Tool Support for
Invariant Based Programming

Ralph - Johan Back and Magnus Myreen
Abstract

Invariant based programming is an approach to program construction where we provide the program pre- and postconditions as well as loop invariants before we construct the code itself. This approach allows us to construct a program and its correctness proof hand in hand. We describe here an extension to an existing mathematics editor that supports this style of program construction. The main help that the tool provides is automatic simplification of verification conditions that are generated in the programming process. The tool shows the user a check list of those conditions that it was not able to prove automatically. The user can use this check list to complete the proof (either manually or using an interactive theorem prover) or to find errors in the program.

Keywords: program construction, program verification, correctness, invariant based programming, situation analysis, invariant diagrams, state charts, weakest preconditions, mechanized verification, automatic simplification, algorithms

TUCS Laboratory
Software Construction Laboratory
1 Introduction

Invariant based programming is an approach to program construction where the program invariants are written before the code itself [5, 2]. This approach has been considered, in a number of different forms and with different details by, e.g., Dijkstra [10, 11, 12], Reynolds [18], Back [2, 3, 4] and van Emden [20]. Formulating program invariants explicitly increases our understanding of the program logic and makes it much easier to verify the correctness of the program. When invariants are produced as part of the programming process, the main obstacle to program verification, finding the appropriate class and loop invariants, simply disappear. What remains, however, is how to prove the correctness of the verification conditions for the program. A program proof will generate a large number of lemmas to be proved. Most of these are quite simple and shallow, and their proof can often be automated.

We describe here a tool that supports both the construction and the verification of invariant based program. The main focus is on helping the programmer to prove the verification conditions for the program. The tool integrates an automatic simplifier with an existing tool for carrying out mathematical derivations. Our aim is to give a proof of the concept that coding and verification can be combined in a natural and practical way.

We describe the idea of invariant based programs and how to represent programs using them in Section 2. In Section 3 we give a brief account of the textual representation of invariant based programs and of the programs that our tool uses. In Section 4 we present case studies on binary search, partition and quick sort. We end with conclusions and some perspectives on further work.

2 Invariant based programs

As already said, the main idea in invariant based programming is to formulate the program pre- and postconditions and the program invariants before constructing the code. A general overview of invariant based programming is given in [5]. Here we will be content with a brief overview of the approach.

The work flow for constructing a simple invariant based program is essentially the following:

Problem Start from an informal description of the programming problem.

Requirements Analyze the problem and write down the precondition and postcondition of the program to be written

Algorithm Work out a rough informal idea of the algorithmic solution to the programming problem.

Invariants Identify the central repeating situations (invariants) in the execution of the algorithm, and write down these invariants

Transitions Show how to move forward from each non-terminal situation (precondition or invariant) by executing program statements (the transitions)
Verification Verify that each transition preserves the correctness of the program invariants and that there are no infinite loops.

Invariant based programs are conveniently described using invariant diagrams [5]. These are directed graphs such that the nodes are predicates and the edges are transitions. The predicates describe properties of the state of the program and the transitions are updates of the state. The state updates are simple program statements: assignments, procedure calls, if-statements, assumptions and assertions. Loop constructs are not allowed in transitions.

The programming problem is to devise a path of transitions that lead from initial states to goal states. The disjunction of the initial state predicates can be referred to as the precondition of the program and the disjunction of the goal state predicates can be considered to be the postcondition. The challenge is to find intermediate predicates describing the stages the program visits while making progress towards a predicate describing a goal state.

Figure 1 describes a small invariant diagram for calculating the greatest common divisor of two positive integers \( m \) and \( n \). The rectangular boxes give the different stages that the program can be in during execution and the arrows define the transitions between the stages. The transitions can branch on conditions. The condition of a branch is written above the line of the arrow and the program statements that realise the state update are written below the arrow. To avoid ambiguity, we sometimes put the conditions between brackets, but these are omitted in this diagram.

The invariant rectangles in the diagram can be nested. Figure 2 shows an alternative invariant diagram for the algorithm in Figure 1. The predicate that restricts the types of the variables and requires that \( m > 0 \land n > 0 \) has been taken out of the rectan-
Figure 2: A nested invariant diagram for computing the GCD as in Figure 1.

