TUCS

Ralph-Johan Back

Invariant Based Programming Revisited

Turku Centre *for* Computer Science

TUCS Technical Report
No 661, January 2005

Invariant Based Programming Revisited

Ralph-Johan Back

TUCS Technical Report

No 661, January 2005

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 study here an 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. Invariants are described as sets and program code as transitions between the sets. Nesting of invariants provide an extension hierarchy that allows us to express the invariants in a very compact manner. We study the feasibility of formulating specifications and loop invariants before the code itself has been written in a number of case studies. We propose 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 in the program. We discuss the correctness criteria for invariant based programs. Finally, we provide a complementary textual representation of invariant based programs which we refer to as *situation analysis.* This format is is roughly equivalent to nested invariant diagrams, but is better suited for carrying out proofs of the verification conditions .

Keywords: programming methodology, invariant based programming, program verification, state charts,verification conditions, program correctness

TUCS Laboratory
Software Construction Laboratory

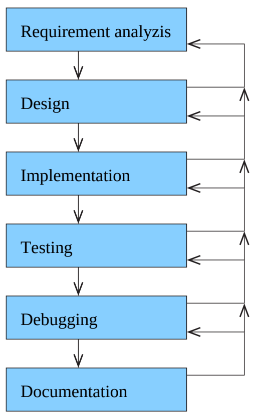


Figure 1: The programming process

1 Introduction

Program construction proceeds through a sequence of rather well-established steps: *Requirement analysis* (understand the problem domain and the specification of the problem), *design* (work out an overall structure for the program), *implementation* (code the program in some 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 that was developed, for users, for maintenance and for later extensions and modifications). Figure 1 illustrates the programming process and the feedback loops in it. This is the traditional design-implement-test-debug cycle.

Testing alone does not establish the correctness of program, as is well known. Correctness requires 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 larger software projects. A verification step would not replace testing, because most programs are not correct to start with, and need to be debugged before one can even attempt to verify them. In the programming process above, verification would be a step after debugging and before documentation.

Let us look at the verification of a simple program in more detail. The steps involved in verification would be:

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. prove that each verification condition is indeed true.

Each of these tasks can be quite difficult. The formulation of 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 is very 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 (maybe 70- 80%) 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 a theorem prover with a little bit of help and guidance (an *interactive theorem prover*).

This *a posteriori* verification of software is known to be cumbersome. An alternative approach propagated by Dijkstra is therefore to construct the program and the correctness proof hand in hand, which he referred to as a *constructive* approach to verification [5]. In other words, the verification is done in the coding step rather than after 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*, this approach allows the reliable construction of quite large programs.

In this paper, we will propose an even earlier place for program verification, immediately following the requirement analysis and in the design phase. 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, as a continuation of the design. This will require 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 will refer to this approach as *invariant based programming*. This is not a new idea, similar ideas have been proposed earlier in the end of the 70s, by a number of authors, like John Reynolds [11], Martin van Emden [12], Eric Hehner [10], and myself [1, 2, 3], in different forms and variations. Dijkstra's other work also points in this direction [7], 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. John Reynolds also considered a visual formalism that makes it easier to describe invariant properties of arrays. 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. The idea that the program invariants should be formulated before the code itself is written has not, however, received much attention.

Reynolds original approach was to consider a program as a transition diagram, where the invariants were the nodes and program statements were transitions between the nodes. Invariants are seen as program labels, and the transitions between invariants are done with gotos. 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,

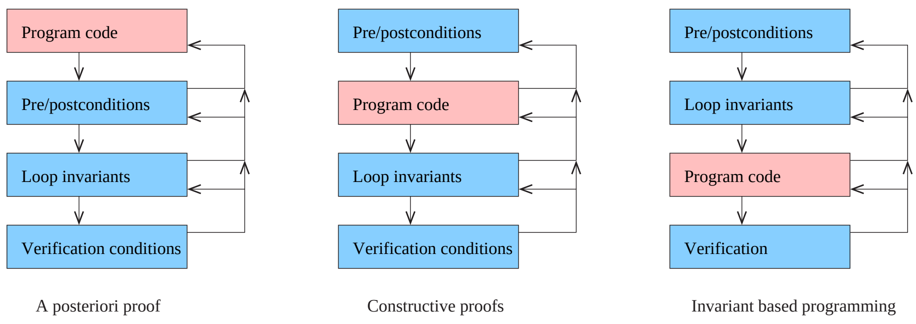


Figure 2: Three different approaches to verification

Reynolds proposes a visual way of describing properties of arrays. Both these ideas served as inspiration for my own early work on this same topic. Van Emden considered programming as just a way of writing verification conditions. His approach is quite similar to the one that we took in describing situation analysis. My own early work was independent of van Emden's work, but heavily influenced by Reynolds' and Dijkstra's work.

Hehner's approach is again very similar to situation analysis, but comes with a different interpretation. 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 of invariant based programming, it is possible to construct correct programs with approximately the same amount of work that we today spend on programs that are just tested. The main new enhancement of these early ideas is the use of nested invariant diagrams as a visual tool for describing the structure of invariant based programs. We also consider the methodological questions here in more detail, in particular how to find and express the invariants before the code has been written.

The rest of the paper is as follows. We start by showing how to construct a very simple program invariants first, in the next section. In Section 3, we show how to describe invariant based programs using *nested invariant diagrams*. These are essentially state transition diagrams where the states denote invariants and the transitions denote executable program statements. The main new idea here is to use nesting to structure the collection of invariants in a program. Section 4 provides a little bit of background theory for the approach and considers the issue of correctness of invariant based programs. In Section 5 we provide a first example of how to construct invariant based programs, and comment on the issues involved in this process. Section 6 gives two other examples of constructing an invariant based program. Section 7 shows how to combine stepwise refinement with invariant based programs using a small example. The purpose here is to avoid too much nesting of invariants by proper identification of subprograms. In Section 8 we provide an alternative textual description of invariant based programs called *situation analysis* (revised from [1, 2] to take nesting into account). We conclude with some general observations and a summary.

2 Constructing a very simple summation program

Let us start by considering a very simple task: sum the integers from 0 to n. We assume that the attributes n and sum are given, and the task is to set $sum = 0+1+\cdots+n$. We need to build an algorithm that achieves this goal, knowing that initially $n \geq 0$ holds. We assume that the only allowed program operations are addition and testing for equality. (If we permitted multiplication and division, then we could solve the problem using the well-known formula $1+2+\cdots+n = n(n+1)/2$).

Constructing the program Before we can formulate an invariant for this program, we need to have a rough idea of how the algorithm is intended to work. Here we will take a very simple approach: we let a new attribute k iteratively take the values $0,1,\ldots,n$, each time adding the new value of k to sum. Initially we set sum to 0.

We think of the invariant as an *intermediate repeated situation* during program execution. This situation can be reached directly from the initial situation, and we can return to it while at the same time making progress, until we have reached our final situation (final goal). In this case the invariant is that sum contains the sum computed thus far, i.e., that

$$sum = 0+1+\cdots+k \wedge 0 \leq k \leq n$$

The initial situation is that

$$n \geq 0$$

and the final situation that we want to establish is that

$$sum = 0+1+0+\cdots+n$$

Once we have decided on the initial and final situation and the invariant, we can start creating the program code. We can achieve the intermediate situation from the initial situation with the statement

$$sum, k := 0, 0$$

This is easily checked:

$$\begin{aligned}
& wp.(sum, k := 0,0).(sum = 0+1+\cdots+k \wedge 0 \leq k \leq n) \\
\equiv\ & 0 = 0+1+\cdots+0 \wedge 0 \leq 0 \leq n \\
\equiv\ & T \wedge 0 \leq n \\
\Leftarrow\ & n \geq 0
\end{aligned}$$

The invariant establishes the final situation when

$$k = n$$

This is also easily checked:

$$sum = 0+1+\cdots+k \wedge 0 \leq k \leq n \wedge k = n$$

$\Rightarrow sum = 0+1+\cdots+n$

Finally, we need to show that we can make progress towards the final goal while preserving the invariant. The way we make progress is to increase k by one (so that we get closer to n) and at the some time adjusting *sum* so that the invariant is still satisfied after this increase. The adjustment needed is to increase *sum* with the new value of k. Thus, the following statement will take us one step closer to n while preserving the invariant:

$$k := k+1; sum := sum+k$$

This alternative is taken if $k \neq n$.

