WHAT EVERY SOFTWARE ENGINEER SHOULD KNOW ON AXIOMATIC SEMANTICS

Herwig Egghart, Scientific Games International

April 6, 2016

Abstract:

Among the most valuable aids in reasoning about 3GL code are embedded assertions that are known to be true when reached by program execution. Not only can such assertions facilitate intuitive comprehension, but they are also amenable to precise calculation based on the program statements they surround. Officially known as "axiomatic semantics", this calculus of assertions dates back to the 1960s but is still hardly known among software engineers today. At the same time, however, even experienced professionals routinely fail on small problems like constructing a correct binary search! This paper tackles both issues by teaching a practical style of assertional reasoning and applying it to derive a convincing binary-search solution.

1. BINARY SEARCH – THE PROBLEM

Consider the following little programming task: An array **a** is filled from index **1** through **n** in ascending order. The question is how many of these **n** values are less than a given value **x** (which has the same data type as the elements of **a**). Note that since **a** is sorted, we needn't look at every single element: If an element in the middle is less than **x**, so are all elements before it. If an element in the middle is greater than **x**, so are all elements after it. Thus, our goal is an efficient *binary-search* loop that keeps eliminating half of the unsearched elements until the last element less than **x** has been identified.

Not difficult in principle, this problem has puzzled generations of programmers because getting all the details right is anything but trivial. (Years ago, I used to collect binary searches from workshop participants but eventually stopped after the 17th incorrect solution.) Obviously there are programming tasks for which our "natural" way of thinking is not sufficient – binary search being one striking example. So let's now turn from *operational semantics* ("what the computer does") to *axiomatic semantics* ("what we know at which point"), and the first step in this direction is to express the desired outcome as a simple *assertion*. If **i** is the number of elements less than **x**, then **a(i)** is the last element less than **x** and **a(i+1)** is the first element at least as big as **x**:

```
[…]
{ a (1..i) < x ≤ a (i+1..n) }
```

The placeholder **[…]** stands for the program to be designed, and the predicate within curly brackets is the assertion we want to hold immediately after. Note the possibility that **i** equals **0** or **n**, in which case our assertion contains the empty sub-array **a(1..0)** or **a(n+1..n)**, respectively. But this is not a problem, because *anything* is true for elements that don't actually exist. In the extreme case **n = 0**, the solution is **i = 0** with **a(1..0) < x ≤ a(1..0)**.

What's left is to find suitable program statements and intermediate assertions, such that the specified final assertion is provably established. In order to do so, however, we first need the knowledge how to properly "play the game" of assertional reasoning. This is the topic of the following section.

2. THE RULES OF THE GAME

During the early history of axiomatic semantics, a variety of theoretical styles have been developed. Peter Naur analyzed algorithms using snapshots [1]; Robert Floyd attached assertions to flowcharts [2]; Tony Hoare worked with triples of the form **{precondition} program {postcondition}** [3]; and Edsger W. Dijkstra focused on weakest preconditions for given postconditions [4].

However, from a practical software-engineering viewpoint, it turns out that the formal differences between these theories are immaterial. Instead, all we need is a well-balanced mix of source code and assertions (sometimes called *proof outline* or *annotated program*), where every programming construct is locally in balance with the assertions next to it. So with this general plan in mind, let us now have a look at different constructs and their balancing rules.

2.1. How to jump over assignments

Let **v** be an arbitrary program variable and **τ** a valid programming-language term whose data type is compatible with **v**. Then the assignment **v := τ;** means that **τ** is evaluated in the current run-time environment, and the result is stored as the new value of **v**. While this *operational* meaning is of course basic programming knowledge, it's much less obvious how it translates to an equivalent *assertional* meaning. In other words, if **P** and **Q** are predicates over program variables, how can we know if the following fragment balances?

```
{ P }
v := τ;
{ Q }
```