In invariant based programming really has two different aspects. The first aspect, emphasized above, is that we write down the program invariants before the code. This forces us to a new way of thinking about program execution, and encourages a more
careful and thorough style of programming. Expressing the invariants explicitly re-
quires that one really understands the program behavior in detail. Expressing the in-
variants before the code is thus very useful in itself.

The other aspect is that once the invariants have been precisely formulated, it be-
comes easier to check the correctness of the program. The main obstacle to program
verification has traditionally been seen as finding the loop invariants for a given pro-
gram. This problem is taken care of in invariant based programming. The remaining
problem is to check that the verification conditions are correct. In practice, the prob-
lem is that there are very many conditions that need to be checked, so the task is quite
time consuming. Also, most of the checks are mathematically quite trivial and hence
boring. On the other hand, this is the perfect place for machine automation.

There is a long tradition of using computers to prove verification conditions. Our
approach is probably closest to the semantic checker that has been developed by Leino
and Nelson [15, 9], and it also uses the same proof/simplification engine, Simplify,
which has been developed by Nelson [16, 7].

We can use a computer to verify the verification conditions. As the verification
conditions are not in themselves interesting, the computer should only show those
conditions that it is not able to verify. The computer can fail to prove a verification
condition because the verification condition is not true, i.e., there is an error in the
program, or because it is too difficult to prove. The latter can again be because the
proof is mathematically too difficult, or because the computer does not know enough
about the underlying theory.

In any case, the programmer needs to take actions on the verification conditions
that are not proved automatically. The action depends on the programmers understand-
ing of the situation. He can either try to correct an error in the program, or he can try
to prove the condition by hand, or then he can provide additional information to the
system and then try to reverify the condition automatically. In case the verification
condition is wrong, then the error may either be in the invariants or in the transitions.
Either one, or both, have to be adjusted in order to get a verification condition that is
actually true.

The tool that we have developed has proved to be capable of proving the cor-
rectness of almost all the correctness conditions, as well as pointing out mistakes the
programmer may have made when the proof has not succeeded. Many of these errors
being simple “off-by-one” index errors. We will next describe this tool in somewhat
more detail.

3 An editor for invariant based programs

We start with a brief description of the syntax used to represent invariant based pro-
grams in a textual form. We refer to this representation as situation analysis [2, 3].
In the textual representation the rectangles of the diagrams are called situations or re-
gions. The term “region” refers to the interpretation that the rectangles are considered
to be subsets of the state space.

There is one feature in the textual representation that isn’t covered in the diagrams,
namely procedure definitions. Adding procedures does not introduce a new concept,
but restricts the diagrams to a certain format. We make this restriction in order to get
Algorithm 1 The GCD program in the textual representation.

procedure GCD (m, n : Int; result r : Int)
  PRE m > 0 ∧ n > 0
  PRE m = m₀ ∧ n = n₀
  POST gcd(m₀, n₀) = r
  [] m ≥ n → LOOP
  [] m < n → m, n := n, m; LOOP
region LOOP [m]
  • m ≥ n ∧ n > 0
  • gcd(m₀, n₀) = gcd(m, n)
  [] m = n → r := m; POST
  [] m > n ∧ m − n ≥ n → m := m − n; LOOP
  [] m > n ∧ m − n < n → m, n := n, m − n; LOOP

clean interfaces between procedures. The diagrams are presented in textual form as procedures with one region defining the allowed initial states and one region for the final states. These regions are called PRE and POST respectively. This is in fact an unnecessary restriction, as multiple exit statements form a much more natural interpretation for transitions in refinement diagrams [3]. However, we make this restriction here for simplicity.

The syntax for procedures is the following:

procedure name (parameter declaration) [variant]
preconditions
postconditions
local variables
initial transitions
region definitions

where each region definition has the form

region name [variant]
assertions
transitions
region definitions

All the lists, preconditions, postconditions, transitions, assertions etc, are separated by new-line characters. The precondition of the procedure is the conjunction of the lines starting with PRE. The postcondition is the conjunction of lines beginning with POST. Variables are declared on lines starting with var and transitions on lines starting with [] followed by a condition and →. The notation for transitions is inspired by Dijkstra’s notation for selection in the guarded commands language[12]. A textual representation of the GCD program is shown in Algorithm 1.

