

Tool Support for Invariant Based Programming

Ralph-Johan Back and Magnus Myreen

Abo Akademi University, Department of Computer Science
Lemminkäinenkatu 14 A, FIN-20520 Turku, Finland
Email: backrj@abo.fi, magnus.myreen@iki.fi

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.

I. INTRODUCTION

Invariant based programming is an approach to program construction where the program invariants are written before the code itself [1], [2]. This approach has been considered, in a number of different forms and with different details by, e.g., Dijkstra [3], [4], [5], Reynolds [6], Back [2], [7], [8] and van Emden [9]. 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. 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 II. In Section III we give a brief account of the textual representation of invariant based programs and of the programs that our tool uses. In Section IV we present case studies on binary search, partition and quick sort. We end with conclusions and some perspectives on further work.

II. INVARIANT BASED PROGRAMS

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 [1]. Here we will be content with an overview of the approach before presenting how it can be used in practice, section IV.

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

- 1) Start from an informal description of the programming problem.
- 2) Analyze the problem and write down the precondition and postcondition of the program to be written.
- 3) Work out a rough informal idea of the algorithmic solution to the programming problem.
- 4) Identify the central repeating situations (invariants) in the execution of the algorithm, and write down these invariants.
- 5) Show how to move forward from each non-terminal situation (precondition or invariant) by executing program statements (the transitions).
- 6) 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* [1]. 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.

Fig. 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

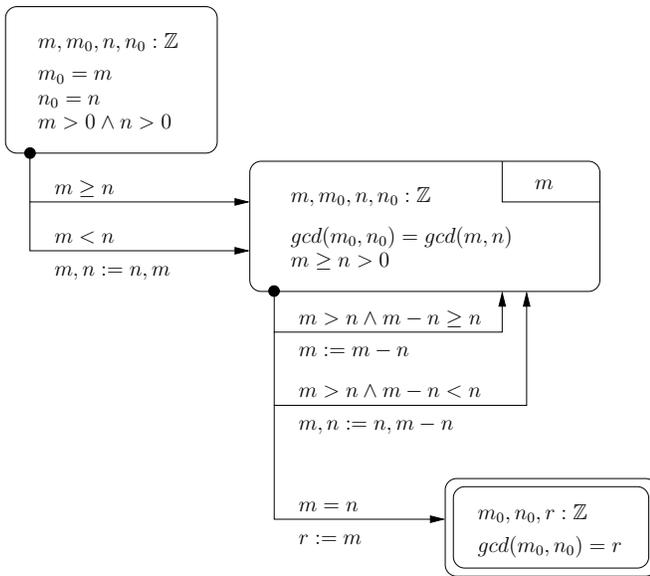


Fig. 1. An invariant diagram for computing the GCD of two positive integers.

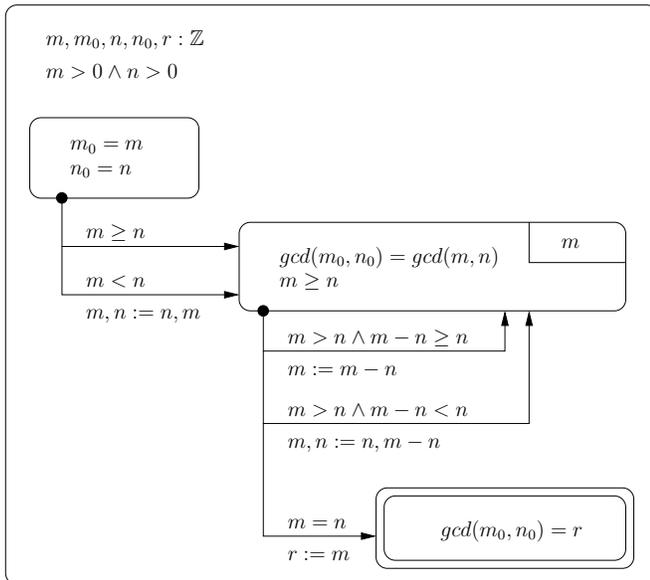


Fig. 2. A nested invariant diagram for computing the GCD as in Fig. 1.

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. Fig. 2 shows an alternative invariant diagram for the algorithm in Fig. 1. The predicate that restricts the types of the variables and requires that $m > 0 \wedge n > 0$ has been taken out of the rectangles and put in a rectangle that contains all the other rectangles. The semantics of this nesting is simple: the predicates associated with a rectangle are those that are

given explicitly in that rectangle together with the predicates associated with all enclosing rectangles.