Although it's clear that the relationship between **P** and **Q** must have to do with variable **v** and term **τ**, the exact rule which the pioneers of axiomatic semantics found out seems hardly intuitive to anyone who sees it for the first time. The trick is that in contrast to program execution, which flows downward from **P** to **Q**, assertions have to be calculated just the other way round, i.e. we start at **Q** and replace **v** by **τ** as we "jump upward" over the assignment:

```
{ [v := τ] Q }
v := τ;
{ Q }
```

Note that the square-bracket notation means substitution, and that the resultant predicate **[v:=τ]Q** is always the least restrictive (or *weakest*) **P** satisfying **{P} v := τ; {Q}**. Since this also happens to be the "best" **P** for practical purposes, we shall not explore alternative **P**s at this point but illustrate the rule just learned with a little example.

2.1.1. Swapping with temporary variable

Suppose we have two assignment-compatible variables **a** and **b** and want to exchange their values. Unlike binary search, this problem is so simple that operational reasoning is perfectly sufficient. All we need to do is save the old value of **a** in a temporary variable, copy **b** to **a**, and then copy the saved value from the temporary variable to **b**:

```
t := a;
a := b;
b := t;
```

As this program consists only of assignments, we should already be able to calculate assertions for it. Let's refer to the original values of **a** and **b** by what's called *rigid variables* **A** and **B** (written in uppercase to distinguish them from program variables). Then the goal of our program can be captured by the final assertion **b=A ∧ a=B** [**∧** meaning "and"], and by repeated calculation from bottom to top, we eventually get a fully annotated program:

```
                                                            { a = A ∧ b = B }
t := a;              t := a;              t := a;              t := a;
                                          { t = A ∧ b = B }    { t = A ∧ b = B }
a := b;              a := b;              a := b;              a := b;
                     { t = A ∧ a = B }    { t = A ∧ a = B }    { t = A ∧ a = B }
b := t;              b := t;              b := t;              b := t;
{ b = A ∧ a = B }    { b = A ∧ a = B }    { b = A ∧ a = B }    { b = A ∧ a = B }
```

Note how each step of this calculation yields a new assertion by performing a correct "upward jump". And if we now look at *precondition* **a=A ∧ b=B** and *postcondition* **b=A ∧ a=B**, we see that the net effect is indeed a variable swap.

2.1.2. From verification to construction

The previous example shows a typical mix of operational and assertional reasoning: First, operational semantics was used to "guess" the correct program, and then axiomatic semantics was used to verify the result. But wouldn't it be interesting to watch the whole program emerge from assertional reasoning alone? In other words, can we construct the annotated program as a single entity, yielding the correctness proof as a by-product of the construction process?

From a purely axiomatic viewpoint, we are looking for a *predicate transformer* that turns postcondition **b=A ∧ a=B** into precondition **a=A ∧ b=B**. However, if we try one of the substitutions **[b := a]** or **[a := b]**, we obtain:

```
{ a = A ∧ a = B }    i.e.  a = A = B          { b = A ∧ b = B }    i.e.  b = A = B
b := a;                                       a := b;
{ b = A ∧ a = B }                             { b = A ∧ a = B }
```

Of course, both fragments have to be read from bottom to top again, and we see that the assertions on top can only be true if **A = B** (meaning that the two variables to be swapped have the same original value). But since we want a program that also works for **A ≠ B**, we must change the right-hand side of the assignment, e.g. to a new variable **t**:

```
{ t = A ∧ a = B }
b := t;
{ b = A ∧ a = B }
```

Now we can safely apply **[a := b]** without running into **A = B** again!

```
{ t = A ∧ b = B }
a := b;
{ t = A ∧ a = B }
```

This looks almost like the precondition already; we just need to apply **[t := a]** to finish our construction.