We check that this is indeed the case. First we check that the invariant is preserved by this code:

$$\begin{aligned}
& wp.(k := k+1; sum := sum+k).(sum = 0+1+\cdots+k \wedge 0 \leq k \leq n) \\
\equiv\quad & wp.(k := k+1).(sum+k = 0+1+\cdots+k \wedge 0 \leq k \leq n) \\
\equiv\quad & sum+k+1 = 0+1+\cdots+k+1 \wedge 0 \leq k+1 \leq n \\
\equiv\quad & sum = 0+1+\cdots+k \wedge -1 \leq k \leq n-1 \\
\Leftarrow\quad & sum = 0+1+\cdots+k \wedge 0 \leq k \leq n \wedge k \neq n
\end{aligned}$$

We need to show that some quantity (a *termination function*) is decreased by each iteration, yet cannot decrease for ever. We choose $n-k$ as the termination function. First we check that $n-k$ is bounded from below. In this case, we can show that $0 \leq n-k$ always holds when the invariant holds:

$$\begin{aligned}
& sum = 0+1+\cdots+k \wedge 0 \leq k \leq n \\
\Rightarrow\quad & 0 \leq n-k\ .
\end{aligned}$$

The termination function is indeed decreased when we return to the invariant state:

$$\begin{aligned}
& wp.(k := k+1; sum := sum+k).(n-k < n-k_0) \\
\equiv\quad & n-(k+1) < n-k_0 \\
\equiv\quad & k+1 > k_0 \\
\Leftarrow\quad & sum = 0+1+\cdots+k \wedge 0 \leq k \leq n \wedge k = k_0
\end{aligned}$$

We have now derived the little program in Algorithm 1 for doing the summation:

Algorithm 1 Summation program with gotos

```
$ComputeSum$::  $sum, k := 0, 0$; goto $IncreaseSum$

$IncreaseSum$::  if $n = k$ then goto $Ready$

                     else $k := k+1$; $sum := sum+k$; goto $IncreaseSum$

$Ready$::
```

The summation program expressed with a while loop is shown in Algorithm 2.

Algorithm 2 Summation program with while loop

```
ComputeSum:

    $sum, k := 0, 0$;
    while $n \neq k$ do
        $k := k + 1$;
        $sum := sum + k$
    od
```

Inventing the invariant The crucial step in constructing the program was to identify the invariant. We can describe the invariant graphically, as shown in Figure 3.

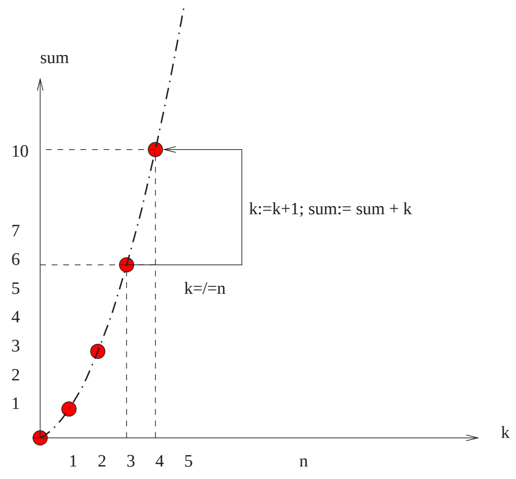


Figure 3: Summing program

The graph shows the location of all integer valued pairs (k, sum) that satisfies the requirement $sum = 1 + 2 + \cdot + k \wedge 0 \leq k \leq n$. We refer to the collection of all states that satisfy this condition as a *situation*. The initial statement $k, s := 0, 0$ puts the state in this situation, because $0 + 1 + \cdots + 0 = 0$. The exit condition $k = n$ in this situation establishes the required result, $s = \sum_{i=0}^{n} i$. Finally, the statement $k := k + 1; s := s + k$ preserve the situation (leads back to the invariant situation) while decreasing the termination function $n - k$. The arrow in the figure shows that if $k \neq n$, then the update $k := k + 1; s := s + k$ will keep the state in the same situation.

The approach to constructing an invariant based program can be summarized as follows:

Analyze requirements: Formulate the initial situation and the final situation precisely.

Formulate intermediate situations: Introduce some intermediate situations, and express these precisely

Provide program code: Show how to move between the situations by program statements.

Check the code: Show that each program statement establishes the destination invariant whenever the source invariant holds.

Termination: Show that some termination function is bounded in an invariant and is decreased before re-entering the invariant.

This leaves us with the question of how identify and formulate the intermediate situations. For this, we must have a rough idea of the way the algorithm is supposed to work. We can gain this understanding by drawing figures that illustrates the basic data structures involved and how they will be changed during execution of the algorithm. We can also hand simulate the execution of the algorithm with concrete data. From the figures and hand simulation, we can then try to identify recurring situations and intermediate goals, and express these generally.

When checking the code, it is a good policy to start by checking the simplest conditions firsts, or the conditions most doubtful to hold, in order to find errors as early as possible. Usually we needs to make adjustments and additions in the situation and/or the program statements before they pass all the tests.

3 Invariants first programming

Invariant based programming is dual to the more traditional approach where one starts by first constructing the program code. With the code in hand, one tries to identify the loop invariants. Once the loop invariants have been identified, then the procedure is the same as above. The difference is thus which comes first, program statements or the invariants.

With the traditional approach, it is very important that the thing that one is constructing, the program code, is kept as simple and intuitive as possible. The programmer has to have a good understanding of how the code works, as this is what he is inventing and modifying until he gets it right. This is the motivation for the *structured programming* principle [6], which emphasize single-entry-single-exit control structures. The intention is to keep the code simple and disciplined, so that it is easy to build, understand, extend and modify.

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. When we are programming invariants first, then we should focus on the structure of the invariants rather than the structure of the code. The purpose is to express the invariants so that they are as simple as possible to build, understand, extend and modify. We will look at this issue next.

Consider again the summation program. Figure 4 describes the program that we constructed above for the summation.

Figure 5 is the same program, but we have emphasized the situations (invariants) by writing them out explicitly .

We see that a termination function is needed here, to show that the computation does not loop forever.

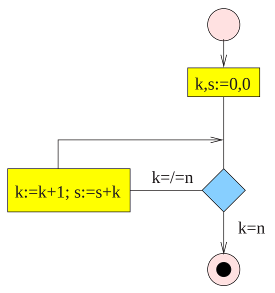


Figure 4: Simple flowchart

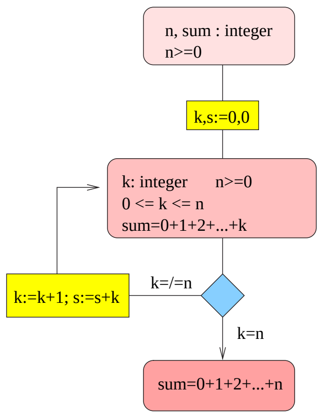


Figure 5: Flow chart and invariants

The third version in Figure 6 is equivalent to the first one, but now we have deemphasized progrma statements further, writing them as just annotations on the arrows between the situations.

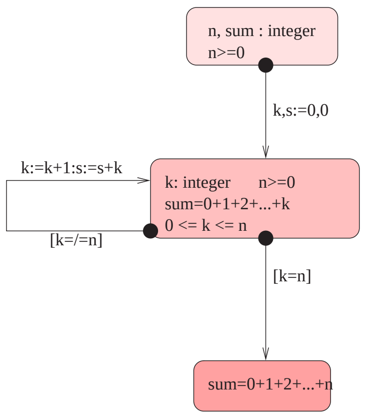


Figure 6: Invariant diagram

We write both the condition for traversing an arrow (the *guard)* and the effect of traversing the arrow (the *update statement*) on the arrow.

We are looking for a way to impose some structure on the situations. Basically, a situation describes a set of states (the set of states that satisfies the condition for the situation). 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 situations as sets of states, we can use Venn diagrams to describe this strengthening of situations. The diagram can then be expressed equivalently as in Figure 7.

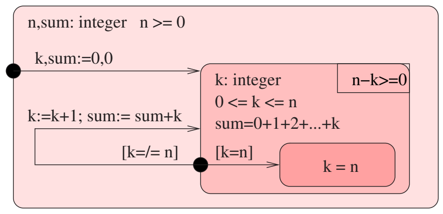


Figure 7: Nested invariant diagram

This is the same diagram as above, except that we have nested the situations to show the strengthening. We have also 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 statement arrows between the situations are the same as before.

