates, the initial condition ($y \geq 0$) insures that, when the loop terminates or is skipped, $y = 0$, so that the goal is achieved.

9.7 Variant functions and proof of termination

There is still one more step required to complete the verification of a `while` loop with respect to a proposed invariant; we must show that the loop *terminates*. This is done by finding a *variant* function (sometimes called a *bound* function) of the variables in the loop-body; this should be a function whose value changes at each execution of the loop in such a way that we can guarantee that the loop must eventually terminate.

More precisely, we define a function v(Variables) to be a *variant* for the loop

```
while (B) //{ I }
S
```

if there exists a fixed $\Delta > 0$ such that initially, v(Variables) ≥ 0, and

{B and I } S { v(old(Variables)) - v(Variables) ≥ Δ }

and

I and v(Variables) ≤ 0 implies not B.[20]

Thus at each iteration of the loop, the variant decreases by at least Δ, and so if the value of the variant is $V \geq 0$ prior to executing the loop, v(Variables) must be ≤ 0 after V/Δ iterations, at which point, the loop condition is false, and the loop terminates.

Another way to think of the variant function is as a clock which gives a bound on the time remaining in the execution of the loop, with each iteration taking at least Δ time units. When the clock 'runs out' (the time showing is ≤ 0), then the loop terminates.

For simplicity, we will restrict variant functions to be *integer-valued*, and choose

$\Delta = 1$.

To use wp to verify termination, we specify a `while`-statement in the form

```
   while(B)
//{ invariant(I) and variant(v(Variables))}
    S
```

Termination is verified if we can establish that

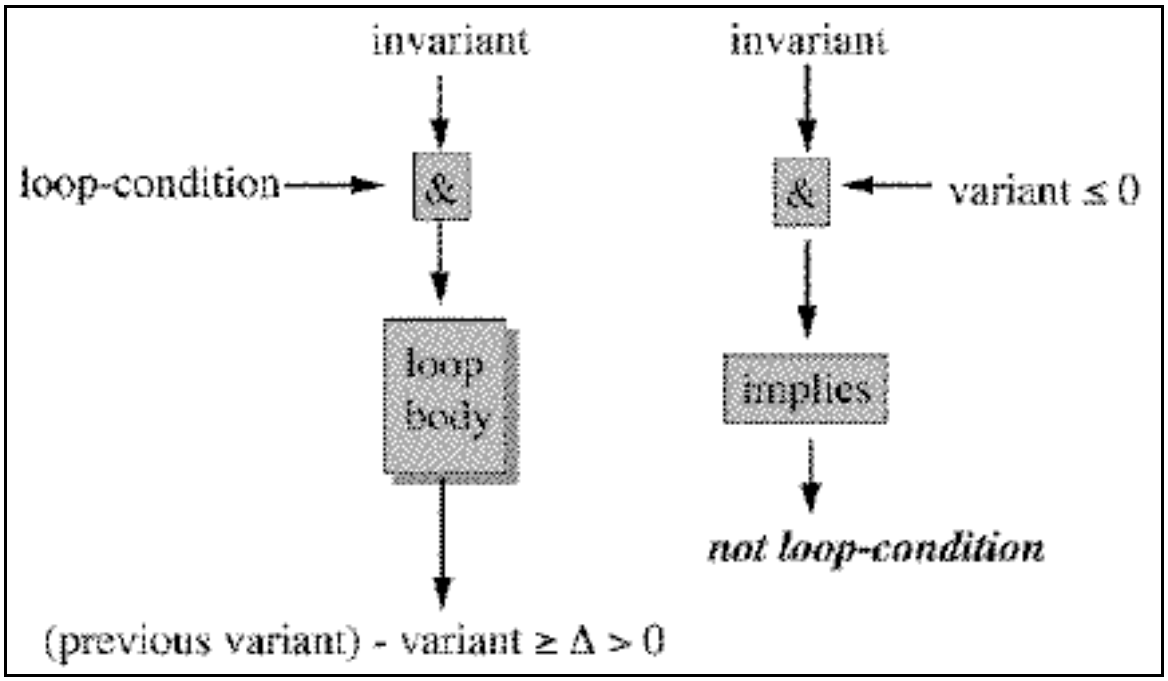
{B and I } S { v(old(Variables)) - v(Variables) >0 }

and

I and v(Variables) ≤ 0 implies not B.