2.1.3. Swapping without temporary variable

Time for a more advanced exercise: Can we get rid of the temporary variable and still swap **a** and **b** correctly (even if **a ≠ b**)? Obviously not if the right-hand side of the last assignment is just **a** or **b** – as we saw in the previous section. So why not try the composite term **a + b**, assuming e.g. that **a** and **b** are integer variables?

```
{ b = A ∧ a + b = B }
a := a + b;
{ b = A ∧ a = B }
```

Note how closely the new predicate **a + b = B** resembles **b = B**. All we need to do is "subtract" **a** somehow, in the simplest case by one of the substitutions **[a := a – a]** or **[b := b – a]**. The first option boils down to **[a := 0]**, trapping us in the undesirable predicate **b = A = B** again:

```
{ b = A ∧ 0 + b = B }
a := 0;
{ b = A ∧ a + b = B }
```

So maybe subtracting from **b** is more promising?

```
{ b – a = A ∧ a + (b – a) = B }
b := b – a;
{ b = A ∧ a + b = B }
```

Looks indeed much better, especially because **a + (b – a) = b**. And if we introduce a **NULL** statement that does nothing, we can formally continue with:

```
{ b – a = A ∧ b = B }
NULL;
{ b – a = A ∧ a + (b – a) = B }
```

Now we shouldn't touch **b** any more, in order to preserve **b = B**. Hence, we'll have to change **a** to achieve the missing transformation **b – a** → **a**. But what term **τ** shall we assign to **a**? Let's see if we can calculate the answer:

```
{ b – τ = A }
a := τ;
{ b – a = A }
```

So if we want to have **a=A** on top, we only need to solve **b – τ = a** for **τ**:

```
b – τ = a    | + τ
b = a + τ    | – a
b – a = τ
```

In other words, our last missing substitution is **[a := b – a]**:

```
{ a = A ∧ b = B }
NULL;
{ b – (b – a) = A ∧ b = B }
a := b – a;
{ b – a = A ∧ b = B }
```

Thus, the complete program looks as follows:

```
{ a = A ∧ b = B }
NULL;
{ b – (b – a) = A ∧ b = B }
a := b – a;
{ b – a = A ∧ b = B }
NULL;
{ b – a = A ∧ a + (b – a) = B }
b := b – a;
{ b = A ∧ a + b = B }
a := a + b;
{ b = A ∧ a = B }
```

2.2. How to rewrite assertions

For the sake of completeness, let us also try to characterize the **NULL** statement used in the previous construction. Its operational meaning is simply to leave all variables unchanged (i.e. to “do nothing”), and from an assertional viewpoint we can say that a **NULL** statement is in balance with two equivalent assertions around it. Formally this gives rise to a *conditional* balancing rule [with a bar under the condition and **⇔** meaning “equivalent”]:

$$\frac{P \Leftrightarrow Q}{\begin{array}{l}\{ P \}\\ \text{NULL;}\\ \{ Q \}\end{array}}$$

Practically speaking, we can always “rewrite” an assertion to some other equivalent formula and put a **NULL;** in between (or just empty space where permitted by programming-language syntax).

2.2.1. Alternative swapping solutions

You may wonder if we have just been lucky that the composite term **a + b** in 2.1.3. worked out so nicely. What if we had started with integer *subtraction* instead of addition? Well, let's see:

```
$\{ a = A \wedge b = B \}$
$\{ (a + b) - b = A \wedge b = B \}$
a := a + b;
$\{ a - b = A \wedge b = B \}$
$\{ a - b = A \wedge a - (a - b) = B \}$
b := a – b;
$\{ b = A \wedge a - b = B \}$
a := a – b;
$\{ b = A \wedge a = B \}$
```

```
$\{ a = A \wedge b = B \}$
$\{ (a - b) + b = A \wedge b = B \}$
a := a – b;
$\{ a + b = A \wedge b = B \}$
$\{ a + b = A \wedge (a + b) - a = B \}$
b := a + b;
$\{ b = A \wedge b - a = B \}$
a := b – a;
$\{ b = A \wedge a = B \}$
```

Apparently it doesn't matter *how* **a** and **b** are combined in the last statement (as long as the total amount of information is preserved). But once that decision is made, the rest is dictated by the first goal **b = B** and the second goal **a = A**. Also note the four “invisible **NULL** statements” at the points where assertions have been rewritten.

