Invariant based programming: basic approach and teaching experiences

Ralph-Johan Back

To cite this version:

Ralph-Johan Back. Invariant based programming: basic approach and teaching experiences. Formal Aspects of Computing, Springer Verlag, 2008, 21 (3), pp.227-244. <10.1007/s00165-008-0070-y>. <hal-00477903>

HAL Id: hal-00477903

https://hal.archives-ouvertes.fr/hal-00477903

Submitted on 30 Apr 2010

DOI 10.1007/s00165-008-0070-y
BCS © 2008
Formal Aspects of Computing (2009) 21: 227–244

Formal Aspects of Computing

Invariant based programming: basic approach and teaching experiences

Ralph-Johan Back
Abo Akademi University, Turku, Finland. E-mail: backrj@abo.fi

Abstract. Program verification is usually done by adding specifications and invariants to the program and then proving that the verification conditions are all true. This makes program verification an alternative to or a complement to testing. We describe here another approach to program construction, which we refer to as *invariant based programming*, where we start by formulating the specifications and the internal loop invariants for the program, before we write the program code itself. The correctness of the code is then easy to check at the same time as one is constructing it. In this approach, program verification becomes a complement to coding rather than to testing. The purpose is to produce programs and software that are correct by construction. We present a new kind of diagrams, *nested invariant diagrams*, where program specifications and invariants (rather than the control) provide the main organizing structure. Nesting of invariants provide an extension hierarchy that allows us to express the invariants in a very compact manner. We have studied the feasibility of formulating specifications and loop invariants before the code itself has been written in a number of case studies. Our experience is that a systematic use of figures, in combination with a rough idea of the intended behavior of the algorithm, makes it rather straightforward to formulate the invariants needed for the program, to construct the code around these invariants and to check that the resulting program is indeed correct. We describe our experiences from using invariant based programming in practice, both from teaching programmers how to construct programs that they prove correct themselves, and from teaching invariant based programming for CS students in class.

1. Introduction

Program construction traditionally proceeds through a sequence of rather well-established steps: *Requirement analysis* (understand the problem domain and work out a specification of the problem), *design* (work out an overall structure for the program), *implementation* (code the program in the chosen programming language), *testing* (check that the program works as intended), *debugging* (correct the errors that testing has revealed), and *documentation* (provide a report on the program for users, for maintenance and as a basis for later extensions and modifications). Figure 1 illustrates the traditional design-implement-test-debug cycle and the feedback loops in it.

Testing increases our confidence in the correctness of the program, but does not prove its correctness. To establish correctness, we need a precise mathematical specification of what the program is intended to do, and a mathematical proof that the implementation satisfies the specification. This kind of software verification is both difficult and time consuming, and is presently not considered cost effective in practice. Moreover, verification would not replace testing in the above work flow, because most programs are not correct to start with, and need to be debugged before one can even attempt to verify them. This is one reason why modern verification tools

Correspondence and offprint requests to: R.-J. Back, E-mail: backrj@abo.fi

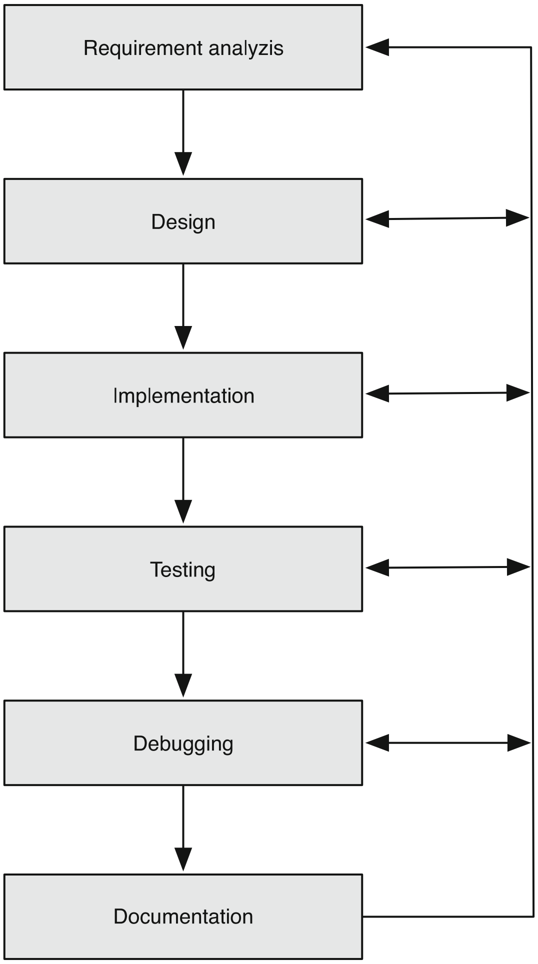

Fig. 1. The programming process

like ESC Java [LN98], JML [BCC+05] and Spec# [BLS04] are focused more on debugging than on complete verification of program correctness.

Verifying a simple program is usually broken down in the following steps:

1. provide the *pre*- and *postconditions* for the program,
2. provide the necessary *loop invariants* for the program,
3. compute the *verification conditions* for the program, and
4. *check* that each verification condition is indeed true.

The difficulty of these steps varies. Formulating pre- and postconditions requires that we have some formal or semi-formal theory of the problem domain. It can also be difficult to identify all the conditions implied by the informal problem statement. The loop invariants must be inferred from the code, and they are often quite difficult to formulate and to make complete enough so that the verification conditions can be proved. Computing verification conditions can be tedious by hand, but this step can be easily automated. Finally, proving that the verification conditions are correct can sometimes be difficult, but for most parts it is quite straightforward. The verification conditions usually consist of a large number of rather trivial lemmas that an *automatic theorem prover* can verify (provided it has a good understanding of the domain theory). The rest of the lemmas are more difficult, and have to be proved by hand or by an *interactive theorem prover* with some help and guidance from the user.

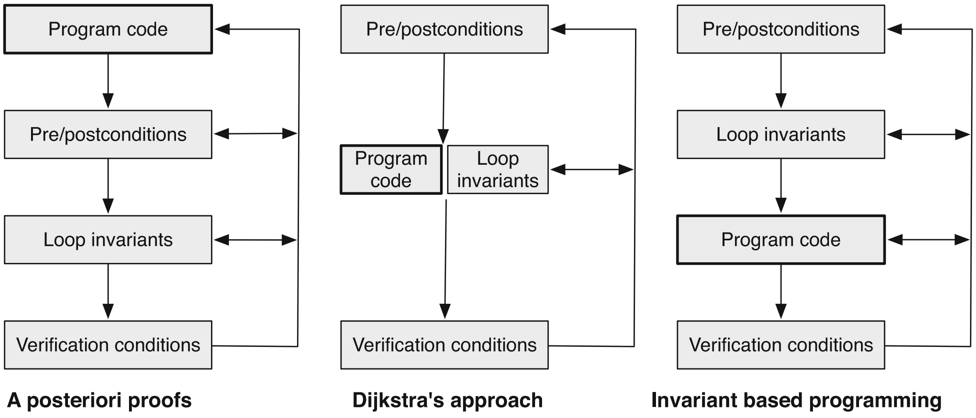

Fig. 2. Three different approaches to verification

The work flow above is known as *a posteriori* verification of software, and is known to be cumbersome. An alternative approach, originally propagated by Dijkstra, is therefore to construct the program and the correctness proof hand in hand. This is known as the *correct by construction* approach to verification [Dij68]. In other words, verification is done in the coding step rather than after (or as part of) the testing step. This means that each subunit of the program is specified before it is coded, and it is checked for correctness immediately after it has been written. Writing pre- and postconditions for the program explicitly, as well as loop invariants, is a considerably help in understanding the program and avoids a large number of errors from being introduced in the first place. Combined with *stepwise refinement* [Dij72], this approach allows the reliable construction of quite large programs.