Represented diagrammatically:

[20] Variables stands for the list of variables of which the variant is a function, and old(Variables) is shorthand for the list of values of these variables prior to executing S, which we will assume are in the form 'old *V*'.



As an example, consider

```
//{x >= 0}
while ( x != 0 )
//{ invariant(x+y = 'old x' + 'old y') and variant(x)}
      { x= x-1; y = y+1; }

//{ y = 'old x' + 'old y'}
```

The output from `wp` is:

```
x may not terminate loop.
Cannot verify y+x=old y+old x and x<=0 implies x=0
// PRE: true
Initial condition is compatible with the goal.
```

`wp` was able to verify that the proposed variant `x` decreases on each iteration, but was not able to show that when it goes negative, the loop halts. This illustrates a problem with verifying termination; the loop may indeed terminate, as in this case, but the *invariant* specified may not be strong enough to establish it. In general, we can only expect to prove termination if the invariant captures sufficient information about the state of the variables involved in a proposed variant; in this example, there is more about `x` that needs to be included in the invariant, namely that $x \geq 0$.

Exercise 9.16: Use wp to show that if $x \geq 0$ is added to the invariant in the preceding example, the conjunction is also an invariant and the function x now satisfies the definition of a variant function for the loop.

Another example, in which a combination of program variables is used to provide a variant:

```
while (k < m)
//{invariant(y = x**k ) and variant(m - k)}
  {
     y = y*x; k = k+ 1;
}
```

To show that this terminates, we need to establish that

```
{'old k' < 'old m' and y = x**'old k'}
y = y*x; k = k+ 1;
{ ('old m' - 'old k') - (m - k) > 0}
```

(We assume that the invariant is already established.)

We can calculate

wp("y = y*x; k = k + 1;", ('old m' - 'old k') - (m - k) > 0)
≡ (('old m' - 'old k') - (m - (k+1)) > 0

and indeed this is implied by 'old k' < 'old m', since, using the implicit equalities in the pre-condition that m = 'old m' and k = 'old k', the weakest pre-condition simplifies to

1 > 0 .

In addition, it is easily checked that

m - k ≤ 0 implies not(k < m)

to complete the verification.

9.8 Conditional correctness

If we provide an input of the form

while (B**)** ***//{*** I **}**
  S
//{Q**}**

wp will attempt to establish the claim

{I and pre("**while (**B**)** S", true) } **while (**B**)** S {Q}.

This claim is, however, not quite a proof that the while-loop is correct with respect to the pre-condition I and the goal Q, for without a proof of termination, the additional pre-condition pre("**while** B **do** S", true) is required to supply as a hypothesis that the condition that the loop terminates.

A claim of this form is therefore called a claim of **conditional correctness**; if wp("**while (**B**)** S", true) can be established, then the loop will terminate with Q.

verify C/Java code fragment using pre-condition calculation
context: the code fragment consists of assigment statements, `if`-statements, and `while` loops, followed by a goal assertion as a comment //{ *goal* }, and possibly preceded by an optional precondition. For each `while` statement, an invariant and a variant function must be proposed .
method: use `wp` to compute a pre-condition for the fragment. If the proposed invariants are all verified and the computed precondition is `true`, the fragment is conditionally correct with respect to the goal. If the conditionally correct fragment includes no `while` statement it is unconditionally correct. If it contains `while` statements all of whose variants are verified as well, then the code is unconditionally correct.
background: theory files, simplification.rules, and the Well-Behaved Expression assumption

The 3n+1 problem

A interesting example of a program whose conditional correctness is very easy to establish but for which no proof of termination is known is the so-called 3n+1[21] problem.

```
//{ n > 0 }
   while (n > 1) {
         if (even(n)) n = n div 2;
         else n = 3*n+1;
   }
//{ n = 1 }
```

Notice that in this case, no complicated invariant is need to show conditional correctness. `n > 0` will do as an invariant (why?), and obviously (given that `n` is an integer) we have:

n> 0 and not (n > 1) implies n = 1,

so if the loop terminates, the goal is achieved.

It seems plausible to suppose that no matter what the initial value of `n` is, it will inevitably become equal sooner or later to a power of 2, from which point it descends precipitously to 1. But despite a great deal of both computational and mathematical

[21] Also known as the "hailstone" or Collatz problem.

effort (see, for example, [Lagarius, 1985]), no one has yet succeeded in finding an appropriate variant for the loop with which one can establish that this is indeed true.