As a final remark, due to symmetry it would also have been possible to change **b** instead of **a** in the last assignment:

```
$\{ a = A \wedge b = B \}$
$\{ a = A \wedge a - (a - b) = B \}$
b := a – b;
$\{ a = A \wedge a - b = B \}$
$\{ (a - b) + b = A \wedge a - b = B \}$
a := a – b;
$\{ a + b = A \wedge a = B \}$
b := a + b;
$\{ b = A \wedge a = B \}$
```

```
$\{ a = A \wedge b = B \}$
$\{ a = A \wedge a + (b - a) = B \}$
b := b – a;
$\{ a = A \wedge a + b = B \}$
$\{ (a + b) - b = A \wedge a + b = B \}$
a := a + b;
$\{ a - b = A \wedge a = B \}$
b := a – b;
$\{ b = A \wedge a = B \}$
```

```
$\{ a = A \wedge b = B \}$
$\{ a = A \wedge (a + b) - a = B \}$
b := a + b;
$\{ a = A \wedge b - a = B \}$
$\{ b - (b - a) = A \wedge b - a = B \}$
a := b – a;
$\{ b - a = A \wedge a = B \}$
b := b – a;
$\{ b = A \wedge a = B \}$
```

2.3. How to split into branches

We shall now turn from assignment sequences to programs whose execution path depends on run-time conditions. Let **β** be a Boolean programming-language term without any side-effect on program variables. Then the **IF** construct

```
IF β THEN
        […]
ELSE
        […]
END IF;
```

operationally means that **β** is evaluated in the current run-time environment, and depending on the outcome either the **THEN** branch is executed (viz. if the evaluation has yielded **TRUE**) – or the **ELSE** branch is executed (if **FALSE**).

Like with assignment and **NULL** statements, we would like to know when the following **IF** fragment balances:

```
{ P }
IF β THEN
        […]
ELSE
        […]
END IF;
{ Q }
```

Clearly, this will depend on balancing conditions within the two branches. If **P** holds before the **IF**, it will still hold after the **THEN** or **ELSE** (thanks to **β** being free of side-effects). In addition, we can assert **β** at the beginning of the **THEN** branch and **¬β** at the beginning of the **ELSE** branch [¬ meaning “not”]. And in order to guarantee **Q** after the **END IF**, we need to guarantee it at the end of either branch:

```
{ P }
IF β THEN
        { P ∧ β }
        […]
        { Q }
ELSE
        { P ∧ ¬β }
        […]
        { Q }
END IF;
{ Q }
```

Note that unlike the balancing rules we have seen before, this new rule is not “self-contained” in that it contains unknown parts indicated by **[…]**. However, these unknown parts are constrained by embedded assertions **{ P ∧ β }**, **{ P ∧ ¬β }**, and **{ Q }** – so if they both balance locally, the rule makes sure that the fragment is in balance as a whole.

2.3.1. Swapping once again

Our next example addresses a potential problem with the last swapping programs based on integer arithmetic. Everything would be fine within the set **Z** of mathematical integers, which is closed under addition and subtraction. However, a real-world variable **v** can only represent a finite subset of **Z** – typically within the limits **–L ≤ v < L**, where **L** is some large power of **2**. So in order to “survive” the first assignment, we need **–L ≤ a ± b < L** in addition:

```
{ a = A ∧ b = B ∧ –L ≤ a+b < L }              { a = A ∧ b = B ∧ –L ≤ a–b < L }
a := a + b;                                   a := a – b;
b := a – b;                                   b := a + b;
a := a – b;                                   a := b – a;
{ b = A ∧ a = B }                             { b = A ∧ a = B }
```

Note that the second and the third assignment are always safe: The postcondition tells us that **b** becomes **A** and **a** becomes **B**, both of which are valid machine integers to begin with. (The formal safety proofs are left as an exercise.)