The situations are expressed more concisely with nesting. The diagram also shows more clearly how the different situations are related. We will refer to diagrams of this kind as *nested invariant diagrams*.

We also need a way to indicate the termination function. Here we write the termination function inside the box in the right upper corner of the invariant. It shows that $n-k \geq 0$ must hold in the indicated situation, and that the function $n-k$ must be decreased before re-entering this situation.

Note that this description of the flow chart is equivalent to all the previous ones, the difference is just in the presentation. This presentation emphasizes the structure of situations rather than the structure of code.

Nested invariant diagrams are similar to *state charts* [9, 8]. Both are essentially extensions of state transition diagrams. However, the interpretation and intended use is 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. State charts can probably be extended with invariant annotations and would then be quite close to invariant diagrams. The semantics of state charts would then need to be adapted to correctness reasoning. Here we have chosen the simpler approach, and defined the semantics of invariant diagrams directly. The formalism can be extended with features from state charts that are deemed useful (like product states, and signals/procedure calls), but we have here tried to concentrate on the bare essentials.

4 Invariants and attributes

Before proceeding with slighly more demanding examples of how to construct invariant based programs, we need to be more precise about the semantics and correctness of invariant based programs. We base the presentation below on the refinement calculus approach, as described in [4].

Attributes, expressions and assignments We assume that we have an infinite collection of *attributes* (*program variables*). These are essentially observations of properties of an underlying state which we assume cannot be observed directly. Each attribute is uniquely identified by its name (an identifier). The set of all possible states is denoted Σ, while individual states in Σ are denoted by σ, σ_1, etc.

We will assume here that all attributes range over the same set of values V. In other words, our attributes are untyped. The value of attribute x is given by the *value* operation val_x: $val_x.\sigma \in V$ is the value of attribute x in state σ. The attribute x can be set by the *set* operation set_x: $set_x.v.\sigma \in \Sigma$ is the new state that we reach when we set the value of attribute x to value v. The value and set operations satisfy some basic assumptions that are expressed by a collection of axioms (see [4] for details). An example assumption is

$$val_y.(set_x.v.\sigma) = val_y.\sigma$$

when $x \neq y$. This formalizes the assumption that setting a new value for attribute x does not change the value of a different attribute y.

The value of an expression like $x+y+1$ is determined by the value of the attributes in the expression:

$$val.(x+y+1).\sigma \quad \equiv \quad val_x.\sigma + val_y.\sigma + 1$$

The assignment statement maps states into states. It can be defined using value and set operations as

$$(x := e).\sigma \quad \equiv \quad set_x.(val.e).\sigma$$

Situations A *situation* is a boolean formula that may contain free occurrences of attributes. A situation is identified with the set of states that satisfies it. For instance, the situation

$$x \leq y \wedge z \geq 0$$

identifies the set

$$\{\sigma \in \Sigma \,|\, val.x.\sigma \leq val.y.\sigma \wedge val.z. \geq 0\}$$

This situation does not constrain attributes different from x, y, z in any way. The fact that we can consider situations as either constraints or sets allows us to use either the language of logic or the language of set theory to express situations and requirements on situations. For instance, $P \subseteq Q$ and $P \Rightarrow Q$ express the same thing, that any state that satisfies P must also satisfy Q. In the latter formulation, there is an implicit quantification over all states, i.e., $P \Rightarrow Q$ really stands for $(\forall \sigma \in \Sigma \cdot P \Rightarrow Q))$

We treat typing of attributes as constraints on their values. Thus, $x : Nat$ (or $x \in Nat$) denotes the set of states

$$\{\sigma \in \Sigma \,|\, val_x.\sigma \in Nat\}$$

The situations are sets and can hence be described by Venn-diagrams. Nesting of two situations is shown as nesting of the corresponding set outlines. However, the analogy with Venn diagrams is not complete, because we will not associate any meaning to two situations being drawn as disjoint set outlines. In Venn diagrams, this means that they have a nonempty intersection, but we cannot make this interpretation. Consider as an example two situations, $A \equiv x \geq 0$ and $B \equiv y \geq 0$. We draw these two situations as disjoint sets, because they have no constraints in common. However, $A = \{\sigma | val_x.\sigma \geq 0\}$ and $B = \{\sigma | val_y.\sigma \geq 0\}$, so $A \cap B = \{\sigma | val_x.\sigma \geq 0 \wedge val_y.\sigma \geq 0\} \neq \emptyset$. This means that these two situations do in fact overlap, as there are states that belong to both situations.

Transitions A *transition* is a sequence of arrows that start from one situation and end in the same or another situation. Each arrow can be labelled with

- an *assume (*or *guard*) *statement* $[b]$, where b is a boolean condition
- an *assert statement* $\{b\}$, where b is a boolean condition,
- an *assignment statement* $x := e$, where x is an attribute (or a list of attributes) and e is an expression (or a list of expressions of the same length as x).

An assume statement $[b]$ tells us that we may assume that b holds at the indicated place in the transition. An assert statement $\{b\}$ tells us that we have to show that b holds at the indicated place in the transition. If this is not the case, then the transition will fail to reach the target situation. An assignment statement changes the values of the attributes in the usual way.

Consider a transition from P to Q,

$$P : \xrightarrow{S_1} \xrightarrow{S_2} \ldots \xrightarrow{S_n} Q$$

The *statement* S of this transition is the sequential composition of the statements that label the successive arrows, $S = S_1; S_2; \ldots; S_n$. We say that the statement S *goes from* P *to* Q, if S is the statement of a transition from P to Q.

There may be many transitions that start from the same situation P. Transitions with the same initial statements can be combined into a tree-like structure, to simplify the presentation as well as the logic of the transition.

A collection of (possibly nested) situations together with transitions between the situations is referred to as a *nested invariant diagram*. An *initial situation* is one that does not have any actions entering it. A *final situation* is one that does not have any actions leaving it. In general, we may have one or more initial and final situations in a diagram, corresponding to different starting assumptions and different final goals that we have set for our program. [1]

[1]We consider here for simplicity a very simple language for transitions, only using assume, assert and assignment statements. In principle, we can use any refinement calculus statement for a transition. We will consider this generalization in a later paper.

Execution Execution of a statement may either *succeed (miraculously), fail* or *terminate normally.* Executing a statement $[b]$ terminates normally without changing the state σ if $b.\sigma = T$, but *succeeds* otherwise. A statement $\{b\}$ has the same behavior when $b.\sigma = T$, but *fails* otherwise. The assignment statement changes the values of the attributes in the usual way and terminates normally.

A sequential composition of statements $S = S_1; \ldots ; S_n$ will fail for an initial state, if some statement in the sequence fails and all previous statements have terminated normally. Similarly, S will succeed miraculously if some statement in the sequence succeeds miraculously, and all previous statements have terminated normally. If neither of these two conditions hold for an initial state, then S will terminate normally for that initial state.

Consider a situation P in the diagram. Assume that the state σ satisfies P. Execution will *fail at P* if there is at least one transition from P that fails. Execution will *terminate at P* if all transitions from P succeed. Otherwise, there must be at least one transition from P to some Q that terminates normally. We then say that execution from *P proceeds normally* to Q. We say that execution *proceeds normally from P* if execution does not fail at P and there is a transition that proceeds normally from P to some Q.

A diagram is is executed by starting from some situation in some specific initial state σ_0. If execution fails at the situation, then we are done, and the whole execution fails. If execution succeeds at the situation, then we are also done, and the whole execution terminates. Otherwise, we pick nondeterministically one of the transitions that terminates normally from the situation, and execute the statement of that transition. Then we repeat the execution with the target situation and with the state with which we reached the target situation, and so on. In this way, execution will either continue forever, or it will eventually terminate at some situation, or it will eventually fail at some situation.

We do not assume that execution has to start from an initial situation, it is possible to start from any situation in the diagram. Also, execution does not have to terminate at a final situation, termination can occur at any situation. However, execution must terminate if it reaches a final situation (there are no outgoing transitions from a final situation, so all transitions from this situation succeed).