Our tool for checking programs was built as part of MathEdit [6]. MathEdit is a
Text editor for writing mathematical papers, implemented in Python [21]. It assists in writing proofs in a derivational style that supports nesting. The user defines the syntax and rules that are to be used. MathEdit can then apply once or repeatedly rules to expressions. It verifies that proofs are correct and highlights mistakes.

MathEdit has been extended to support verification of programs written in the style described in this report. The parser parses the expressions in the program statements as normal MathEdit expressions and hence makes it possible to define and use user-defined syntax and types from MathEdit in the programs. It performs standard semantical checks before producing the verification conditions. The verification conditions for invariants are produced by applying the \( wp \) function to the program statements in the transitions, as described in [5]. The conditions on variants are not well supported at the time of writing. They are unnecessarily strong. A variant is required to decrease at every transition inside the strongly connected components of the graph.\(^1\)

MathEdit uses Simplify [7] as an external validity checker. The translation of the predefined syntax of expressions is done in a straight-forward manner for the operators that have corresponding operators in Simplify’s input language, for example \( \neg \), \( \land \) and \( \forall \). Operators that do not translate straight to Simplify’s input language are translated to functions where the name of the function includes an encoding of its name and type in MathEdit.\(^2\) Some operators cannot be translated, for instance operators that introduce local scope (except \( \forall \) and \( \exists \)).

The connection from MathEdit to Simplify is not solely intended for checking verification conditions, it can also be used in normal MathEdit proofs. MathEdit also uses Python’s built-in simplifier and has a simple interface to HOL. In the future, we hope that it will be possible to use MathEdit as a front-end to a variety of proof tools.

The logical step after parsing, semantical checks and proof of correctness would be to compile the program into an executable. At present our tool does not compile programs, but the translation to common imperative programming languages is straightforward provided that the user indicates how user-defined syntax is to be translated and that the program statements are executable. The resulting program would make use of constructs equivalent to labels and goto-statements.

4 Case studies

We will demonstrate how our tool can be used in invariant based programming by going through a few case studies: binary search, partition and quick sort. The case study on binary search is an introduction to our tool, the case study on partitioning shows how partial implementations can be checked and the case study on quick sort shows how unproved verification conditions can be narrowed down to simpler conditions.

\(^1\)A better approach would be to have the programmer indicate where the variant is meant to decrease and then have MathEdit check whether it does decrease and whether the indicated places are sufficient to guarantee termination.

\(^2\)Variables and functions are not declared in Simplify’s input language. Hence, the type name must be included in order to make a distinction between \( f : A \rightarrow A \) and \( f : B \rightarrow B \).
4.1 Binary search

Binary search is a simple algorithm that finds an element $x$ in a sorted array $a$ in $O(\log N)$ time where $N$ is the length of $a$. It achieves the fast running time by canceling out one half of the remaining segment for each cycle. We choose to present a development of binary search because it is an algorithm with a simple idea, but where mistakes are easily made when writing an implementation.

**Specification**  The first step in implementing binary search by invariant based programming is to decide what the program should do. We need to define the parameters as well as and the pre- and postcondition for the procedure. A rather obvious definition of $\text{BinarySearch}$ is then the following.

```latex
procedure BinarySearch (const $a$: Int[0..N]; const $x$: Int; result $n$: Int)

PRE ($\forall i$: Int $•$ $0 < i \land i < N \rightarrow a[i-1] \leq a[i]$)

POST ($\forall i$: Int $•$ $0 \leq i \land i \leq n \rightarrow a[i] \leq x$)

POST ($\forall i$: Int $•$ $n < i \land i < N \rightarrow x < a[i]$)
```

Before tackling the problem of how to get from PRE to POST, it is worth thinking twice about what problem is be solved. Finding a program for the wrong pre- and postconditions is a waste of time. By looking closely at the conditions one might even be able to simplify them and maybe solve a more general problem.