In this paper, we will propose to move correctness concerns to an even earlier place, to the design phase and immediately following the requirement analysis. This means that not only pre- and postconditions but also loop invariants (and class invariants in object-oriented systems) are written before the code itself is written. We can consider this as a continuation of the design, and as a way of recording design decisions for the algorithm under construction. This requires that the invariants are expressed in the framework and language of the design phase (figures, formulas, texts and diagrams), rather than in the framework and language of the coding phase (a programming language).

We refer to this approach as *invariant based programming*. The idea itself is not new, similar ideas were proposed in the late 70's by John Reynolds, Martin van Emden, and myself, in different forms and variations. Dijkstra's later work on program construction also points in this direction [Dij76], he emphasizes the formulation of a loop invariant as a central step in deriving the program code. Basic for all these approaches is that the loop invariants are formulated before the program code is written. Eric Hehner was working along similar lines, but chose relations rather than predicates as the basic construct. Figure 2 illustrates the differences in work flow between these three approaches.

The idea that program code should be enhanced with program invariants is a repeating theme in software engineering. It becomes particularly pressing to include program invariants when we want to provide mechanized support for program verification, because synthesizing program invariants from the code seems to be too hard to mechanize efficiently. The idea that the program invariants could be formulated before the code itself is written has not, however, received much attention.

Reynolds' original approach [Rey78] was to consider a program as a transition diagram, where the invariants were the nodes and program statements were transitions between the nodes. Invariants become program labels in the code, and the transitions between invariants are implemented with goto's. Reynolds starts by formulating the invariants hand in hand with the transitions between the invariants. The need to formulate invariants early on also highlights the problem of describing the invariants in an intuitive way. For this purpose, Reynolds proposes a visual way of describing properties of arrays. Reynolds' work served as the basis for my own early attempt at invariant based programming, where I studied the method in case studies, and proposed multiple-exit statements as programming construct for invariant based programming [Bac78, Bac80, Bac83]. Van Emden considered programming as just a way of writing down verification conditions [vE79].

Hehner's approach is in spirit very similar to invariant based programming, but comes with different building blocks [Heh79]. He describes imperative programs using tail recursion. This means that he is calling parameterless recursive procedures rather than jumping to labels. Semantically, he is working with input-output specifications (relations) rather than with invariants (predicates). This approach nicely supports stepwise refinement in a non-structured programming language, but the basic paradigm for constructing programs is different from invariant based programming.

I hope to show in this paper that, by a collection of proper enhancements to these early ideas on invariant based programming, it is possible to construct programs that are proved correct with approximately the same amount of work that we today spend on programs that are just tested for correctness. The main issues that we will tackle in the sequel are:

- what is a suitable notation for writing invariant based programs,
- what is a suitable methodology for constructing invariant based programs (in particular, how do find the invariants before the code has been written),
- how do we check the correctness of invariant based programs, and
- what are the specific challenges and opportunities with invariant based programming.

We will then continue with reporting on our experiences from teaching invariant based programming in practice during the last three years. We have carried out a number of sessions where we have taught programmers how to use invariant based programming to solve a standard programming problems and verify that their solution is correct. We have studied how fast they have learned to use the method, what aspects were difficult for them and what aspects were easy. We have also given two courses on invariant based programming. The first course was given when we were still working out the details of the approach, and was given for Ph.D. students in Computer Science with a special background in formal methods. The second course was given for (mostly) first year students in Computer Science at our university, with no a priori experience of formal methods and usually only one or two programming courses behind them.

2. Nested invariant diagrams

The traditional programming approach requires that the program code is kept as simple and intuitive as possible. The programmer has to have a good understanding of the structure of the code and of how the code works, because he understands the code in terms of its possible execution sequences. This is the motivation for the *structured programming* principle [Dij72], which emphasizes single-entry-single-exit control structures in order to keep the code simple and disciplined. This makes it easier to build, understand, extend and modify the code.

When one tries to make the program code as simple as possible, there is a danger that the invariants become more complicated. This is because the invariant has to fit the code, and there is no guarantee that simple code leads to simple invariants. Essentially, we have two possible ways of structuring the program. We can either use control as the primary structure and the invariants as the secondary structure, which has to be adapted to the primary structure. Or, we can use invariants as the primary structure and code as the secondary structure, which has to be adapted to the invariants. We have to choose either one as the primary structure, because the natural structure for code need not be the same as the natural structure for invariants.

When starting from invariants, the program has to be structured around the invariants, so that they are as simple as possible to build, understand, extend and modify. We will look at this issue next. Consider the simple summation program in Fig. 3, described here as a flowchart. The program adds the first n integers, and shows the result in the variable sum.

Figure 4 shows the same program (on the left), but we have now added the pre- and postconditions and the loop invariant. Let us introduce the term *situation* for a condition that describes a collection of states (a condition is a predicate on states; it can be identified with the set of all states that satisfy this predicate). Here, the precondition, the postcondition and the loop invariant are all situations. We notice that a termination function is also needed, to show that the computation does not loop forever.

The program on the right in the same figure is equivalent to the previous one, but now we have deemphasized the program statements by writing them as annotations on the arrows between the situations. We write both the condition for traversing an arrow (the *guard* $[g]$) and the effect of traversing the arrow (an *assignment statement*) on the arrow.

We are looking for a way to impose some structure on the situations described here. A situation describes a set of states. The typical thing that we do with a situation is that we strengthen it by adding new constraints. This means that we identify a subset of the original situation where the new constraints are also satisfied. Thinking of

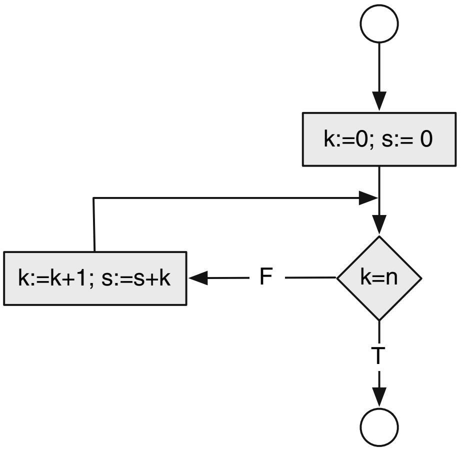

Fig. 3. Simple flowchart

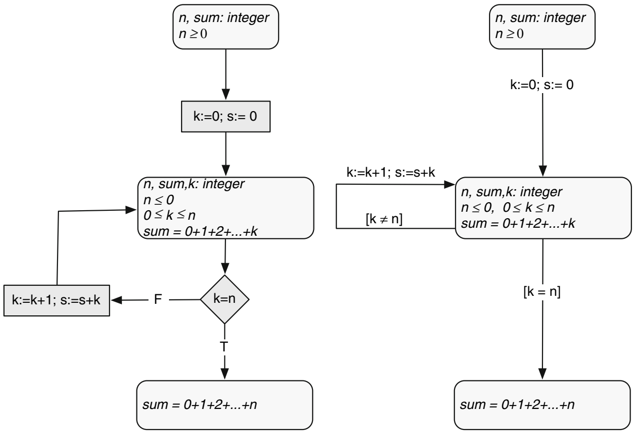

Fig. 4. Flow chart and invariants

situations as sets of states, we can use Venn diagrams to describe this strengthening of situations. Figure 5 shows this nesting of situations for our example program.