Correctness The *weakest precondition* of a statement S to establish a condition Q is denoted $wp.S.Q$. We define inductively

$$\begin{array}{rcl}
wp.[b].Q & = & b \Rightarrow Q \\
wp.\{b\}.Q & = & b \wedge Q \\
wp.(x := e).Q & = & Q[x := e] \\
wp.(S_1;S_2).Q & = & wp.S_1.(S_2.Q)
\end{array}$$

Here we consider the situations as logical constraints. In set theoretic notation, we would write the weakest precondition for the assume statement as $\neg b \cup Q$ and the weakest precondition for the assert statement as $b \cap Q$.

The invariant diagram is *consistent* if

$$P \Rightarrow wp.S.Q$$

hold for any situations P and Q in the diagram and for any transition from P to Q , where S is the statement of this transition.

We say that an execution is *legal* when its initial state satisfies the situation in which the execution is started.

Consistency guarantees that any situation reached during a legal execution will be satisfied by the state of the execution when that situation is reached. Consistency also guarantees that a legal execution cannot fail. However, consistency does allow an execution to never terminate.

The invariant diagram *terminates* if there are no legal infinite execution in the diagram. We prove this by showing that for each cycle of transitions in the diagram, at least one situation in the cycle has a *termination function*. The integer expression t is a termination function for situation P, if

- $P \Rightarrow t \geq 0$, i.e., the value of t is bounded from below in any state that satisfies P,
- the value of t is never increased in the diagram, and
- the value of t has been decreased each time P is re-entered in the execution.

This shows that execution of every loop in the diagram must eventually terminate, and hence that there are no infinite legal executions.

Consistency and termination together shows that there are no failures during execution, and that every loop terminates. However, this does not exclude the possibility that execution will terminate at an interior situation. This happens if all transitions that lead out of that situation terminate successfully. Hence, any situation is a potential final state. If we want to exclude this possibility, then we need to additionally prove that the diagram is *live*, in the sense that each non-final situation has at least one transition that terminates normally.

Consistency, termination and liveness together implies that any legal execution of the diagram will eventually terminate in a final situation with a final state that satisfies the final situation.

We way that an invariant based program is *correct* if it is consistent, terminating and live.

Invariant based programs vs ordinary programs There is a difference in philosophy between ordinary programs and invariant based programs, in particular concerning the notion of correctness. The correctness notion for invariant based programs is stronger than the traditional notion. 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 situation can be an initial state. In particular, this means that we require that correctness also holds for states in interior situations that cannot even be reached by a legal execution from an 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, but all interior situations must also be satisfied during execution.

One could argue that the correctness requirement for invariant based programs is too strong, that the traditional program correctness notion should be enough. 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 establishing the correctness of the program 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.

The advantage of invariant based programs is that one can reason about them in a local fashion. One only needs to consider each situation at a time, together with the situations that can be reached from this situation with simple transitions. Only the decrease of termination functions requires an overall view of the program, because one needs to check that each possible cycle in the diagram decreases some termination function. This also means that one can change and fix the invariant based program in a local fashion. This locality of reasoning is the payoff of using the stronger notion of correctness that we propose here for invariant based programs (and which we get anyway with traditional proof techniques).

Programming methodology The execution model of an invariant based program is quite general, as we allow for failure, (miraculous) success, infinite execution and normal termination. The minimum requirement for an invariant based program is that it is consistent. The program does not have to terminatie or be live. Liveness may not hold 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 some redefinition of some invariants.

We see program construction as a sequence of successive refinements, where each refinement preserves the consistency of the previous version. A refinement can involve adding some new situations (initial situation, intermediate situations or final situations), or it can involve adding some new transitions to a situation, to increase liveness of the program. A refinement may also modify or remove some transitions or situations, as long as the consistency of all transitions is preserved. A refinement may also change the program so that termination is established for some cycle in the diagram. Consistency is thus preserved throughout program construction, while termination and liveness are properties that may be increased or decreased in successive versions of the program, depending on what are the requirements for the final program.

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. Consistency checks can be done at different levels of rigour, but a good rule of thumb is to use the same rigour as one would use for checking a normal mathematical lemma.

5 A simple sorting program

As our first more demanding example, we consider the simplest possible sorting program, insertion sort. Essentially, we sort the array 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.

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 or equal to any element in A at index i or higher, and that $Permutation(A,A0)$ means that the elements in array A form a permutation of the elements in array $A0$.

Let us first identify the initial situation and the required final situation. These are shown in Figure 8.

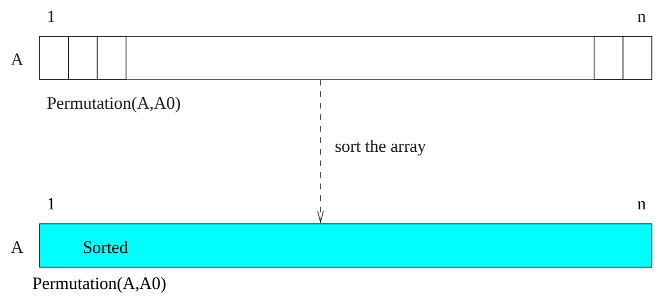


Figure 8: Sorting algorithm specification

We thus assume that

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

holds initially. The final situation requires that in addition

$$Sorted(A, A_0)$$

holds.

The next step is to identify an intermediate situation. The most plausible one is that in the intermediate situation, part of the array has been already sorted, and that none of the remaining elements are smaller than any of the elements in the sorted part. This is illustrated in Figure 9.

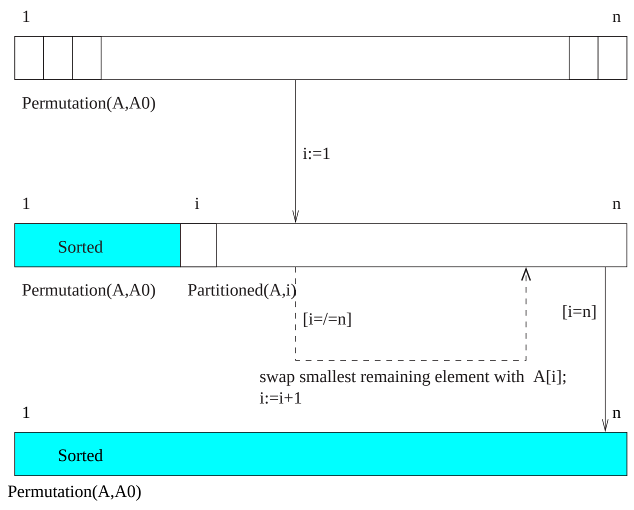


Figure 9: Sorting with invariant

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

$$i : int \land 1 \leq i \leq n \land Sorted(A,1,i-1) \land Partitioned(A,i)$$

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 while maintaining the intermediate situation.

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 situation, where part of the unsorted elements have already been scanned for the least element. The new situation is shown in Figure 10.

The new situation is characterized by the additional constraints

$$k, j : int \land i \leq k \leq j \leq n \land A[k] = min\{A[h] | i \leq h \leq j\}$$

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

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

will establish the first intermediate situation, as indicated in the diagram.

Finally, we need to check that the second invariant is preserved while making progress. We need to show that when $j \neq n$, the statement

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

preserves the second invariant. This is also easily checked. 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.

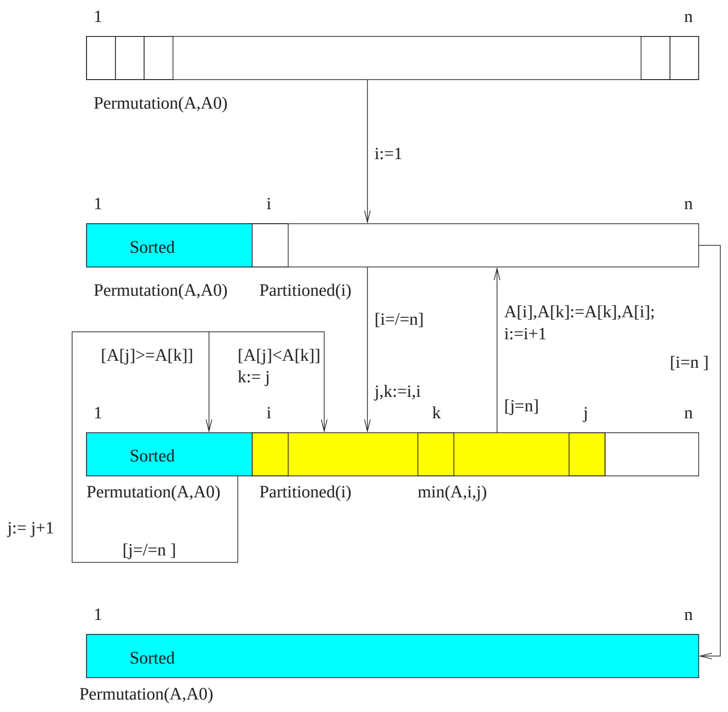


Figure 10: Sorting program with two invariants

This concludes our derivation of the sorting algorithm. Figure 11 shows the invariant diagram for the final program that we have derived.