Now suppose the list has more than one element and that $a[0] \leq x \land x < a[N-1]$, i.e. $n$ has to lie in $[0, N)$. Using transitivity of $\leq$ and $<$ we can then define the postcondition as follows:

- POST $0 \leq n \land n < N - 1$
- POST $a[n] \leq x \land x < a[n+1]$

As Kaldewaij points out in [14], binary search doesn’t need to assume a sorted list in order to reach this postcondition. Why would we then assume a sorted list? In fact, with a few changes to the postcondition, binary search can be defined with no precondition at all. In order to avoid cluttering the program with trivial cases we use the following precondition:

- PRE $N > 1$
- PRE $a[0] \leq x \land x < a[N-1]$

This means that we do not consider the situation when the value $x$ is less than the first element in the array or greater or equal to the last element in the array. These cases can, however, be taken care of by a direct test at the beginning of the program. This test would then also establish the precondition above.

**Implementation**  Using the simpler pre- and postcondition, a first approximation of the $\text{BinarySearch}$ might look something like the program shown in Algorithm 2.

At this point we would usually try to justify the implementation by informal reasoning or by testing a number of inputs. However, we demonstrate the tool here by just plunging into the verification task directly, and ask the tool to prove this implementation correct.
Algorithm 2 The binary search program, first version.

```
procedure BinarySearch (const a : Int[0..N); const x : Int; result n : Int)
  PRE N > 1
  PRE a[0] ≤ x ∧ x < a[N − 1]
  POST 0 ≤ n ∧ n < N − 1
  POST a[n] ≤ x ∧ x < a[n + 1]
  var m, n, k : Int
  [] ⊤ → n, m := 0, N; SPLIT
  region SPLIT [ m − n ]
    • a[n] ≤ x ∧ x < a[m]
    • n < m
    [] m − n = 1 → POST
    [] m − n > 1 →
      k := (m + n) div 2;
      if a[k] ≤ x → n := k; SPLIT
      [] x < a[k] → m := k; SPLIT
```

Verification When our tool is asked to prove the program in Algorithm 2 it prints out four verification conditions that need attention. The first condition is proved not to be valid. The computer shows us the part of the verification condition that it cannot prove correct (or, as in this case, it can prove to be incorrect). In this case, we get the following:

Condition: Case ⊤ in initial transitions (BinarySearch)
  • Note: This was proved to be false by a validity checker.
Assumptions:
  N > 1
  a[0] ≤ x
  x < a[N − 1]
  0 ≤ N
Imply:
  N < N
  x < a[N]

The first condition after “imply” comes from the array declaration (each array access has to have an index in the indicated array range). From this condition we see immediately that there is something inconsistent between our specification and implementation. The bug is easy to fix. The mistake was to initialise m to N instead of N − 1.

Our tool is unable to determine whether the other three verification conditions are valid or not. The first one of the other three unproved verification conditions is:
Algorithm 3 The binary search program, final version

procedure BinarySearch (const a : Int[0..N]; const x : Int; result n : Int)
    PRE N > 1
    PRE a[0] ≤ x ∧ x < a[N - 1]
    POST 0 ≤ n ∧ n < N - 1
    POST a[n] ≤ x ∧ x < a[n + 1]
    var m, n, k : Int
    [] ⊤ → n, m := 0, N - 1; SPLIT
    region SPLIT [ m − n ]
        • a[n] ≤ x ∧ x < a[m]
        • 0 ≤ n ∧ n < m ∧ m < N
        [] m − n = 1 → POST
        [] m − n > 1 →
            k := (m + n) div 2;
            if a[k] ≤ x → n := k; SPLIT
            [] x < a[k] → m := k; SPLIT

Condition: Case m − n = 1 in SPLIT (BinarySearch)
Assumptions:
    a[n] ≤ x
    x < a[m]
    n < m
    m − n ≥ 0
    0 ≤ N
    m − n = 1
Imply:
    0 ≤ n
    n < N
    0 ≤ n + 1
    n + 1 < N
    0 ≤ n
    n < N − 1

This condition indicates that our specification isn’t strong enough. We allow n to have too wide a range of values. Our intuition tells us that 0 ≤ n and n < N − 1. Rather than writing n < N − 1, we will write m < N because n < m. This correction is a correction of the two remaining verification conditions as well. Our tool is able to prove the updated binary search program correct. The final program is shown in Algorithm 3.