We have here omitted conditions that are implicit because of nesting. For instance, $n \geq 0$ is written only once in the outermost situation, but it will hold in all nested situations. The situations are thus expressed more concisely with nesting. The diagram also shows more clearly how the different situations are related. The three situations described in this diagram are as follows:

$$\begin{aligned} \mathit{Outer} &= n, \mathit{sum} \in \mathit{integer} \wedge n \geq 0 \\ \mathit{Middle} &= \mathit{Outer} \wedge k \in \mathit{integer} \wedge 0 \leq k \leq n \wedge \mathit{sum} = 0 + 1 + 2 + \cdots + k \\ \mathit{Inner} &= \mathit{Middle} \wedge k = n \end{aligned}$$

Figure 6 shows the same program as before, but now with nested situations. The transitions are exactly as before. We will refer to diagrams of this kind as *(nested) invariant diagrams* (or *situation diagrams*). This description of the original flow chart is equivalent to the previous ones, the difference is just in the presentation, which emphasizes the structure of situations rather than the structure of code.

We also need a way to indicate the termination function in order to establish correctness. We write the termination function inside the box in the right upper corner of the invariant. The box shows both a termination function $n - k$ and a lower bound 0, in the form of an inequality $0 \leq n - k$. It requires that the termination function must be bounded from below (here $0 \leq n-k$) in the indicated situation, and that the termination function $n - k$ must be decreased before re-entering this situation. The first requirement is established by showing that

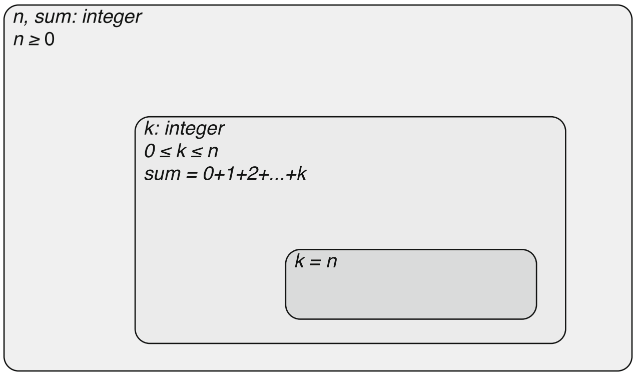

Fig. 5. Nested situations

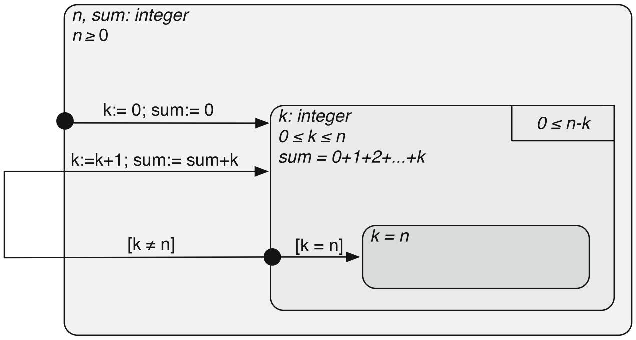

Fig. 6. Nested invariant diagram

$0 \le n - k$ follows from the conditions in the loop invariant, the second is established by analysing the loop transition.

Nested invariant diagrams are similar to *state charts* [Har87, Fow99]. Both are essentially extensions of state transition diagrams. However, the interpretation and intended use is very different. State charts are intended to specify the control flow in reactive systems, without any concern for correctness, whereas invariant diagrams specifically address the correctness issue of algorithms. A state chart is usually seen as describing some specific aspect of a larger software system, i.e., it is a form of abstraction, whereas invariant diagrams describe the whole program.

3. Constructing invariant based programs

We next describe the *work flow* for constructing invariant based programs, with the help of an example. We consider the simplest possible sorting program, *selection sort*. The array is sorted by moving a cursor from left to right in the array. At each stage we find the smallest element to the right of the cursor, and exchange this element with the cursor element. After this, we advance the cursor, until we have traversed the whole array.

Formulating the problem Our first step is to formulate the problem more precisely, and identify the basic concepts that we need for this. We assume that $Sorted(A, i, j)$ means that the array elements are non-decreasing in the (closed) interval $[i, j]$, that $Partitioned(A, i)$ means that every element in array A below index i is smaller than or equal to any element in A at index i or higher, and that $Permutation(A, A_0)$ means that the elements in array A form a permutation of the elements in array A_0.

We first identify the initial situation and the required final situation. Figure 7 shows how we can visualize the programming problem. The shaded region indicates the elements that are already sorted. In the initial situation, no elements are sorted, so there is no shading, in the final situation all elements are sorted, so the whole array is shaded.

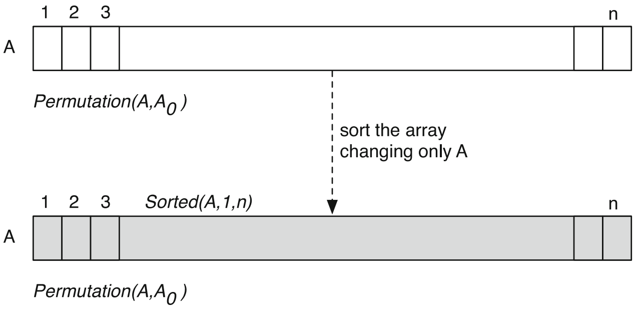

Fig. 7. Visualizing the specification

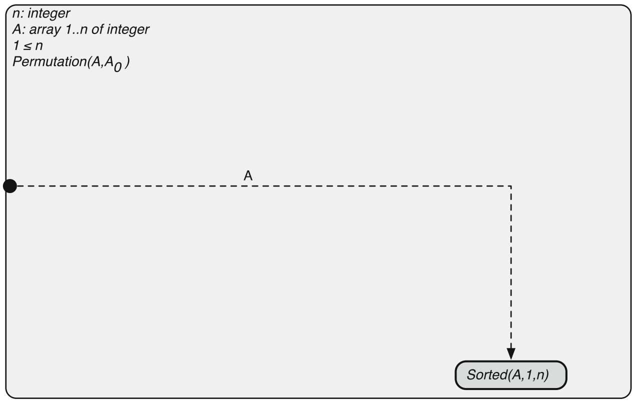

Fig. 8. Initial and final situation

We assume that

$$n \,:\, integer \wedge A : array\ 1..n\ \ of\ integer \wedge n \geq 1 \wedge Permutation(A, A_0)$$

holds initially. The final situation requires that in addition

$$Sorted(A, 1, n)$$

holds. We describe this using the nested invariant diagram in Fig. 8.

We write $n \,:\, integer$ for the constraint $n \in integer$. We assume that we have an enumerable collection of program variables available, and that a program variable may take any values. The constraint $n : integer$ only allows states where the value of n is an integer.

Here the dashed arrow indicates the computation that we want to define. The annotation A on the arrow shows that only program variable A (among the variables introduced so far) may be changed.

Increasing sortedness The next step is to identify an intermediate situation (an invariant). The most plausible one is that part of the array that has already been sorted in the intermediate situation, and that none of the remaining elements are smaller than any of the elements in the sorted part. This is illustrated in Fig. 9, where we describe a new situation between the initial and final situation (the part of the array that has already been sorted is shaded in the figure).

The intermediate situation can be characterized by adding the following conditions to the initial situation

$$i : integer \wedge 1 \leq i \leq n \wedge Sorted(A, 1, i-1) \wedge Partitioned(A, i)$$

This gives us the invariant diagram in Fig. 10.

It is quite easy to check that the initial assignment $i := 1$ will establish this intermediate situation. It is also easy to check that the condition $i = n$ will imply the final situation. Hence, the only thing that remains is to figure out how to make progress towards this condition (decreasing termination function $n - i$) while maintaining the intermediate situation. This is to be done by only changing A and i.