We see that neither program works in all cases: If **a** and **b** have the *same sign* (plus or minus) and their total absolute value is too big, we can't compute **a + b**. On the other hand, if they have *different signs* and are too far apart, we can't compute **a – b**. But within the same sign we can always *subtract*, and across different signs we can always *add* (even if we put the “sign-less” **0** together with the positive integers to reduce the number of combinations to be analyzed):

```
–L ≤ a < 0                    –L ≤ a < 0              0 ≤ a < L                0 ≤ a < L
–L ≤ b < 0, i.e. 0 < –b ≤ L     0 ≤ b < L             –L ≤ b < 0               0 ≤ b < L, i.e. –L < –b ≤ 0
                 ----------   ----------              ----------                              ----------
        –L+0 < a–b < 0+L      –L+0 ≤ a+b < 0+L        0–L ≤ a+b < L+0                  0–L < a–b < L+0
```

Now we have all the ingredients for a robust integer swapper without temporary variable:

```
{ a = A ∧ b = B }
IF (a < 0) = (b < 0) THEN
        { a = A ∧ b = B ∧ (a < 0) = (b < 0) } i.e. both are negative or both are non-negative
        { a = A ∧ b = B ∧ –L < a–b < L }
        { a = A ∧ b = B ∧ –L ≤ a–b < L }
        a := a – b;
        b := a + b;
        a := b – a;
        { b = A ∧ a = B }
ELSE
        { a = A ∧ b = B ∧ (a < 0) ≠ (b < 0) } i.e. one is negative and the other non-negative
        { a = A ∧ b = B ∧ –L ≤ a+b < L }
        a := a + b;
        b := a – b;
        a := a – b;
        { b = A ∧ a = B }
END IF;
{ b = A ∧ a = B }
```

2.4. How to strengthen and weaken

The attentive reader may have noticed that the (invisible) **NULL** statements in the previous example are *not surrounded by equivalent assertions* – as would have been required by our balancing rule from section 2.2. Although we do know that **$(a < 0) = (b < 0)$** implies **$-L < a-b < L$**, for example, the opposite direction doesn't necessarily hold, as the counter-example **$a = -1 \land b = 0$** illustrates. The same is true for the next **NULL** transformation, where "<" (above) turns into "≤" (below) to achieve a seamless connection to the sub-fragment within the **THEN** branch.

So rather than being equivalent, the upper assertion is always more restrictive (or *stronger*), and the lower assertion is more relaxed (or *weaker*) in this example. Nevertheless, such a **NULL** fragment is still in balance according to theory – and we can generalize the double-headed *equivalence arrow* [⇔] to a single-headed *implication arrow* [⇒]:

$$\frac{P \Rightarrow Q}{\{ P \}\ \text{NULL};\ \{ Q \}}$$

```
{ P }
NULL;
{ Q }
```

2.4.1. The invisible branch

A final warning about the **IF** construct is in order. Many programming languages offer a variation where the **ELSE** branch is omitted, so you might be tempted to believe that the **ELSE** branch in the balancing rule is optional as well:

```
                                    { P }
IF β THEN                           IF β THEN
                                            { P ∧ β }
        […]                                 […]
                                            { Q }
END IF;                             END IF;
                                    { Q }
```

This reasoning is flawed, however, because the operational meaning of above construct is to execute the **THEN** branch if **β** holds and to "do nothing" if **¬β** holds. But "doing nothing" is a **NULL** statement – so the correct rule is:

```
                                    { P }
IF β THEN                           IF β THEN
                                            { P ∧ β }
        […]                                 […]
                                            { Q }
ELSE                                ELSE
                                            { P ∧ ¬β }
        NULL;                               NULL;
                                            { Q }
END IF;                             END IF;
                                    { Q }
```

Of course, the **ELSE** branch is subject to the standard **NULL** rule, in this case requiring **$P \land \neg\beta \Rightarrow Q$**. And this is exactly the pitfall of an "invisible" **ELSE** branch: The **THEN** branch may be perfectly in balance, but if you forget to balance the **ELSE** branch too (which can easily happen if you don't see it), you cannot assert **Q** after the **END IF**.