The outermost situation gives the background assumptions for the algorithm. The nested situation is what holds when we have sorted the array up to $i-1$, but have not yet started to scan for the smallest element in the rest of the array. 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 first 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 just nested inside the initial situation.

To illustrate the difference between traditional programming and invariant based programming, we show the flow diagram that corresponds to the above invariant diagram in Figure 12. We can see from this how the situations /invariants become invisible when we concentrate on the statements that need to be executed and the control flow between the statements.

Figure 13 provides a slightly different description of the same solution, with five different situations rather than four as above. The most complicated transition, when i is updated, is shown here using its own situation. We give this solution to illustrate the point that situations are not only useful when constructing loops, but are generally useful when taking stock of the different cases that can arise during the execution of an algorithm.

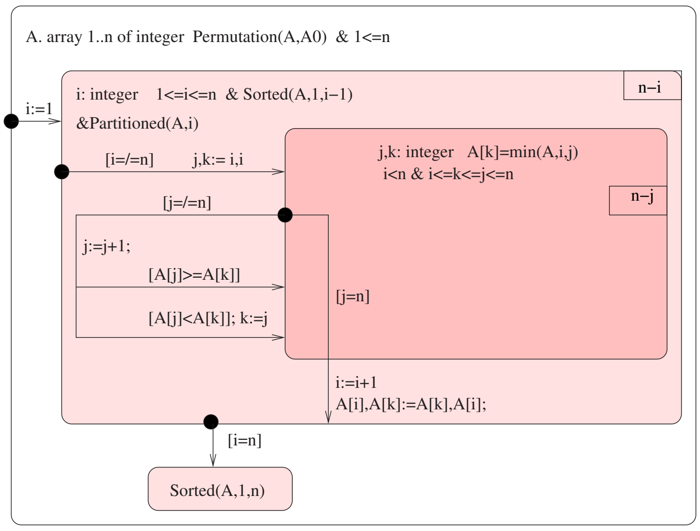


Figure 11: Invariant diagram for sorting program

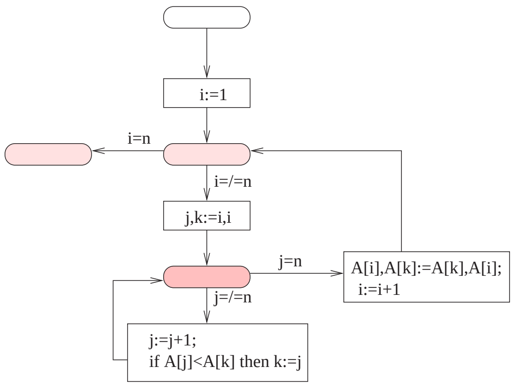


Figure 12: Flow chart for sorting

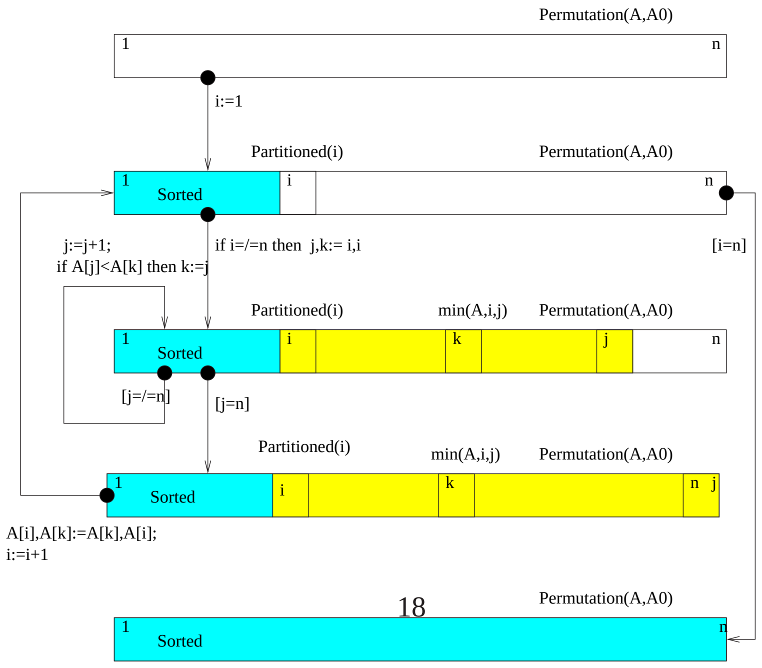


Figure 13: Alternative invariants for sorting program

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 suitable when one is checking that the invariants are preserved by the transitions, and are also more compact because of the nesting.

Any programmer who is solving this problem will (or at least should) draw the kind of figures shown above, to get a feeling for how the program should behave. In invariant based programming, these figures are preserved as the invariants. In ordinary programming, they are usually lost in subsequent steps (some figures may be included in the final documentation, but usually become outdated by later changes in code or during maintenance). 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. For instance, if we want to change an invariant, we would probably first want to check the corresponding figure, to see how the change affects the overall situation, before we update the invariant. Only after the invariant has been updated would we change the transitions between the invariants, and check that the invariants are still preserved by the modified transitions.

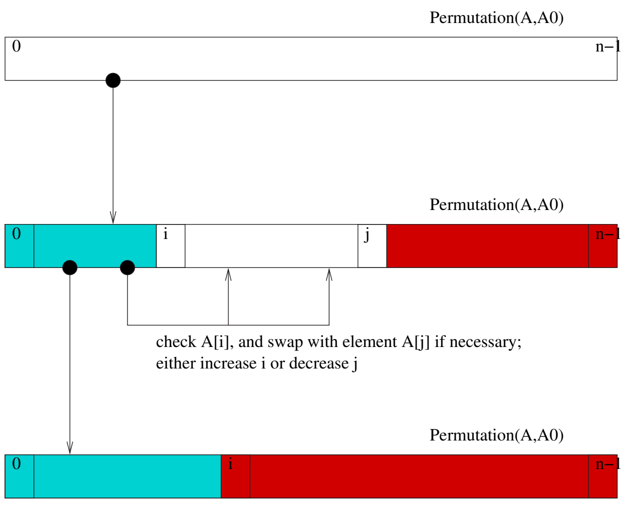
6 Arranging elements in an array

We consider next two different versions of the same problem: arranging elements in an array. The first problem is to partition an array into two parts, the second one considers how to partition the array into three parts.

Partitioning an array with a value We consider the following problem. We are given an array of n integers, and an integer x. The problem is to rearrange the elements in the array so that all the elements less than or equal to x come first.

The solution idea is to keep two indexes in the array, i and j, such that all elements below i are less than or equal to x, and all elements above j are greater than x. One inspects $A[i]$. If this element is smaller than or equal to x, then one can just increase i. If the element is greater than x, then one can in stead swap it with the element at j , and then decrease j instead. The problem here is to get all the index manipulations correct, so that termination happens at the right moment, and one avoids off-by-one errors.

The solution idea, expressed in terms of the domain structures, is described in Figure 14. Here blue elements are smaller than or equal to x, and red elements are larger than x. The invariant diagram in Figure 15 describes the exact solution.