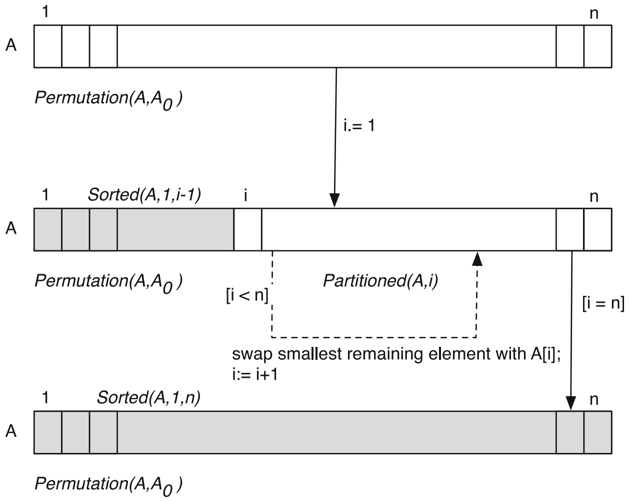

Fig. 9. Sorting with invariant

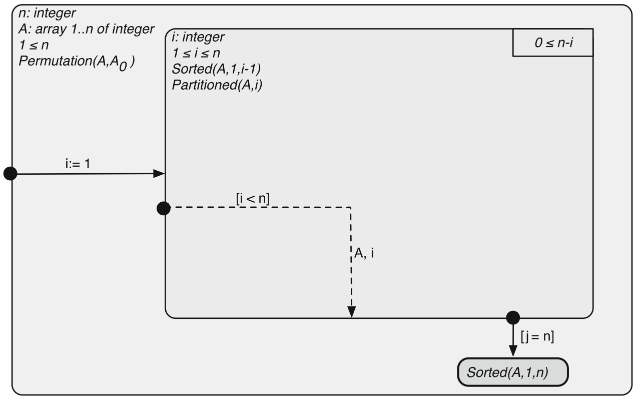

Fig. 10. Intermediate situation

Finding smallest element Finding the smallest remaining element indicates that we need to scan over all the remaining elements, so we obviously need a loop here also. We add a fourth innermost situation, where part of the unsorted elements have already been scanned for the least element. This part of the array is shown with a lighter shading. The new situation is shown in Fig. 11.

The new situation is characterized by the additional constraints

$$k, j : integer \wedge i < n \wedge i \leq k \leq j \leq n \wedge A[k] = min\{A[h] \mid i \leq h \leq j\}$$

We check that this situation is established from the previous intermediate situation by the assignment $j, k := i, i$ when $i < n$. We also check that if $j = n$, then

$$A[i], A[k] := A[k], A[i];\ \ i := i + 1$$

will establish the previous intermediate situation, as indicated in the diagram. The corresponding invariant diagram is shown in Fig. 12.

Final step Finally, we need to make progress while preserving the second invariant. We need to show that when $j < n$, the statement

$$j := j + 1;\ \ \text{if } A[j] < A[k] \text{ then } k := j \text{ fi}$$

preserves the second invariant. This is easily checked.

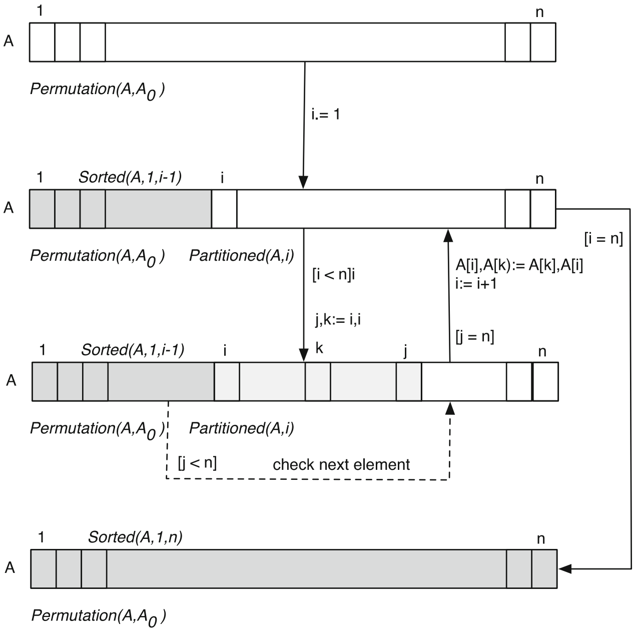

Fig. 11. Sorting program with two invariants

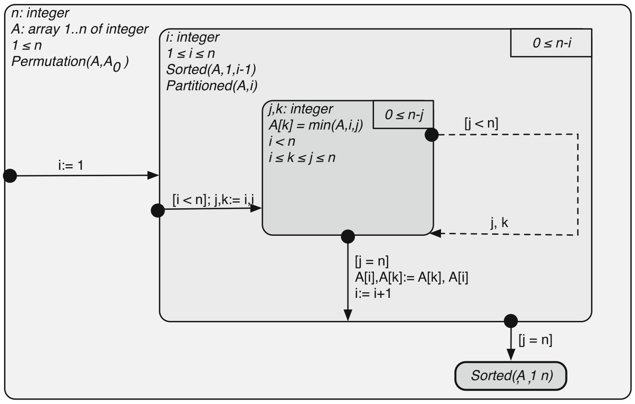

Fig. 12. Second invariant

The inner loop will eventually terminate because $n - j$ is decreased but is bounded from below. The outer loop will terminate because $n - i$ is decreased and is bounded from below. The inner loop does not change the value of the termination function of the outer loop.

This concludes our derivation of the sorting algorithm. Figure 13 shows the invariant diagram for the final program that we have derived.

Executing the invariant based program The invariant based program we have derived can be directly expressed as a flowchart, and it can be compiled to any programming language, either a high level language or directly to assembler, for execution. The latter may be preferable, because no new insight is really gained by looking at the program code in a language like Java or Python. A simple compiler will only look at the transitions in the diagram, and ignore the information given in the situations. A slightly more sophisticated compiler will also take into account the program variable declarations in the invariants, and use this to type check the program. Figure 14 shows the information that is used by such a compiler. The figure also shows how nested diagrams can be useful for describing the looping structure and the variable declaration structure of a program.

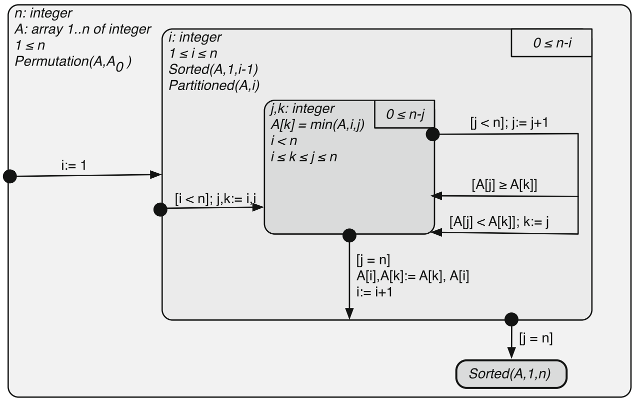

Fig. 13. Invariant diagram for selection sort

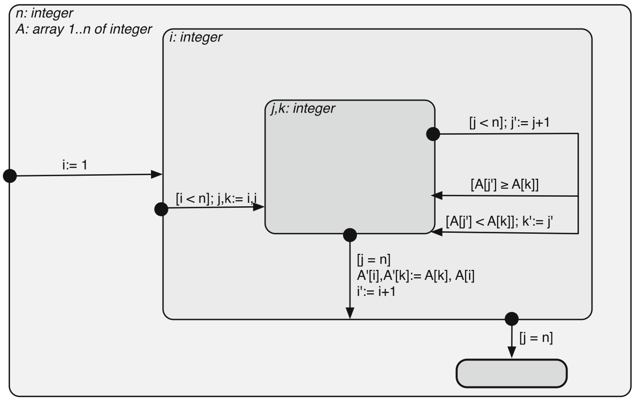

Fig. 14. Control flow and program variable nesting

An even more sophisticated compiler can generate run time checks from the constraints in a situation, either checks for all the constraints in a situation or for a subset of constraints that can be efficiently checked during execution. This will allow us to test whether the situations hold during execution. This can be used as an alternative or complement to verifying the correctness of the invariant based program. This feature is built into the Socos environment described in [BEM07].