There is a close analogy between nested invariant diagrams and Venn diagrams. We identify a predicate with the set of states that satisfies the predicate. If we draw box A inside box B , this means that every state in A is also a state in B . In other words, the conditions of B also hold in predicate A .

The analogy cannot be taken much further because invariant boxes cannot be overlapping. The boxes may only be drawn completely inside one another or disjoint from each other. However, disjoint boxes can still have states in common. For instance the states where $m = m_0 = n = n_0 = r$ is true satisfies all of the predicates in all of the boxes in the figures describing GCD.

To prove the correctness of a program described by an invariant diagram, one needs to prove (1) the validity of the transitions and (2) that the program cannot start an infinite loop. A transition from predicate P_1 to P_2 using program statements S is valid if and only if $P_1 \Rightarrow wp.S.P_2$ where wp is Dijkstra's weakest-precondition predicate transformer [5]. We show that a program terminates by providing a bounded variant function, which is bounded from below and is decrease by every cycle in the diagram. We refer to [1] for a more detailed description of invariant based programming and the notion of correctness for invariant based programs.

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 requires that one really understands the program behavior in detail. Expressing the invariants before the code is thus very useful in itself.

The other aspect is that once the invariants have been precisely formulated, it becomes 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 program. 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 problem 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 [10], [11], and it also uses the same proof engine, Simplify, which has been developed by Nelson [12], [13].

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 mathe-

matically 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 understanding 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 correctness of almost all the correctness conditions in our tests, 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 describe this tool in more detail next.

III. AN EDITOR FOR INVARIANT BASED PROGRAMS

We start with a brief description of the syntax used to represent invariant based programmings in a textual form. We refer to this representation as *situation analysis* [2], [7]. In the textual representation the rectangles of the diagrams are called *situations* or *regions*. 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 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 [7]. 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

prefixed by PRE. The postcondition is the conjunction of lines beginning with POST. Variables are declared on lines marked by var and transitions on lines beginning with [] followed by a condition and \rightarrow . The notation for transitions is inspired by Dijkstra’s notation for selection in the guarded commands language [5]. A textual representation of the GCD program:

```
procedure GCD ( m, n: Int; result r: Int )
  PRE m > 0  $\wedge$  n > 0
  PRE m = m0  $\wedge$  n = n0
  POST gcd(m0, n0) = r
  [] m  $\geq$  n  $\rightarrow$  LOOP
  [] m < n  $\rightarrow$  m, n := n, m; LOOP
  region LOOP [ m ]
    • m  $\geq$  n > 0
    • gcd(m0, n0) = gcd(m, n)
    [] m = n  $\rightarrow$  r := m; POST
    [] m > n  $\wedge$  m - n  $\geq$  n  $\rightarrow$  m := m - n; LOOP
    [] m > n  $\wedge$  m - n < n  $\rightarrow$  m, n := n, m - n; LOOP
```

Our tool for checking programs was built as part of MathEdit [14]. MathEdit is a text editor for writing mathematical papers, implemented in Python [15]. 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 are produced by applying the *wp* function to the program statements in the transitions, as described in [1].

MathEdit uses Simplify [13] 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 , \wedge 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.¹ Some operators cannot be translated, for instance operators that introduce local scope (except \forall and \exists).

The logical step after parsing, semantical checks and proof of correctness is to compile the program into an executable. Our tool is capable of compiling programs into Python code. The compilation is a straight-forward translation.

IV. 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

¹Variables 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 $f : B \rightarrow B$.

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.

A. Binary Search

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

1) *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 the pre- and postcondition for the procedure. Binary search can be specified as follows:

```

procedure BSearch ( const a: Int[0..N]; const x: Int; result n: Int )
  PRE (∀i : Int • 0 < i < N ⇒ a[i - 1] ≤ a[i])
  POST (∀i : Int • 0 ≤ i ≤ n ⇒ a[i] ≤ x)
  POST (∀i : Int • n < i < N ⇒ x < a[i])

```

Before tackling the problem of how to get from PRE to POST, it is worth thinking twice about what problem is to 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 $a[0] \leq x < a[N - 1]$, i.e. n has to lie in $[0, N)$. Using transitivity of \leq and $<$ we can then define the postcondition as follows:

```

POST 0 ≤ n < N - 1
POST a[n] ≤ x < a[n + 1]

```

As Kaldewaij points out in [16], binary search doesn't need to assume a sorted list in order to reach this postcondition. 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] ≤ 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.