9.9 FOR loops

For-loops are a special case of while-loops with the advantage that no proof of termination is needed in order to show unconditional correctness. An equivalent[22] `while` statement for the `for` statement

`for(int` *V* `=` *Initial*`;` *V* `<` *Limit*`;` *V*`++)` *S*

is

`int` *V* `=` *Initial*`; while(`*V* `<` *Limit* `) {`*S* `;` *V*`++;}.`

Applying the rules for calculating a pre-condition for a `while` statement with the goal G yields the condition:

If I and *V* < *Limit* implies wp("*S*; *V*++", I) and I and *Limit* ≤ *V* implies G, then

{I and wp("`for(int` *V* `=` *Initial*`;` *V* `<` *Limit*`;` *V*`++)` *S*", true) }
`for(int` *V* `=` *Initial*`;` *V* `<` *Limit*`;` *V*`++)` *S*
{G}.

But we can simplify this substantially if we assume that *S* contains no `break` statement and does not modify the loop counter *V.* This guarantees that if the loop is entered, it will exit with *V* = *Limit*. So we can drop the hypothesis

wp("`for(int` *V* `=` *Initial*`;` *V* `<` *Limit*`;` *V*`++)` *S*", true);

since it is guaranteed to be true, and we can break I and *Limit* ≤ *V* into two cases:

Limit ≤ *Initial* and I[*Initial* / *V*] implies G, and
Initial < *Limit* and I[*Limit* / *V*] implies G,

since if *Initial* < *Limit* , *V* will eventually = *Limit* .

Finally, the invariant condition simplifies to

I and *V* < *Limit* implies pre("*S*", I[*V* + 1 / *V*]),

and, if all the conditions are satisfied, the pre-condition for the for-statement is just
I[*Initial* / *V*]

This can all be summarized in the verification scenario:

[22]Well, almost equivalent, but not quite, since a `continue` or `break` statement will have different effects in the two loops.

verify C/Java `for`-statement using pre-condition calculation
context: `for(int V = Initial; V < Limit; V++)` `//{I}S` `//{G}`.
method: use wp to compute and verify the proposed invariant and verify that when the loop halts or is skipped, the goal G is achieved. Compute the pre-condition I[*Initial* / *V*] .
Proof-obligations: I and *V* < *Limit* implies pre("*S*", I[*V* + 1 / *V*]), and the "boundary conditions": *Initial* < *Limit* and I[*Limit* / *V*] implies G, *Limit* ≤ *Initial* and I[*Initial* / *V*] implies G. If the proposed invariant is verified and the computed precondition is `true`, the `for`-statement is unconditionally correct with respect to the goal.
background: theory files, simplification.rules, the Well-Behaved Expression assumption, and the assumptions that *S* contains no break statement and does not modify *V* .

Exercise 9.17: Give a detailed justification for each of the three proof-obligations in the above scenario.

To see how this works, let's use wp to verify an example we verified using induction in Chapter 5:

```
% wp
. . .
Version 2.0, July 31, 2001
Loading /cs/dept/course/2000-01/F/3341/arithmetic.simp
Loading /cs/home/fac2/peter/3341/equality.simp
Loading /cs/dept/course/2000-01/F/3341/logic.simp
Loading /cs/dept/course/2000-01/F/3341/array.simp
Loading /cs/home/fac2/peter/3341/simplification.rules

|: p = 1; for(int i = 0; i < n; i++) //{ p = x**i }
|: p = x*p;
|: //{ p = x**n }

p=x**i may not be a precondition for the FOR-loop goal.
Cannot verify p=1 and n<=0 implies x**n=p.

// PRE: true
```

The pre-condition for the entire fragment has been computed as `true`, so if we can discharge all the proof-obligations, the code is verified. Two of the three conditions are verified using the available simplification rules (since there is no message output concerning them), but the third condition cannot be verified unless we assume, as we did in Chapter 5, that `n` is a natural number, so that $n \geq 0$.

Exercise 9.18: Add a pre-condition on `n` to the example above and use `wp` (and whatever simplification rules are needed) to show that all the proof-obligations for the `for` statement are met.

Exercise 9.19: Use the verification scenario above to work out by hand exactly which propositions which `wp` must have proved in Exercise 9.18.