Discussion The outermost situation gives the background assumptions for the algorithm. The middle situation is what holds when we have sorted the array up to $i - 1$. The innermost situation holds while we are scanning for the least element. The second nested situation is the final situation. We could also have nested the final situation inside the middle invariant. However, that would have indicated that we also had some information about the value of i at exit. As this is not needed, we prefer to keep this invariant nested only inside the initial situation.

We used figures quite extensively in the derivation of the invariant diagram. The figures allow us to read out the logical formulation of the invariants in a rather straightforward way. Most of the design of the algorithm is done using the figures. The invariant diagrams are, however, more precise and do not have any bias due to badly chosen figures, so they are more suitable for mathematical reasoning about the correctness of the transitions in the diagram.

4. Verifying correctness

The correctness of the sorting algorithm that we have developed depends on the following three basic properties:

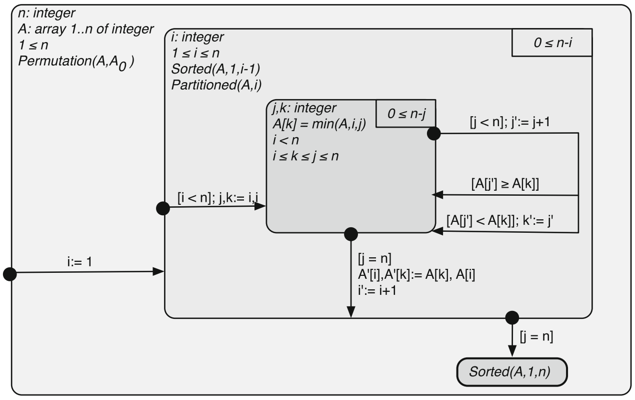

Fig. 15. Single assignment statements

$$
\begin{array}{rcl}
\text{(outer)} & & n \in integer \wedge A \in array\, 1..n\, of\, integer \wedge Permutation(A, A_0) \wedge 1 \leq n \\
\text{(middle)} & \wedge & i \in integer \wedge 1 \leq i \leq n \wedge Sorted(A, 1, i-1) \wedge Partitioned(A, i) \\
\text{(inner)} & \wedge & j, i \in integer \wedge A[k] = min(A, i, j) \wedge i < n \wedge i \leq k \leq j \leq n \\
\text{(trans.)} & \wedge & j = n \wedge A'[i] = A[k] \wedge A'[k] = A[i] \wedge i' = i + 1 \wedge \\
 & \wedge & (\forall h \neq k, i \cdot A'[h] = A[h]) \\
 & \Rightarrow & \\
\text{(outer)} & & n \in integer \wedge A' \in array\, 1..n\, of\, integer \wedge Permutation(A', A_0) \wedge 1 \leq n \\
\text{(middle)} & \wedge & i' \in integer \wedge 1 \leq i' \leq n \wedge Sorted(A', 1, i'-1) \wedge Partitioned(A', i)
\end{array}
$$

Fig. 16. Verification condition

- The diagram is *consistent*, in the sense that each transition will establish its target situation when executed from its starting situation,
- the diagram is *terminating*, in the sense that there are no infinite execution paths, and
- the diagram is *non-blocking*, in the sense that execution will eventually reach a final situation.

Let us consider these issues in more detail. Consistency is the central property that we always require from an invariant based program. It means that all the verification conditions are satisfied. The verification conditions can be constructed in different ways, using backward propagation (weakest preconditions) or forward propagation (strongest postconditions). A quite intuitive way is to take the diagram and turn all assignments into single assignment statements, by introducing new primed versions of program variables whose values are changed. Thus, x stands for the original value of the program variable, x' stands for the value after the first assignment to x, x'' stands for the value after the second assignment to x, and so on. Figure 15 shows the selection sort diagram adapted in this way. Then the diagram shows explicitly all the assumptions that are associated with a specific path from one situation to the other. The conclusion that needs to be proved is that the situation at the end of the path is true for the values that the program variables have upon completion of the transtition.

Consider as an example the transition from the innermost invariant to the middle invariant. The verification condition that we need to prove is shown in Fig. 16.

Note that line 5 in the verification condition is the frame condition for array assignment, i.e., array elements that are not explicitly changed remain the same.

In this case, there is quite a lot to be proved. However, proving each fact at a time is a very efficient way of finding errors, either in the transition or in the situations. The amount of checking that has to be done can be decreased quite drastically by omitting those facts that cannot be wrong. Thus, the transition from the inner situation back to itself in case $A[j] \geq A[k]$ requires us to only prove the following facts (under the appropriate assumptions):

$$j' \in integer \wedge A[k] = min(A, i, j') \wedge k \leq j' \leq n.$$

In principle we should prove that all facts in $outer \wedge middle \wedge inner$ hold after the transition, but these other facts all state properties about program variables that cannot be changed by this transition.

Termination can be checked in a separate stage, or it can be verified at the same time as the consistency of the transitions. We need to show two things for each loop invariant: that the termination function is bounded from below and that the termination function has been decreased when the loop invariant is re-entered, and has not been increased in between. For the middle situation, this means that we have to prove

$$\begin{aligned} \mathit{outer} \wedge \mathit{middle} &\Rightarrow 0 \leq n - i \\ \mathit{outer} \wedge \mathit{middle} \wedge \mathit{inner} \wedge \mathit{transition}_1 &\Rightarrow n - i' < n - i \end{aligned}$$

where $\mathit{transition}_1$ leads from the inner situation to the middle situation. For the inner situation, we have to prove that

$$\begin{aligned} \mathit{outer} \wedge \mathit{middle} \wedge \mathit{inner} &\Rightarrow 0 \leq n - j \\ \mathit{outer} \wedge \mathit{middle} \wedge \mathit{inner} \wedge \mathit{transition}_2 &\Rightarrow n - j' < n - j \end{aligned}$$

where $\mathit{transition}_2$ leads from the inner situation back to the inner situation. In addition, we need to check that the termination function $n - i$ is not increased in any other transition in the diagram, and similarly for the termination function $n - j$.

Finally, we also need to show that the execution does not get blocked at an intermediate situation in the diagram. In our case, we need to show that the final situation $\mathit{Sorted}(A, 1, n)$ is eventually reached. Execution can get stuck at an intermediate situation if there is no enabled transition out of the situation. There are two transitions from the inner situation, one of which is enabled when $j < n$ and the other of which is enabled when $j = n$. Similarly, for the middle situations, there are two transitions, one enabled when $i < n$ and the other enabled when $i = n$. Hence, we need to prove that

$$\begin{aligned} \mathit{outer} \wedge \mathit{middle} \wedge \mathit{inner} &\Rightarrow j < n \vee j = n \\ \mathit{outer} \wedge \mathit{middle} &\Rightarrow i < n \vee i = n \end{aligned}$$

Both of these statements are easily seen to be true.