Now suppose we made a worse mistake. Suppose we wrote the conditions for the transitions from SPLIT as m − n = 0 and m − n > 0. This mistake makes all the transitions correct, but it will not terminate. When we try to verify the program with these mistakes, our tool brings this correctness condition to our attention:
Condition: Variant decreases when leaving SPLIT (BinarySearch)

Assumptions:

\[
\begin{align*}
  a[n] &\leq x \\
  x &< a[m] \\
  0 &\leq n \\
  n &< m \\
  m &< N \\
  m - n &\geq 0 \\
  0 &\leq N \\
  V & = m - n \\
  m - n &> 0 \\
  a[(m+n) \div 2] &\leq x
\end{align*}
\]

Imply:

\[
m - (m+n) \div 2 < V
\]

With a little experience it becomes easy to spot the mistake from unproved conditions like this.

**The verification process** Our tool verifies the invariant based program by generating the verification conditions and uses a validity checker to attempt to prove the conditions either valid or not valid. If the validity checker is unable to prove the validity of a condition, the condition is split into a conjunction of smaller conditions. For instance, a verification condition is often of the form

\[
A_1 \land A_2 \land \ldots \land A_m \Rightarrow B_1 \land B_2 \land \ldots \land B_n
\]

which is then split into

\[
(A_1 \land A_2 \land \ldots \land A_m \Rightarrow B_1) \land (A_1 \land A_2 \land \ldots \land A_m \Rightarrow B_2) \land \ldots \land (A_1 \land A_2 \land \ldots \land A_m \Rightarrow B_n)
\]

All of the new conditions are tested and the ones that were not proved are shown to the user. The original conditions can be very long and usually contain a lot of simple conditions on bounds of variables. The amount of detail that the user has to check is thus considerably less in this approach than what it would be without the tool.

**4.2 Partition**

Next we show how the process of specification, implementation and verification can be combined into the single process of developing a program using our tool.

**Specification** A partitioning algorithm rearranges a list so that the elements are split into groups. We consider a procedure that splits the elements into two groups: one with all elements \( \leq x \) and the other one with all elements \( > x \), where \( x \) is the value of some element in the original list. A straight-forward specification of such a partitioning program:
procedure Partition (valres a : Int[0..N); result k : Int)
  PRE N > 0
  PRE permutation(a₀, a)
  POST permutation(a₀, a)
  POST (∀i : Int ⋆ 0 ≤ i ≤ k → a[i] ≤ a[k])
  POST (∀i : Int ⋆ k < i < N → a[k] < a[i])
  POST 0 ≤ k ∧ k < N

Again it is worth thinking carefully about the pre- and postconditions before proceeding. The specification can be made more general by only considering a segment of the array. Here is a specification that partitions only the segment a[m..n):

procedure Partition (valres a : Int[0..N); value m, n: Int; result k : Int)
  PRE 0 ≤ m ∧ m < n ∧ n ≤ N
  PRE m = m₀ ∧ n = n₀
  PRE permutation(a₀, a)
  POST permutation(a₀, a)
  POST (∀i : Int ⋆ m₀ ≤ i ≤ k → a[i] ≤ a[k])
  POST (∀i : Int ⋆ k < i < n₀ → a[k] < a[i])
  POST (∀i : Int ⋆ 0 ≤ i < m₀ → a₀[i] = a[i])
  POST (∀i : Int ⋆ n₀ ≤ i < N → a₀[i] = a[i])
  POST m₀ ≤ k ∧ k < n₀

The identifiers with subscripted zeros are initial value constants. They are handy to use in specifications. Even though the above specification is not complicated, it is getting hard to read. We will use the following syntactic abbreviations to make the definition more readable.

\[
\begin{align*}
gt(a, x, m, n) &= (∀i : Int ⋆ n ≤ i < m → a[i] > x) \\
\itreq(a, x, m, n) &= (∀i : Int ⋆ n ≤ i < m → a[i] ≤ x) \\
equal(a, b, n, m) &= (∀i : Int ⋆ n ≤ i < m → a[i] = b[i])
\end{align*}
\]