Permutation(A,A0)
0
n−1
Permutation(A,A0)
0
i
j
n−1
check A[i], and swap with element A[j] if necessary;
either increase i or decrease j
Permutation(A,A0)
0
i
n−1

Figure 14: Swapping program invariants

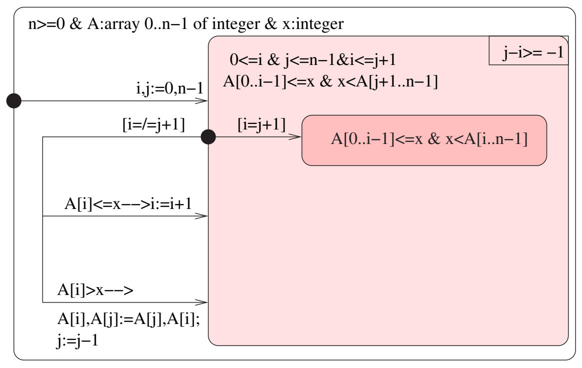


Figure 15: Invariant diagram for swapping program

Note that we have to prove both that each transition leads to the indicated situation and that the termination function is decreased whenever we return to the situation. In this case, the termination function is $j-i$. We also need to check that the termination function is bounded from below. This follows from the invariant here, because $i \leq j+1 \Rightarrow j-i \geq -1$.

Dutch national flag This problem is a variant of the previous problem. Arrange balls in an array so that the blue balls are first, then the white and last the read balls. The solution is similar to the previous one, except that we need to keep track of three indexes in stead of just two. Figure 16 shows the invariants involved. The corresponding invariant diagram is shown in Figure 17.

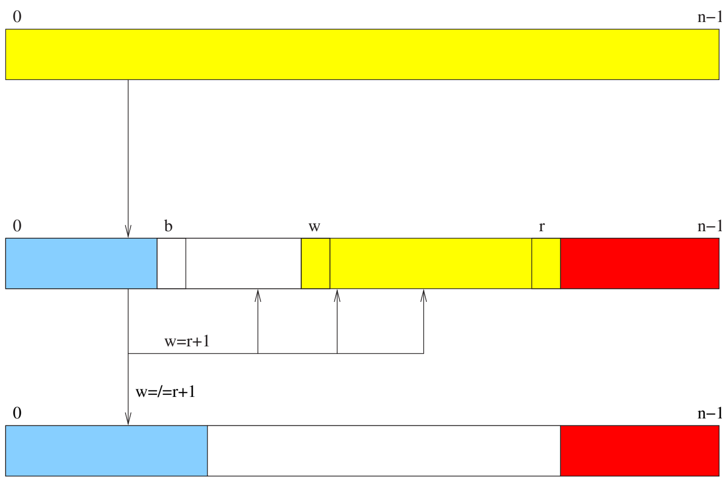


Figure 16: Dutch national flag

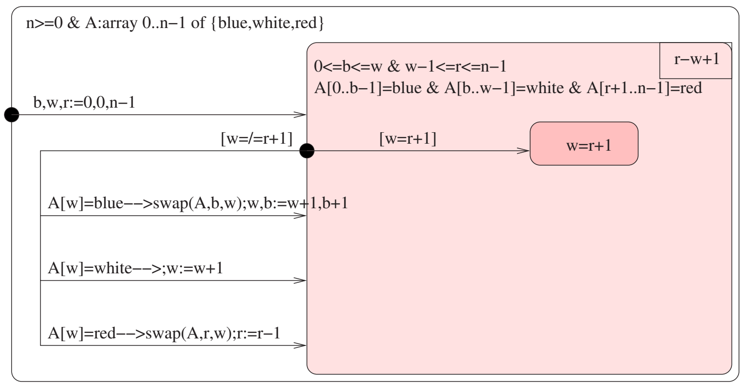

n>=0 & A:array 0..n−1 of {blue,white,red}
0<=b<=w & w−1<=r<=n−1
A[0..b−1]=blue & A[b..w−1]=white & A[r+1..n−1]=red
r−w+1
b,w,r:=0,0,n−1
[w=/=r+1]
[w=r+1]
w=r+1
A[w]=blue−−>swap(A,b,w);w,b:=w+1,b+1
A[w]=white−−>;w:=w+1
A[w]=red−−>swap(A,r,w);r:=r−1

Figure 17: Diagram for Dutch national flag

7 Stepwise refinement of an algorithm

The previous problems have all been solved by finding the invariants first, and then connecting these with statements. When the programming problem gets more complicated, then this is not sufficient. We need to introduce subprograms (procedures) in order to decompose the problem into smaller, more manageable parts. This is usually known as *stepwise refinement* of an algorithm.

We show how to derive a simple array program that finds a specific element in the array. The array is assumed to be sorted row-wise, and the element is assumed to be in the array. We will here use a very simple (and inefficient) strategy: we scan the rows first, to find the row where the array is, and then we scan this row for the element we are looking for.

Solution outline The intended structure of the program is shown in Figure 18. The essential idea is here that we decompose the invariant diagram into two sub-diagrams, *FindRow* and *FindCol*. We show that the problem can be solved with the help of these.

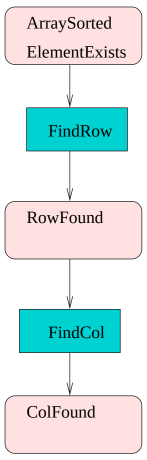


Figure 18: Outline for sorting program

We then construct these two sub-diagrams in separate steps. The diagram that we are looking for is described in Figure 19.

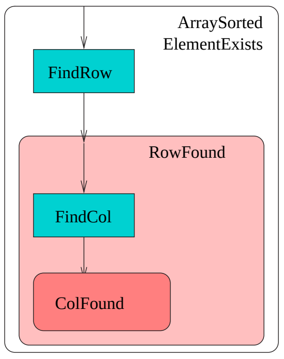


Figure 19: Sorting program invariant diagram

In situation *RowFound* we have found the right row. In situation *ColFound* we have also found the right column.

First, we need to define what we may assume at the start.

- $RowSorted(i) \equiv (\forall j \in [1,n) \cdot a[i,j] \leq a[i,j+1])$
- $ArraySorted \equiv (\forall i \in [1,m] \cdot RowSorted(i)) \wedge (\forall i \in [1,m) \cdot a[i,n] \leq a[i+1,1])$
- $ElementExists \equiv (\exists i \in [1,m], j \in [1,n] \cdot a[i,j] = x)$

Then we need to define what we want to achieve:

- $RowFound \equiv 1 \leq i \leq m \wedge a[i,1] \leq x \wedge (i = m \vee x < a[i+1,1])$
- $ColFound \equiv RowFound \wedge 1 \leq j \leq n \wedge a[i,j] \leq x \wedge (j = n \vee x < a[i,j+1])$

Here *ColFound* is the goal that was to be achieved by the program.

Implementing *FindCol* The diagram in Figure 20 is an implementation of *FindCol*.

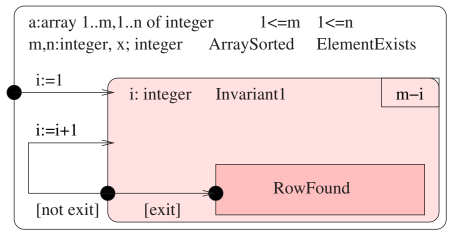


Figure 20: FindCol diagram

Here we define

$$Invariant1 \equiv 1 \leq i \leq m \wedge a[i,1] \leq x$$

and

$$exit \equiv i = m \vee x < a[i+1,1]$$

It is quite straightforward to check that all the required conditions are satisfied, and that this implementation of *FindCol* really does lead us from the initial condition to the *RowFound* situation.

FindCol Next, we implement *FindCol*, in the way shown in Figure 21.

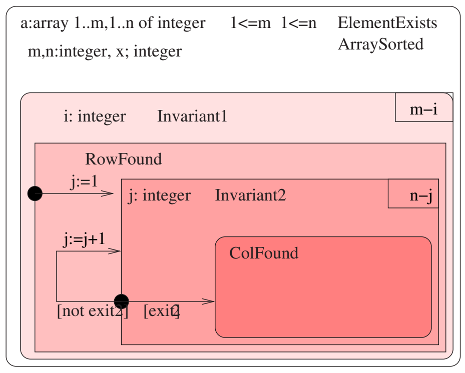


Figure 21: Implementation of ColFound

Note that we have to retain the outer situation descriptions, because we need to know what to assume inside the new situations. The definitions used in this diagram are

$$Invariant2 \equiv RowFound \wedge 1 \leq j \leq n \wedge a[i,j] \leq x$$

and

$$exit2 \equiv (j = n \vee x < a[i,j+1])$$

This shows that the following program solves the given problem:

$i := 1;$

do $i \neq m \wedge x \geq a[i+1,1] \rightarrow i := i+1$od;

$j := 1$;

do $j \neq n \wedge x \geq a[i, j+1] \rightarrow i := i+1$od

The whole program, expressed in terms of invariants first, is shown in Figure 22.