In fact, we could also have a transition that blocks in the middle of its execution, if we reach a branching point where none of the guards are enabled. In our example, the transition from the inner situation back to the inner situation has a conditional branching point, but the two guards together exhaust all possibilities ($A[i] < A[k] \vee A[i] \geq A[k]$), so execution will always proceed past this point.

The diagram has been designed so that the verification condition can be read directly from the diagram. Correctness can thus be checked by careful inspection of the diagram. In manual proofs, it is easy to add primes to the program variables that are updated in a transition, either by hand in the diagram, or just mentally. In a computer supported environment, the single assignment version of the diagram can be seen as just another, easily computed view of the diagram.

5. Invariant based programs vs ordinary programs

The difference between programming in the usual way and invariant based programming is quite big, and implies a profound change in how one approaches the programming problem. We discuss below some of the most important differences.

Correctness notion There is a difference in philosophy between ordinary programs and invariant based programs, in particular concerning the notion of correctness. The traditional notion of correctness states that if a program is started in an initial state that satisfies the precondition of the program, then the program must terminate in a final state that satisfies the postcondition of the program.

For an invariant based program, this requirement is strengthened, because any state in any situation can be taken as an initial state. In particular, this means that we require that correctness also holds for initial states in interior situations that may not even be reachable by a legal execution from a state in the given initial situation. The correctness of an invariant based program is also stronger in that it is not sufficient that the final situations are satisfied upon termination, all interior situations must also be satisfied whenever execution passes through them.

This means that we can have an invariant based program that is correct in the traditional sense, but not correct as an invariant based program. The program does compute the intended input-output relation, but the transitions do not all preserve the situations. We can then either argue that the program should be considered correct, because it does solve the given problem, or that it should be considered incorrect, because it does not solve the problem in the way the programmer intended the problem to be solved. The programmer's intentions are recorded as the invariants of the program.

The correctness requirement for invariant based programs is in a way stronger than needed. However, as soon as a program has at least one loop, an inductive argument is needed for the correctness proof. If this argument takes the form of a proof with invariant assertions, one will in fact end up proving that the program is correct as an invariant based program. In other words, the usual proof technique establishes a stronger correctness property for a program than what is required by the traditional notion of correctness. This is a consequence of program verification being essentially an induction proof, and it is quite common for the induction hypothesis to be stronger than the lemma that we are proving.

Establishing correctness as a side effect Standard programming methods produce code that needs to be tested and debugged. It encourages a trial and error approach to program construction, and can only achieve higher reliability by spending a considerable amount of effort on testing. Verification can in principle be done a posteriori, but this is usually not done. The problem is considered as already solved (kind of), there is already code that works. The additional verification effort is quite time consuming, and the market situation is anyway such that one can get away with a reasonable number of errors in the code (to be fixed in the next release).

With invariant based programming, we get the invariants for free, as part of the programming process. Checking that the invariants are preserved by the code is done continuously, as part of the program construction process. Writing down the invariants does require an additional effort, but on the other hand, it helps in understanding and solving the programming problem and leads to much higher reliability of the program code.

Role of figures The work flow for constructing invariant based programming is based on using figures as stepping stones to formulating the invariants. Any programmer who is solving a problem like the one above will (or at least should) draw the kind of figures we have shown, to get a feeling for how the program behaves. These figures are preserved as the invariants in invariant based programming. In ordinary programming, they are usually lost in subsequent development steps. By elevating these figures to a more distinguished position in program construction, we are more likely to preserve them and keep them up to date. The role of figures here is similar to the role of figures in physics and engineering: it establishes the central constants and variables that are needed to express the problem and the solution, and shows how they are related to each other.

Smaller grain of modularity Invariant based programming provides a grain of modularity smaller than procedures, the usual smallest units of modularity in imperative programming languages. We can *reason* about invariant based programs in a very local fashion. We can check each situation and its transitions one by one, ignoring other situations for the moment. This locality of reasoning is the payoff of using the stronger notion of correctness for invariant based programs.

We also c*onstruct* an invariant based program in a local fashion, as illustrated in the example above. The situations are introduced one by one, as are the transitions that connect the situations to each other. Similarly, we *change and fix* an invariant based program in a local fashion.

Only termination requires an global view of the program: one needs to check that each possible cycle in the diagram decreases some termination function.

Programming methodology The minimum requirement for an invariant based program is that it is consistent. The program does not have to terminate or be free of deadlocks. There may be deadlocks in the program because we have not (yet) covered all possible cases for some internal situation, i.e., there are cases for which we still need to provide transitions. However, the program constructed thus far is consistent (although incomplete). Similarly, we may have a consistent program, but we have yet to tackle the termination of the program, which may require redefinition of some invariants or transitions.

An invariant based program is constructed by a sequence of successive development steps, where each step preserves the consistency of the previous step, while deadlock freedom and termination may vary during development. We can in fact start from the empty program, with no situations and transitions, then first add a precondition, then formulate a postcondition, and so on. Each step will add, modify or remove some situations or transitions in the diagram.

This approach requires that we carefully check the consistency of each transition when it is introduced. Leaving the consistency checks to a later stage in program development will only accumulate errors in the program and make the consistency checking very laborious. It will also decrease the motivation for carrying out consistency checks at all, because too many interdependent things then need to be considered and changed at the same time. Consistency checks can be done at different levels of rigor, but a good rule of thumb is to use the same rigor as one would use for checking a normal mathematical lemma. Computers can also be very useful in checking that the transitions are consistent.

Tool support There is a clear need for tool support for invariant based programming, in particular to help proving the verification conditions. The advantage of invariant based programming over ordinary programming methods is that the invariants are generated as part of the programing process. Hence, correctness proofs can start directly by generating and checking verification conditions, without the preliminary step of guessing loop and class invariants. Building a tool is also a very efficient way of debugging the method and working out the details involved.

The *Socos system* [BM05, BEM07] provides a graphical diagram editor for nested invariant diagrams. It computes the verification conditions automatically for all transitions, and sends them to either an automatic prover (presently *Simplify* [Nel80]) or to an interactive prover (presently *PVS* [OSR92]). The assertions that could not be verified automatically are shown on demand, as a list of possible semantic errors in the program. The constructed invariant based program can also be compiled in Socos into an existing programming language (presently *Python* [VRD03]) for execution.

6. Teaching invariant based programming

We started experimenting with invariant based programming in 2004, in order to evaluate its usefulness in programming in general, and for teaching formal methods in particular. These experiments have taken two main forms:

- We have studied invariant based programming in depth in sessions with (usually) two programmers working together on a given programming problem. The programmers did not have any prior experience of invariant based programming. We observed the programmers during the session and listened to their conversation as they were working through the programming problem. This allowed us to see what aspects of the method they found difficult and what they thought was easy.
- We have given two courses on invariant based programming. The first course, in spring 2005, was directed to Ph.D. students whose research area was formal methods. We gave this course to get a feeling for how easy it is to apply the method in practice. The second course, in spring 2007, was given as an ordinary introductory Computer Science course at Abo Akademi University.

Programming sessions We have organized 15 programming sessions on invariant based programming. About half of these sessions were with people well-informed about formal methods in general, but with no prior experience of invariant based programming. The rest were novices to formal methods. All but one group had a solid programming background (the one exception was a mathematician with no programming experience). These sessions had a rather fixed format. We chose two programmers that were interested in trying out the approach. We used half an hour to one hour for a tutorial on invariant based programming, with the help of an example. We then gave them a specific programming exercise to solve together. The programming problem was expressed in natural language, in the manner typical for programming courses. The programmers had to specify the problem more precisely (formulate pre- and postconditions), construct the solution program, and verify that their solution was correct. Each sessions lasted 2.5–3 h, including the tutorial. We did not take notes systematically from the sessions, as the series of sessions was not really planned beforehand. However the group arranging and organizing the sessions was quite small (three people) and each member participated in a reasonable number of sessions, so a rather good feeling did eventually emerge for how invariant based programming performed in practice. Our conclusions should still be seen as subjective evaluations of our experiences from these sessions, rather than based on a systematic and carefully quantified experiment.