2.5. How to balance a loop

The last construct we want to study is a loop with a Boolean term **β** evaluated at the very beginning. If **TRUE**, the **LOOP** body is executed, followed by a re-execution of the whole construct. If **FALSE**, execution continues after the **END LOOP**. Thus, if **P** holds initially and is kept *invariant* by the **LOOP** body, we can assert **P∧¬β** at the end:

```
                                        { P }
WHILE β LOOP                            WHILE β LOOP
                                                { P ∧ β }
        […]                                     […]
                                                { P }
END LOOP;                               END LOOP;
                                        { P ∧ ¬β }
```

The **LOOP** body of the **WHILE** construct has some similarity with the **THEN** branch of the **IF** construct. Again, we require **β** to be free of side-effects, so if **P** holds before the **WHILE**, it will still hold inside the loop – together with **β**, because otherwise the **LOOP** body wouldn't have been entered. However, instead of some other postcondition **Q**, the **LOOP** body has to re-establish the same invariant **P** at the end, as a precondition for the ensuing re-execution.

2.5.1. Safety and progress

While programs composed of only assignment, **NULL**, and **IF** constructs are guaranteed to finish in a finite time, the **WHILE** construct just introduced may loop forever if the **LOOP** body doesn't eventually force **β** to **FALSE**. The *safety property* **{ P ∧ β } […] { P }** is clearly not enough, as the drastic example **{ P ∧ β } NULL; { P }** illustrates. Such a **LOOP** body would always satisfy the previous rule (because **P ∧ β ⇒ P**), but there is no hope that **β** will ever change from **TRUE** to **FALSE**.

On the other hand, we must not insist that the **LOOP** body establish **¬β** immediately when first executed, because it's perfectly OK to need two or more iterations – as long as it's a finite number. What we do need, however, is a gradual progress in the "right direction", which in turn requires a non-Boolean term **μ** (called a *metric*), which gets smaller with every execution of the **LOOP** body.

The metric **μ** needn't necessarily be a programming-language term, but the set of all possible values of **μ** must be what's called *well-founded*, meaning that all decreasing chains (**μ > μ' > μ'' > ...**) are finite. The set **N** of natural numbers is well-founded, for example, whereas the set **Z** of integers is not. Only under the assumption of well-foundedness does the *progress property* **{ μ = M } […] { μ < M }** ensure termination – with the rigid variable **M** capturing the old value of **μ** for a given iteration:

```
{ P }
WHILE β LOOP
        { P ∧ β ∧ μ = M }
        […]
        { P   ∧   μ < M }
END LOOP;
{ P ∧ ¬β }
```

3. BINARY SEARCH – THE SOLUTION

We are now ready to pick up the binary-search problem from the beginning of this paper. If we look at the desired postcondition **$a\ (1..i) < x \le a\ (i+1..n)$**, we see that the two interesting indices **i** and **$i+1$** have a difference of **1**. This is of course perfect for a *finished* binary search, but we'll certainly need more flexibility while the search is still in progress. So let's generalize from **$i < i+1$** to **$i < j$**, with a new variable **j** assuming the role of **$i+1$**:

P: $0 \le i < j \le n+1 \wedge a\ (1..i) < x \le a\ (j..n)$

The simplest initialization is obviously with **i** and **j** as far apart as possible, such that both sub-arrays become empty. And if we then enter a loop with **P** as invariant, we only need to compare **$i+1$** with **j** in the **WHILE** condition:

Next, we need a well-founded metric to guide us from the initial state with **i** and **j** far apart to the final state with **i** and **j** close together. A reasonable choice is **$j - i$**, which is always a natural number because **$i < j$**:

```
i := 0;
j := n + 1;
{ $P$ }
WHILE $i+1 \neq j$ LOOP
        { $P \wedge i+1 \neq j$ }
        { $P \wedge i+1 < j$ }                { $j - i = M$ }
        [...]
        { $P$ }                               { $j - i < M$ }
END LOOP;
{ $P \wedge i+1 = j$ }
{ $a\ (1..i) < x \le a\ (i+1..n)$ }
```