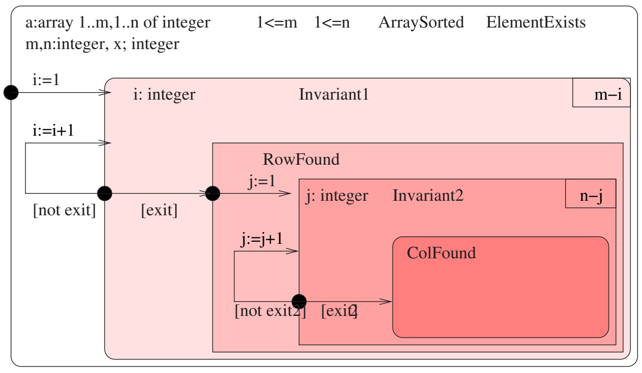


Figure 22: Whole sorting program diagram

Let us finally consider how to describe this algorithm in terms of the domain in which the algorithm is working. This is shown in Figure 23. This shows the subsequent narrowing of the search space. Each indicated area essentially says that the element x is somewhere in this area. Note that the sets are included in each other.

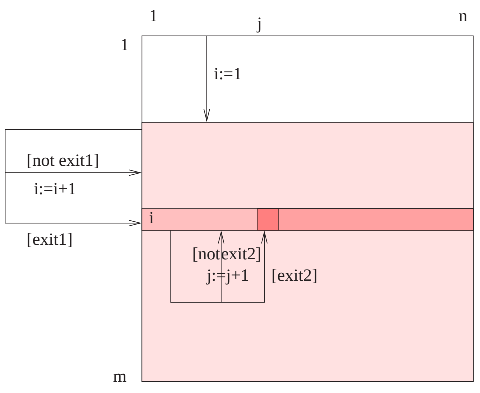


Figure 23: Invariants of sorting program

8 Situation analysis

The invariant diagrams provide a rather simple way of describing the overall structure of the program. However, it is difficult to write out proofs for the transitions in this notation. Therefore, we will provide another format, a textual one that we refer to as *situation analysis*, which can be seen as a complement to the diagram. Situation analysis emphasises the proofs of the verification conditions and is equivalent to a proof of the correctness of the invariant based program.

The textual representation is essentially a list of situations. Each situation is given a name and a short informal explanation. For each situation, we give the conditions associated with that situation, indented one step to the right. For each situation, we also give the transitions that lead out from this situation. We refer to each transition as a **case**. Each case is also indented one step to the right, and can have a descriptive name, if needed. For each case, we provide the statements that are carried out, terminated with a clause that expresses the target situation. We could have written here **goto** "target situation", but we write instead **check** "target situation", to emphasize that the checking of the verification condition is the essential point here.

Invariant based programs as proofs We have two different ways of writing the statements that lead from one invariant to the next. The first approach is to go all the way with the "program as a proof" correspondence, and write the tests as just additional conditions, and write the assignments $x := e$ in the form $x' = e$. In other words, we introduce new (primed, double primed etc.) attributes for each assignment. This allows us to look at an assignment statement as condition also. Then a statement will just amount to a collection of assumptions that lead up to a check-statement. There is an implicit statement like $k, sum := k', sum'$ before returning to an invariant. This can be optimized away in code generation. Algorithm 3 shows the simple summation program in text format:

We also need to include definitions in context, as shown later. These definitions may assume the facts in the path leading up to the definition itself (including other definitions). This allows for flexible mixing of special cases facts and definitions (e.g. piecewise definitions).

The indentation attempts to be systematic and meaningful: an invariant has all its constituents indented one step. The constituents of an invariant are :

- the definitions needed to express the invariant,
- the properties that hold in the invariant,
- the transitions from the invariant (the cases), and
- the invariants that are nested inside this invariant.

Loop termination requires an extra proof obligation (need to reduce termination function before returning). Loop termination also requires that boundedness of the termination function follows from the invariant. Both of these conditions are shown in Figure 3 as additional **check** conditions. We can also have other "check" clauses, e.g. before a conditional statement to check that some alternative is enabled, and before basic operations to check that the operation is defined in the present state.

Proofs are written at the endpoints of the outline, and can be hidden when they are not needed. Assumptions that can be in a proof are all situated at the path that leads from the root to the **check** clause expressing the proof obligation. Algorithm 4 shows the same summation algorithm, but now the proof are written out.

Note that the proofs are written as structured derivations. In this case, the proof are carried out by hand. In practice, doing all the proofs by hand is tedious, and in may

Algorithm 3 The summation program

```
Initial: the initial situation

    $n : integer$

    $n \geq 0$

    case    initialize:
            $k', sum' = 0, 0$
            check $Adding(k', sum')$
    Adding: we have computed the sum up to $k$
            $k, sum : integer$
            $sum = 1 + 2 + \cdots + k$
            $0 \leq k \leq n$
            check   $n - k \geq 0$
            case    exit:
                    $k = n$
                    check $Final$
            case    loop:
                    $k \neq n$
                    $k' = k + 1$
                    $sum' = sum + k'$
                    check $Adding(k', sum')$
                    check $n - k' < n - k$
            Final:  (the whole sum has been computed)
                    $n = k$
```

cases also frustrating, because a large fraction of the lemmas to be proved are rather trivial. I better approach is to build computer support for this approach, where we try to automate the proof. A mechanical proof checker should typically be able to prove 60 - 80% of the lemmas automatically.

If we were using a mechanical proof checker, then the system could automatically analyze the program, and insert at the endpoints (after the check clauses) only those lemmas that it has not been able to prove. The intended effect would be that the system continuously analyzes the program being built and works as a *critic* for the programmer. It lists those lemmas that it cannot itself see that are correct, and asks the programmer to verify these. The programmer then either verifies these by hand, or uses an interactive proof checker.

If we are unable to prove that a check is true, then there is an error in the program. Remember that the notion of a program error is stronger than usual here. It is an error if the program does not preserve all invariants. It is thus possible that the program actually works correctly in the sense that the right output is always produced for any legal input. If the invariants are violated, then the program is still incorrect in the sense that it *does not work as intended*. The remedy is to either change the program so that it works as intended (by modifying the transitions), or change the way we intend the program to work (modify the invariants). Or then change both, in order to achieve consistency.

The above derivation shows the correctness proof of the algorithm. Stripping the program of all extra material, we get the following go to program (Algorithm 5):

Algorithm 5 The summation program

```
$Initial$: $k, s := 0, 0$; $Adding$

$Adding$:      if $k = n$ then $Final$ else $k := k + 1; s := s + k$; $Adding$

$Final$:
```

Refinement calculus formulation of invariant based programs The second approach to expressing statements is to write assume, assert and the assignment statements in the situation analysis as is done in the invariant diagrams. A case collects together some transitions that lead from one situation to another (or back to the situation).

Consider a case of a transition from P to Q with a statement S. The consistency requirement is that $P \subseteq wp.S.Q$. This is again equivalent to

$$wp.([P];S;\{Q\}) = true$$

If we use the refinement calculus convention that a statement S really stands for $wp.S$, then we can consider each case as the proof obligation

$$[P];S;\{Q\} = true$$

The advantage of this view is that we can use different methods to establish the equality. We can move the postcondition backwards (as an assert statement) and we can move the precondition forwards (as an assume statement), without changing the truth of the case condition. In particular, we can move the precondition all the way back to immediately follow the precondition, in which case we just have an implication to prove. Similarly, we can move the precondition all the way forward to the postcondition and again only have an implication to prove.

The following is an example of verifying a check statement by propagating the assertions backwards. The case $Initial : Adding : case 2$ can be written equivalently as in Algorithm 6. This shows that the loop case is in fact true.