We used two programmers working together, so that they would have to talk to each other while solving the programming problem. The programmers were using a white board to draw figures and invariant diagrams. In some cases, we also used a beamer together with a simple diagram editor to draw the invariant diagrams. This setup allowed us to observe the programmers while they were working on the solution, and follow their thought processes. We answered specific questions that they had about the method during the sessions. In a few cases we also advised them on the solution, usually to tell them that the approach that they were considering was not going to lead to a very efficient algorithm, and that there were more efficient solutions to the problem. In some cases we also advised them on what kind of logical notation to use for expressing the constraints of a situation. The programming problem that we gave them was not known to them before. We used rather standard programming problems. The easiest (and most often used) was Dijkstra's Dutch National Flag problem (essentially a sorting problem), which was given to programmers that had no prior experience in formal methods. Programmers who knew about formal methods before usually wanted to solve a problem that they were interested in themselves, and which in some cases was quite challenging. We insisted that programmers followed the work flow described earlier. We did not require completely rigorous proofs, but the programmers had to give careful verbal arguments for the correctness of every transition in the program.

Feedback from the sessions The sessions were all successful, in the sense that at the end each team had solved the programming problem using invariant based programming. They had learned the method, specified the programming problem formally, constructed an invariant based program that solved the problem and checked mathematically that the solution was correct, all in the limited time span of 3 h. In general, both those with prior experience of formal methods and those who were building their first verified program did well. They had no problem in understanding the method, and were able to follow the work flow described above with just a little bit of guidance.

Finding the invariant was usually not that difficult, once the rough algorithmic idea for the solution had been grasped. Drawing good figures turned out to be extremely important. Once the programmers had settled on the right way to visualize the problem and the algorithmic solution, it was rather straightforward (but not trivial) to write down the logical constraints that described the corresponding situations in an invariant based program. Programmers who had no prior knowledge of formal methods had some difficulties with the logical notation that was needed to express the situations. They often resorted to an ad hoc mathematical formulation for the invariants.

The most difficult task was almost always formulating the postcondition for the problem. The reason is that one has to identify the appropriate domain concepts and notation in order to write down the postcondition. The domain theory for the application thus has to be worked out first, before one can start describing a program that works on this domain. In most cases, no standard domain theory is available, so one has to construct the necessary concepts and notation on the spot. This showed up in the sessions as the relatively large amount of time that was devoted to working out a proper data representation for the problem, and for defining the appropriate concepts needed to express the postcondition. On average, almost half of the time available for solving the programming problem went to formulating the postcondition.

The situations (preconditions, postconditions, loop invariants) that were proposed usually contained errors and were often too weak. However, once the programmers started to connect the situations with transitions and verify the consistency of these transitions, they became quickly aware of the errors and omissions. The process alternated between writing and rewriting situation constraints, writing and rewriting the transitions, and checking whether the transitions preserved the situations as they should. The overall process turned out to be quite self correcting, it was very efficient in finding errors in the program and converged surprisingly quickly. Some impressively subtle bugs were found during routine verification of transitions (off by one errors in particular, but also termination errors and unintended side effect errors).

Teaching invariant based programming in class The first course on invariant based programming, for Ph.D. students in formal methods, was given to find out how the approach worked in practice, on standard programming problems. We gave extra points for checking the verification conditions mechanically (using PVS). The experience from this course was quite encouraging, the students learned the basic method without problems. Learning to use PVS was the biggest challenge in this course.

We followed up this course with a new course on invariant based programming that was given to Computer Science students in the spring 2007. The course was not compulsory, but still attracted some 16 participants, approximately half of them first year students. Most students had taken a beginners course in logic before. The course essentially covered the material described in the preceding sections, together with quite a lot of practical exercises and assignments in program construction. The students had to work manually for the first two thirds of the course, while the last third of the course was done using the Socos environment. All together there were 32 h of lecturing (including 12 h of exercises in class). There was a final exam after the course, with programming problems that had to be solved using invariant based programming and questions that tested the students understanding of invariant based programs. The students were also given a questionnaire before the course started, one during the course, and one after the course, to find out their experiences from the course and to measure their attitude towards the approach.

Results from the freshman course The results from the course were very encouraging. Most of the students understood the approach well, and were able to construct and verify working programs on their own. Most students taking the course also passed the exam without problems (11 out of 13 taking the exam immediately after the course).

The attitudes revealed in the questionnaires were in general very positive to the course and the material taught in the course. The course was on average found useful, interesting, even fun. Students thought that invariant based programming was easy to learn, invariant diagrams were easy to understand, and they thought that the method could be useful in practice. All but one student considered that they had sufficient mathematical skills to take the course. The students appreciated learning how to prove the correctness of programs mathematically, and considered invariant based programming to be a rather practical way of constructing correct programs. The most difficult aspects of the method were in their opinion proving the verification conditions (work intensive and difficult in general), formulating invariants for complex programs, and the lack of a clear standard for syntax and proofs. The most useful aspects of the course was in their opinion that they learned to think in general terms about how a program works, without a specific programming language. One student said that invariant based programming felt very time consuming and wondered if he had missed something, he felt that there must be a more efficient way for proving that the programs are correct. The full report on the course is available on the web (at our web site crest.cs.abo.fi/imped).

Main lessons learned Our experience from using invariant based programming in practice has highlighted four central issues that one has to face when attempting to write programs that are correct by construction. It has also dispelled one myth, that of how difficult it is to find the invariants of the program.

The first issue was already touched upon earlier, the need to construct a domain theory for the application to be built. There is usually no standard theory or notation for the application domain available. The notations that exist may not be the best for program verification, they have often been designed for proving general theorems about the application domain. The programmers are thus faced with the up front task of building their own theory for the application domain. They usually have little prior training in a task like this, and the result is an ad hoc notation tailored for the programming problem at hand.

If program verification was more common, then building the domain theory could be amortized over many similar programs, constructed and verified over the same domain. This is what happens in the programming field in general: the central data structures and operations needed by applications in a specific domain are identified and codified in a program library that is used over and over again, by different application programs. Measuring how difficult it is to construct a verified program is presently distorted by the large investment cost in building the necessary domain theory, a cost that with more wide spread use of verification could be recaptured in later application projects. We have to some extent managed to avoid this theory building in our sessions and in the lectures that we have given by focusing on well understood application domains (mostly array manipulation programs).

Another central issue is the need for a good background in logic, both for formulating the constraints and for reasoning about the correctness of transitions. Ordinary (first or higher order) logic seems to be a good choice in most cases. The problem is that programmers in general do not have much experience in using logic in practice. This shows up as a difficulty in formulating an informal situation (described by a figure) as a collection of logical constraints, as well as a difficulty in carrying out completely rigorous proofs of transition correctness. This difficulty would most certainly go away with a little bit of training. Programmers with a good background in formal methods did not see this as a problem at all.