What's left is the design of the **LOOP** body, which should bring **i** and **j** closer together under invariance of **P**. The easiest way to achieve this is one of the substitutions **[i := t]** or **[j := t]**, with **t** somewhere in the gap between **i** and **j**:

```
{ $P \wedge i < t < j \wedge a(t) < x$ }
{ $0 \le t < j \le n+1 \wedge a\ (1..t) < x \le a\ (j..n)$ }
i := t;
{ $P$ }
```

```
{ $P \wedge i < t < j \wedge x \le a(t)$ }
{ $0 \le i < t \le n+1 \wedge a\ (1..i) < x \le a\ (t..n)$ }
j := t;
{ $P$ }
```

Note that there *has* to be such a gap (thanks to **$i+1 < j$**), and for efficiency reasons we'll place **t** right in the middle (with arbitrary rounding if the gap has an even length). Also note that **$0 \le i < t < j \le n+1$** implies **$1 \le t \le n$**, so **$a(t)$** is always a valid array element. Hence, the following **LOOP** body has all the necessary safety and progress properties:

```
{ $P \wedge i+1 < j$ }                        { $j - i = M$ }
t := (i + j) / 2;
{ $P \wedge i < t < j$ }                      { $j - i = M$ }
IF a(t) < x THEN
        { $P \wedge i < t < j \wedge a(t) < x$ }        { $j - i = M$ }
        i := t;
        { $P$ }                                   { $j - i < M$ }
ELSE
        { $P \wedge i < t < j \wedge x \le a(t)$ }        { $j - i = M$ }
        j := t;
        { $P$ }                                   { $j - i < M$ }
END IF;
{ $P$ }                                       { $j - i < M$ }
```

4. ENGINEERING AND INTELLECTUAL CONTROL

The hallmark of a mature engineering discipline is full intellectual control over the artifacts produced. Long before new buildings or machines actually exist, their relevant properties can be predicted with high reliability – and no serious engineer would skip some of the necessary calculations because they're “too much work” or seem “too difficult”. Yet, in the realm of software development, that's what happens all the time: If noticed at all, loss of intellectual control is lightly justified by the fact that the code “will be tested anyway” or is “too tricky to get right”.

For a software engineer with axiomatic-semantics knowledge, there are no such excuses. Whenever operational reasoning reaches its limits, elements of assertional reasoning can immediately be added by asking “What do I know at which point?” and embedding the most helpful answers as (semi-)formal assertion comments in the code. Of course, the validity of each assertion must be verified – if necessary by further assertions and detailed balancing. But if properly done, intellectual control is always attainable; even in the case of “tricky” algorithms like binary search.

Acknowledgments:

My interest in axiomatic semantics was first sparked by Prof. Werner Kuich at TU Vienna and later magnified by Prof. Edgar Knapp at Purdue University, who has also reviewed drafts of the present paper. The material covered is closely tied to my long-term workshop series at Scientific Games Vienna, whose management team I thank for making these workshops possible. Credit also goes to the numerous workshop participants, the interaction with whom has sharpened my view of assertional reasoning as a key to intellectual control in software engineering.

References:

[1] P. Naur. *Proof of Algorithms by General Snapshots.*
BIT, **6**(4):310–316, 1966.

[2] R. W. Floyd. *Assigning Meanings to Programs.*
Proceedings of Symposia in Applied Mathematics, **19**:19–32, 1967.

[3] C. A. R. Hoare. *An Axiomatic Basis for Computer Programming.*
Communications of the ACM, **12**(10):576–583, 1969.

[4] E. W. Dijkstra. *A Discipline of Programming.*
Prentice Hall, Englewood Cliffs, 1976.