Algorithm 6 Case analysis

```
case loop:

    $k \neq n$

    $k := k+1$

    $sum := sum + k$

    check $Adding$

    •    $[Initial]; [Adding]; [k \neq n]; k := k+1; sum := sum + k; \{Adding\}$
    =    $[Initial]; [Adding]; [k \neq n];\ \{sum + k + 1 = 1 + 2 + \cdots + k + 1 \wedge 0 \leq k+1 \leq n\}$
    =    $Initial \wedge Adding \wedge k \neq n \Rightarrow sum + k + 1 = 1 + 2 + \cdots + k + 1 \wedge 0 \leq k+1 \leq n$
    =    $Initial \wedge Adding \wedge k \neq n \Rightarrow sum = 1 + 2 + \cdots + k \wedge 0 \leq k < n$
    =    $\{Adding \Rightarrow sum = 1 + 2 + \cdots + k \wedge 0 \leq k \leq n\ \}$
         $true.$
```

Situation analysis of the array search algorithm. Algorithm 7 shows another example, the array search algorithm in textual form. In this case, we also have some definitions that are used to express the invariants in the program.

Tools for situation analysis An outlining editor has the ability to collapse the subheadings that are not interesting for the present analysis. By using an outlining editor for building and analysing invariant based programs, we can concentrate on one check at a time. All the assumptions needed to verify that the check is true are on the path that leads from the root of the outline to the check. Hence, if the check is visible, then all assumptions for the check must also be visible. All other branches in the outline can be kept closed, in order to not distract from the proof task at hand. In this way, an outlining editor provides a simple way of keeping track of the proof obligations. It also allows the proof itself to be hidden, when one wants to look at the overall program structure but is not concerned with the consistency proofs of the individual transitions.

Advantages of situation analysis An important advantage of the situation analysis is that it combines in one representation a number of different things:

1. It shows a program that when executed achieves the stated goals
2. It shows the proof of the correctness of the program, together with the verification conditions that need to be established.
3. The proof is isomorphic with the program structure.
4. It shows the structure, and scoping of the program variables that are needed in the program

5. It shows the structure and nesting of the program invariants and intermediate conditions that are needed in the program.

9 Conclusions

We have in this paper argued for a different approach to constructing simple algorithms, where we start by constructing the preconditions, postconditions and intermediate invariants of the program (called 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 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 (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. We have also provided a textual form for invariant based programs (situation analysis). This thus gives us three different ways of representing the same invariant based program: figures, invariant diagrams and situation analysis. Each of these have their use. Figures and the transitions between figures are very useful in the initial stages of the construction, when we are trying to work out the intermediate situations or invariants. The invariant diagram provides a concise and compact overview of the whole program, and shows the basic structure of the program. The situation analysis is to be preferred when we carry out the proofs of the consistency of the invariant based program. Invariant diagrams and situation analysis are more or less isomorphic descriptions of the same thing, and it is easy to see that one can be derived from the other.

We have also given a collection of examples showing how to derive invariant based programs. We have on purpose restricted ourselves to a rather simple application domain, array manipulation, because it is a familiar area and the notation for expressing properties of arrays are familiar. We are presently also working with a different application area, the construction of pointer algorithms, where the algorithms are more complex and the problems with expressing the invariants are more sever.

We have on purpose restricted ourselves here to the construction of simple algorithms. However, this approach does scale up quite well to more complicated software constructs, such as procedures, classes, software components and concurrent processes. We are presently working on extensions of the approach in these directions.

Another direction of research that we are currently pursuing is tool support for invariant based programming. We have already built an outlining editor that allows us to carry out situation analysis in a rather straightforward fashion. We have connected this editor to a automatic simplifier and to an interactive proof checker, in order to prove the verification conditions automatically. The initial experiences have been encouraging, and we hope to be able to report on this in the very near future.

References

[1] Ralph-Johan Back. Program construction by situation analysis. Research Report 6, Computing Centre, University of Helsinki, Helsinki, Finland, 1978.

[2] Ralph-Johan Back. Exception handling with multi-exit statements. In H. J. Hoffmann, editor, *6th Fachtagung Programmiersprachen und Programmentwicklungen*, volume 25 of *Informatik Fachberichte*, pages 71–82, Darmstadt, 1980. Springer-Verlag.

[3] Ralph-Johan Back. Invariant based programs and their correctness. In W. Biermann, G Guiho, and Y Kodratoff, editors, *Automatic Program Construction Techniques*, number 223-242. MacMillan Publishing Company, 1983.

[4] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[5] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT*, 8:174 – 186, 1968.

[6] E. W. Dijkstra. Notes on structured programming. In Ole-Johan Dahl, C.A.R Hoare, and E.W Dijkstra, editors, *Structured Programming*. Academic Press, New York, 1972.

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

[8] Martin Fowler. *UML Distilled*. Addison Wesley, 1999.

[9] D. Harel. State charts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[10] E. Hehner. Do considered od: a contribution to the programming calculus. *Acta Informatica*, 11:287 – 304, 1979.

[11] J. C. Reynolds. Programming with transition diagrams. In D. Gries, editor, *Programming Methodology*. Springer Verlag, Berlin, 1978.

[12] M. H. van Emden. Programming with verification conditions. *IEEE Transactions on Software Engineering*, SE-5, 1979.

Algorithm 4 Summation program with proof

```
Initial: the initial situation

    $n : integer$
    $n \geq 0$
    case  initialize:
          $k', sum' = 0, 0$
          check $Adding(k', sum')$
          •     $0, 0 : integer$
          •     $0 = 1 + 2 + \ldots + 0$
          •     $0 \leq 0 \leq n$
    Adding:  we have computed the sum up to $k$
          $k, sum : integer$
          $sum = 1 + 2 + \cdots + k$
          $0 \leq k \leq n$
          check  $n - k \geq 0$
          case   exit:
                 $k = n$
                 check $Final$
                 •     $k = n$
          case   loop:
                 $k \neq n$
                 $k' = k + 1$
                 $sum' = sum + k'$
                 check $Adding(k', sum')$
                 •          $k', sum' : integer$
                 $\equiv$   $k + 1, sum + k + 1 : integer$
                 $\equiv$   $true$
                 •          $sum' = 1 + 2 + \ldots + k + k + 1$
                 $\equiv$   $sum + k + 1 = 1 + 2 + \ldots + k + k + 1$
                 $\equiv$   $sum = 1 + 2 + \ldots + k$
                 $\equiv$   $true$
                 •          $0 \leq k' \leq n$
                 $\equiv$   $0 \leq k + 1 \leq n$
                 $\equiv$   $true$
                 check $n - k' < n - k$
                 •          $n - (k + 1)$
                 $=$        $n - k - 1$
                 $<$        $n - k$
          Final: (the whole sum has been computed)
                 $n = k$
```

Algorithm 7 Array search program

```
define $RowSorted(a,n,i) \equiv (\forall j \in [1,n) \cdot a[i,j] \le a[i,j+1])$
define $ArraySorted(a,m,n) \equiv (\forall i \in [1,m] \cdot RowSorted(i)) \wedge (\forall i \in [1,m) \cdot a[i,n] \le a[i+1,1])$
define $ElementExists(a,m,n,x) \equiv (\exists i \in [1,m], j \in [1,n] \cdot a[i,j] = x)$
Initial: (assumptions at start of program)
    $a : array\,[1..m, 1..n]\, of\, integer$
    $m,n,x : integer;$
    $ArraySorted(a,m,n)$
    $ElementExists(a,m,n,x)$
    $1 \le m \wedge 1 \le n$
    case   $i = 1;$
           check $Invariant1$
    Invariant1: (looking for the right row)
        $i : integer$
        $1 \le i \le m$
        $a[i,1] \le x$
        check  $m - i \ge 0$
        case   found right row
               $i = m \vee x < a[i+1,1]$
               check $RowFound$
        case   row not found
               $i \ne m \wedge x \ge a[i+1,1]$
               $i' = i+1$
               check $Invariant1(i')$
               check $m - i' < m - i$
        RowFound: (we have the right row)
               $i = m \vee x < a[i+1,1]$:
               case   $j = 1$
                      check $Invariant2$
               Invariant2: (looking for the right column)
                      $j : integer$
                      $1 \le j \le n$
                      $a[i,j] \le x$
                      check  $n - j \ge 0$
                      case   (found right column)
                             $j = n \vee x < a[i,j+1]$
                             check $ColFound$
                      case   (not yet right column)
                             $j \ne n \wedge x \ge a[i,j+1]$
                             $j' = j+1$
                             check $Invariant2(j')$
                             check $n - j' < n - j$
                      ColFound: (found the right column)
                             $j = n \vee x < a[i,j+1]$
```

ISBN ISBN 952-12-1496-1
ISSN 1239-1891