The third issue is tool support. Tool support is highly desirable for more routine use of invariant based programming, but is not necessary when learning to use the method. Using a tool like Socos in the very beginning can even be harmful, because the programmers may then revert to a test and debug strategy: they try to guess the constraints needed for the invariant, and then use the automatic theorem prover to check whether the guess was correct. This is a bad strategy, it does not usually converge to a working solution, and it is no substitute for trying to understand the programming problem properly. However, once the programmers had mastered the basic approach, tool support was very much appreciated. It takes the tediousness away from the programming process, by proving most lemmas automatically, leaving only the more challenging ones for the programmer to check. The confidence in the correctness of the final program is also much higher with machine checked proofs.

The fourth issue is the importance of drawing good figures to illustrate the way the algorithm is intended to work. The figures provide the intuition behind the constraints that make up the different situations in the program. Formulating these constraints was often rather straightforward, once the programmers in a sessions had agreed on the figure. However, there was a tendency in many of the sessions to just draw very rough figures (or no figures at all) and go directly to the real issue, i.e., to formulating the precondition, postcondition and the program invariants. In most cases this backfired. Going back and drawing the figures more carefully usually allowed for faster progress again. The figures made it easier to identify the program variables needed for the solution, as well as the auxiliary concepts needed to express the constraints.

The myth that we feel can be (partly) dispelled with is that it is hard to find the program invariants. This is not supported by our experience, neither from the sessions nor from teaching the course. Once the student has a reasonable grasp of how the algorithm is supposed to work, and has made a reasonably good pictorial illustration of the data structures involved, formulating the invariant (before any code has been written) is reasonably straightforward (but never really easy). It is true that it is difficult to get the invariant correct immediately, there are usually some errors and often the invariant is too weak. However, these weaknesses are easily spotted once the programmer starts to connect the invariants with transitions, and a few iterations quickly leads to the right invariant.

Invariant based programming in the CS curriculum Based on the issues that we have identified above, there seem to be at least three different topics that we need to cover when teaching CS students how to write provably correct programs in practice: (a) a working knowledge of logical notation and logical reasoning, (b) how to construct and verify algorithms, and (c) how to analyze and formalize problem domains.

Each of these topics requires a course of its own. These courses should also be very practical in nature. The purpose is to give students a good feeling for how to construct correct programs of increasing difficulty, and to demonstrate that these methods also work in practice. These courses should all be taught at an early stage in the CS education. Leaving the courses to advanced level means that they will not have any influence on the students' programming style and on their appreciation of correctness concerns during programming. We should also resist the temptation to make these courses more difficult than they need to be. The basic notions in logical reasoning, invariant based programming and in defining theories are easy to understand intuitively, and should not be complicated more than necessary.

At Abo Akademi university, we now teach a course on practical logic in the second period of the first year, and a course on invariant based programming in the third period of the first year. A course on specification methods is taught in the second year.

The practical courses outlined above can be followed up (in university level CS education at least) by more theoretical courses, where the the issues involved are studied in greater depth and breadth (mathematical logic, programming theory, specification theory, etc.). These theoretical courses will be much better appreciated if the student already has taken the practical courses suggested above, and has come to understand the usefulness of more theoretical discussions on these same topics.

7. Conclusions and work in progress

We have in this paper argued for an alternative approach to programming, where we start by constructing the preconditions, postconditions and intermediate invariants of the program (the situations) before writing any code. The program code is then constructed in the form of transitions that allow us to move from one situation to another, and checking that the invariants are preserved by these transitions. This allows us to construct a program and its correctness proof at the same time, in a sequence of successive consistency preserving enhancements and modifications to the program. We have argued that the careful use of figures makes it quite straightforward to find the right invariants for the program, once a basic understanding of how the algorithm should work is at hand. We have provided a diagrammatic notation (nested invariant diagrams) that provides an intuitive way of describing invariant based programs, and have shown how to structure a program using nested invariants in this way. The (nested) invariant diagram provides a concise and compact overview of the whole program, and shows the basic structure of the program.

We have also reported on our experience of using invariant based programming in practice, in particular on how to teach invariant based programming to both experienced programmers (in programming sessions) and to novices (teaching in class). We have identified the main issues that have come up during these trials: the need for mastering logic in practice, the problems with formalizing the application domain, the need for good tool support, and the beneficial use of figures as a tool for identifying the situation constraints. Identifying loop invariants does not seem to be such a big issue in this approach. An initial loop invariant is identified quite easily from the figures. Although this initial invariant is usually both incomplete and partly wrong, working out the transitions and trying to verify them will rather quickly improve the invariants to a point where the whole program can be proved correct.

The notations and formalisms for invariant based programming that we have presented here are the basic ones needed for constructing simple programs. We are presently extending the use of invariant based programming to building (possibly recursive) procedures, to data modules and classes, and to the construction of concurrent and distributed systems. We are also evaluating the use of invariant based programming in software projects aimed at producing some real application software. Experiences so far are good, the approach seems to scale up well to both more advanced programming constructs and to larger software systems.

References

[Bac78] Back R-J (1978) Program construction by situation analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki

[Bac80] Back R-J (1980) Exception handling with multi-exit statements. In: Hoffmann HJ (ed) 6th Fachtagung Programmiersprachen und Programmentwicklungen, volume 25 of Informatik Fachberichte. Springer, Darmstadt, pp 71–82

[Bac83] Back R-J (1983) Invariant based programs and their correctness. In: Biermann W, Guiho G, Kodratoff Y (eds) Automatic program construction techniques. MacMillan, New York, pp 223–242

[BCC^{+}05] Burdy L, Cheon Y, Cok DR, Ernst MD, Kiniry JR, Leavens GT, Rustan K, Leino M, Poll E (2005) An overview of jml tools and applications. Softw Tools Technol Transf 7(3)

[BEM07] Back R-J, Eriksson J, Myreen M (2007) Testing and verifying invariant based programs in the socos environment. In: The international conference on tests and proofs (TAP)

[BLS04] Barnett M, Rustan K, Leino M, Schulte W (2004) The spec-sharp programming system: an overview. In: CASSIS 2004 proceedings

[BM05] Back R, Myreen M (2005) Tool support for invariant based programming. In: Proceedings of the 12th Asia-Pacific software engineering conference, Taipei, Taiwan December 2005

[Dij68] Dijkstra EW (1968) A constructive approach to the problem of program correctness. BIT 8:174–186

[Dij72] Dijkstra EW (1972) Notes on structured programming. In: Dahl O-J, Hoare CAR, Dijkstra EW (eds) Structured programming. Academic Press, New York

[Dij76] Dijkstra EW (1976) A discipline of programming. Prentice-Hall, New York

[Fow99] Fowler M (1999) UML distilled. Addison Wesley, Reading

[Har87] Harel D (1987) State charts: a visual formalism for complex systems. Sci Comput Program 8:231–274

[Heh79] Hehner E (1979) Do considered od: a contribution to the programming calculus. Acta Informatica 11:287–304

[LN98] Rustan K, Leino M, Nelson G (1998) An extended static checker for modula-3. In: Proceedings of the 7th international conference on compiler construction, Lecture Notes in Computer Science, vol 1383, pp 302–305

[Nel80] Nelson G (1980) Techniques for program verification. PhD Thesis, Stanford University

[OSR92] Owre S, Shankar N, Rushby J (1992) Pvs: a prototype verification system. In: CADE 11, Saratoga Springs, NY

[Rey78] Reynolds JC (1978) Programming with transition diagrams. In: Gries D (ed) Programming methodology. Springer, Berlin

[vE79] Van Emden MH (1979) Programming with verification conditions. IEEE Trans Softw Eng SE-5

[VRD03] Van Rossum G, Drake FL Jr (2003) The python tutorial—an introduction to python. Network Theory Ltd.,

Received 26 March 2007

Accepted in revised form 6 November 2007 by D. A. Duce, J. Oliveira, P. Boca and R. Boute

Published online 14 February 2008