Using these abbreviations the specification can be written as:

procedure Partition (valres a : Int[0..N); value m, n: Int; result k : Int)
  PRE 0 ≤ m ∧ m < n ∧ n ≤ N
  PRE m = m₀ ∧ n = n₀
  PRE permutation(a₀, a)
  POST permutation(a₀, a)
  POST \itreq(a, a[k], m₀, k + 1) ∧ gt(a, a[k], k + 1, n₀)
  POST equal(a₀, a, 0, m₀) ∧ equal(a₀, a, n₀, N)
  POST m₀ ≤ k ∧ k < n₀

These abbreviation may not seem easier to read, but when they start recurring, they are easier to pick out and understand.
**Implementation and verification**  The most intuitive way of getting from **PRE** to **POST** in Partition is to choose an element $a[k]$ and then go through the remaining segment collecting the elements $\leq a[k]$ at the low end of the segment and those $> a[k]$ at the high end. Stated more precisely, choose a $k$ such that $n_0 \leq k < m_0$ and maintain $\text{lteq}(a,a[k],m_0,m) \wedge \text{gt}(a,a[k],n,n_0)$ whilst making progress towards a situation where $m = n$. This presentation leads to following initial transition and region if we take $k = m$:

$$
\begin{align*}
\top & \rightarrow \text{LOOP} \\
\text{region LOOP } [ n - m ] & \cdot \text{lteq}(a,a[m],m_0,m) \wedge \text{gt}(a,a[m],n,n_0) \\
& \cdot m_0 \leq m \wedge m < n \wedge n \leq n_0 \\
& [ ] n - m \leq 1 \rightarrow k := m; \text{POST} \\
& [ ] n - m > 1 \rightarrow \ldots
\end{align*}
$$

At this stage of writing the program it might be useful to check whether it is correct so far. The correctness of a partial implementation can be checked by inserting the program statement $\text{magic}$ at points that are to be ignored by the correctness check. Hence we change the second transition of the region LOOP to:

$$
[ ] n - m > 1 \rightarrow \text{magic}
$$

It is now possible to check the correctness of the partial implementation. When this is done our tool brings a long condition to the users attention. Most of the unproved requirements concern restrictions on $i$ in the definitions of $\text{lteq}$ and $\text{gt}$. By adding the following assertion to LOOP, we get rid of the unproved conditions:

$$
\cdot 0 \leq m_0 \wedge n_0 \leq N
$$

If the correctness is checked again, the condition is short and informative:

**Condition:** Case $n - m \leq 1$ in LOOP (Partition)

**Assumptions:**

- $\text{lteq}(a,a[m],m_0,m)$
- $\text{gt}(a,a[m],n,n_0)$
- $0 \leq m_0$
- $m_0 \leq m$
- $m < n$
- $n \leq n_0$
- $n_0 \leq N$
- $n - m \leq 0$
- $0 \leq N$
- $n - m \leq 1$

**Imply:**

- $\text{permutation}(a_0,a)$
- $\text{equal}(a_0,a,0,m_0)$
- $\text{equal}(a_0,a,n_0,N)$

We notice that our assertions at LOOP do not guarantee that the array is untouched for indexes $< m_0$ and $\geq n_0$ and that the array is a permutation of the original one. We
can easily fix this by strengthening the assertions with

- $\text{permutation}(a_0, a)$
- $\text{equal}(a_0, a, 0, m_0) \land \text{equal}(a_0, a, n_0, N)$

It remains to replace $\text{magic}$ with executable program statements that give a transition back to LOOP. After some sketches on paper one might suggest something similar to that later. We have assumed an implementation of $\text{Swap}$. The definition of $\text{Swap}$ is given below.

\[
\begin{align*}
\text{if } a[m + 1] < a[m] & \rightarrow \text{Swap}(a, m, m + 1); m := m + 1; \text{LOOP} \\
[1] & a[m + 1] = a[m] \rightarrow m := m + 1; \text{LOOP} \\
[1] & a[m + 1] > a[m] \rightarrow \text{Swap}(a, m + 1, n); n := n - 1; \text{LOOP}
\end{align*}
\]

There is a bug in the code above. It is not easy to spot. By trying to check the correctness we can use our tool to spot any bugs in the code. It brings the following unproved verification condition to the user’s attention:

**Condition: Case $n - m \leq 1$ in LOOP (Partition)**

**Assumptions:**
- $\text{permutation}(a_0, a)$
- $\text{lteq}(a, a[m], m_0, m)$
- $\text{gt}(a, a[m], n, n_0)$
- $\text{equal}(a_0, a, 0, m_0)$
- $\text{equal}(a_0, a, n_0, N)$
- $0 \leq m_0$
- $m_0 \leq m$
- $m < n$
- $n \leq n_0$
- $n_0 \leq N$
- $n - m \leq 0$
- $0 \leq N$
- $n - m > 1$

**Imply:**

**Assumptions:**
- $a[m + 1] > a[m]$
- $\text{permutation}(a, a_2)$
- $a[m + 1] = a_2[n]$
- $a[n] = a_2[m + 1]$
- $(\forall i : \text{Int}) \cdot (i \neq m + 1) \land (i \neq n) \land 0 \leq i \land i < N \land (a_2[i] = a[i])$

**Imply:**
- $\text{equal}(a_0, a_2, n_0, N)$
- $\text{gt}(a_2, a_2[m], n - 1, n_0)$

**Assumptions:**
- $a[m + 1] > a[m]$

**Imply:**
- $n < N$

We can read that the validity checker is unable to prove the case $a[m + 1] > a[m]$. 

13
The final version of the partition program.

Algorithm 4

procedure Partition (valres $a : \text{Int}[0..N]$; value $m,n : \text{Int}$; result $k : \text{Int}$)

PRE $0 \leq m \land m < n \land n \leq N$
PRE $m = m_0 \land n = n_0$
PRE permutation($a_0, a$)
POST permutation($a_0, a$)
POST $\text{lt}(a, a[k], m_0, k + 1) \land \text{gt}(a, a[k], k + 1, n_0)$
POST equal($a_0, a, 0, m_0) \land equal(a_0, a, n_0, N)$
POST $m_0 \leq k \land k < n_0$

[\[ \top \rightarrow \text{LOOP} \]

region LOOP [$n - m$]

- $\text{permutation}(a_0, a)$
- $\text{lt}(a, a[m], m_0, m) \land \text{gt}(a, a[m], n, n_0)$
- $\text{equal}(a_0, a, 0, m_0) \land \text{equal}(a_0, a, n_0, N)$
- $m_0 \leq m \land m < n \land n \leq n_0$
- $0 \leq m_0 \land n_0 \leq N$

[\[ \top \rightarrow \text{LOOP} \]

$n - m \leq 1 \rightarrow$

if $a[m + 1] < a[m] \rightarrow \text{Swap}(a, m, m + 1); m := m + 1; \text{LOOP}$

[\[ \top \rightarrow \text{LOOP} \]

$n - m > 1 \rightarrow$

if $a[m + 1] = a[m] \rightarrow m := m + 1; \text{LOOP}$

[\[ \top \rightarrow \text{LOOP} \]

$n - m < 1 \rightarrow$

if $a[m + 1] > a[m] \rightarrow \text{Swap}(a, m + 1, n - 1); n := n - 1; \text{LOOP}$

If we check the correctness after correcting the mistake we get assured that Partition is correct with respect to its pre- and postcondition. The final version of Partition is shown in Algorithm 4.

Implementing swap

It remains to give an invariant based program for Swap. A simple procedure for swapping two elements of an array is shown below:

procedure Swap (valres $a : \text{Int}[0..N]$; const $m,n : \text{Int}$)

PRE $0 \leq m \land m < N \land 0 \leq n \land n \leq N$
PRE permutation($a_0, a$)
POST permutation($a_0, a$)
POST $(a_0[m] = a[n]) \land (a_0[n] = a[m])$
POST $(\forall i : \text{Int} \bullet i \neq n \land i \neq m \land 0 \leq i \land i < N \rightarrow a_0[i] = a[i])$

$\top \rightarrow a := a[m \mapsto a[n]][n \mapsto a[m]]; \text{POST}$
In our language assignments to expressions are disallowed, hence \( a[n] := x \) is not allowed. We use \( a := a[n \mapsto x] \) instead, where the expression \( a[n \mapsto x] \) evaluates to an array exactly the same as \( a \) except that index \( n \) maps to \( x \).

Our tool has difficulties in proving the last part of the postcondition. The reason is that the concept of a permutation cannot be formalised in first-order logic. The validity checker Simplify used by MathEdit cannot use a definition of permutation \((a, b)\).

Even though the validity checker cannot use a definition of permutation we have supplied it with the property that it is an equivalence relation:

\[
\begin{align*}
\top & \rightarrow \text{partition}(a, a) \\
\text{partition}(a, b) & \rightarrow \text{partition}(b, a) \\
\text{partition}(a, b) \land \text{partition}(b, c) & \rightarrow \text{partition}(a, c)
\end{align*}
\]

We can state that we believe \( a[m \mapsto a[n]][n \mapsto a[m]] \) is a permutation of \( a \) by adding an assumption statement to the initial transition:

\[
\bullet \top \rightarrow a := a[m \mapsto a[n]][n \mapsto a[m]]; [\text{permutation}(a_0, a)]; \text{POST}
\]

MathEdit can prove it correct now, but prints a warning that an assumption was made.

### 4.3 Quick sort

Quick sort[8] is a sorting algorithm with an average running time of \( O(N \log N) \). We present a development of an invariant based program for the purpose of showing how recursion is supported by procedures in programs and how assertions can be used to pinpoint problematic cases in the verification conditions.

**Specification**  Quick sort has a standard sorting algorithm contract:

```plaintext
procedure QuickSort (valres a : Int[0..N]; const m, n : Int)
    PRE 0 ≤ m ∧ m < n ∧ n ≤ N
    PRE permutation(a_0, a)
    POST permutation(a_0, a)
    POST sorted(a, m, n)
    POST equal(a_0, a, 0, m) ∧ equal(a_0, a, n, N)
where sorted is an abbreviation of

\[
\text{sorted}(a, m, n) = (\forall i : \text{Int} \bullet m < i ∧ i < n \rightarrow a[i - 1] ≤ a[i])
\]
```

**Implementation and verification**  The strategy of quick sort is to partition the list around an element of the list and then sort each partition through recursion. An implementation:
Before we can check the correctness of QuickSort it must have a variant defined on its parameters. We use the variant \( n - m \), i.e. the length of the segment to be sorted. The first line of the definition is now:

```plaintext
procedure QuickSort(valres a : Int[0..N]; const m, n : Int) [ n - m ]
```

The correctness check proves everything correct except \( sorted(a, m, n) \). With an informed guess we can locate the possible problematic point by adding an assertion just before POST:

\[
\{(p + 1 < N \rightarrow a[p] \leq a[p + 1]) \land (p - 1 \geq 0 \rightarrow a[p - 1] \leq a[p])\};
\]

The guess is that the joining points of the sorted segments are hard to prove. This guess is correct. The validity checker can prove everything except this assertion. Actually the assertion cannot be proved without a definition of \( permutation \). Hence we cannot expect our tool to prove it. The result is still useful because now the problem has been pinpointed to a small proof that can be done by paper and pen or in a theorem prover.

### 5 Conclusions and further work

Implementation and verification can be combined naturally if a detailed specification is written alongside the implementation. Modern validity checkers are strong enough to prove most of the trivial cases where mistakes are made if a precise enough specification is given. Hence our approach is practical if a good representation of programs is used.

We have also shown that verification can be helpful even when it fails. The programmers can focus their attention to the unproved parts of the program. For instance, they can either prove it correct using stronger tools than validity checkers or convince themselves that the program is correct by extensive testing of the parts that could not be proved correct.

A problem with developing programs using invariant based programming is that it requires quite sophisticated programmers, who have a good education in logical reasoning. This fact does not make our approach impractical, but rather points to a problem in the present day education of programmers and software engineers.

There is scope for a lot of work on this topic. The most obvious feature to add is exporting the unproved conditions to interactive theorem provers such as HOL[13, 1] and PVS[17, 19]. This way the user is given the opportunity to prove the program formally correct even if the validity checkers are unable to prove all the verification conditions valid. Other possible improvements and extensions includes augmenting
the language with record and pointer types as well as object-oriented features; and making the specifications and verification conditions more structured and easier to read.

**Acknowledgment**

We would like to thank Victor Bos and Johannes Eriksson for help in adapting the MathEdit system to invariant based programs.

**References**