2) *Implementation*: Using the simpler pre- and postcondition, a first approximation of the BSearch might look something like the program shown below.

```

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

```

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 asking the tool to verify this implementation.

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

```

Condition: Case ⊤ in initial transitions (BSearch)
NB: This was proved false by a validity checker.
Assumptions:
  1 < N
  a[0] ≤ x < a[N - 1]
ImPLY:
  N < N
  a[0] ≤ x < a[N]

```

The first condition 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:

```

Condition: Case m - n = 1 in SPLIT (BSearch)
Assumptions:
  a[n] ≤ x < a[m]
  n < m
  m - n ≥ 0
  0 ≤ N
  m - n = 1
ImPLY:
  0 ≤ n < N
  0 ≤ n + 1 < N
  0 ≤ 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 $0 \leq n$ and $n < N - 1$. Rather than writing $n < N - 1$, we will write $m < N$ since $n < m$. 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 below.

```

procedure BSearch ( const a: Int[0..N]; const x: Int; result n: Int )
  PRE 1 < N
  PRE a[0] ≤ x < a[N - 1]
  POST 0 ≤ n < N - 1
  POST a[n] ≤ x < a[n + 1]
  var m, n, k : Int
  [] T → n, m := 0, N - 1; SPLIT
  region SPLIT [ m - n ]
    • a[n] ≤ x < a[m]
    • 0 ≤ n < 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

```

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 (BSearch)

Assumptions:

$$a[n] \leq x < a[m]$$

$$0 \leq n < m < N$$

$$m - n \geq 0$$

$$0 \leq N$$

$$V = m - n$$

$$m - n > 0$$

$$a[(m + n) \text{ div } 2] \leq x$$

Imply:

$$m - (m + n) \text{ div } 2 < V$$

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

4) *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

$$A_1 \wedge A_2 \wedge \dots \wedge A_m \Rightarrow B_1 \wedge B_2 \wedge \dots \wedge B_n$$

such conditions are split by distributing \Rightarrow over \wedge

$$A_1 \wedge A_2 \wedge \dots \wedge A_m \Rightarrow B_1$$

$$A_1 \wedge A_2 \wedge \dots \wedge A_m \Rightarrow B_2$$

⋮

$$A_1 \wedge A_2 \wedge \dots \wedge A_m \Rightarrow B_n$$

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.

B. 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.

1) *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 0 < N
  PRE 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 < N
  POST permutation(a0, a)

```

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 < 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 < 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.

$$x < a[m..n] = (\forall i : \text{Int} \bullet m \leq i < n \Rightarrow x < a[i])$$

$$a[m..n] \leq x = (\forall i : \text{Int} \bullet m \leq i < n \Rightarrow a[i] \leq x)$$

$$a[m..n] = b[m..n] = (\forall i : \text{Int} \bullet m \leq i < n \Rightarrow a[i] = b[i])$$

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 < n ≤ N
  PRE m = m0 ∧ n = n0
  PRE permutation(a0, a)
  POST permutation(a0, a)
  POST a[m0..k + 1] ≤ a[k] ∧ a[k] < a[k + 1..n0]
  POST a0[0..m0] = a[0..m0] ∧ a0[n0..N] = a[n0..N]
  POST m0 ≤ k < n0

```

The abbreviations may not seem easier to read, but when they start recurring, they are easier to pick out and understand.

a) *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 $a[m_0..m] \leq a[k] \wedge a[k] < a[n..n_0]$ whilst making progress towards a situation where $m = n$. This presentation leads to following initial transition and region if we take $k = m$:

```
[] T → LOOP
region LOOP [ n - m ]
  • a[m_0..m] ≤ a[m]
  • a[m] < a[n..n_0]
  • m_0 ≤ m < n ≤ n_0
  [] n - m ≤ 1 → k := m; POST
  [] n - m > 1 → ...
```

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 *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 → 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 $a[m_0..m] \leq a[m]$ and $a[m] < a[n..n_0]$. By adding the following assertion to LOOP, we get rid of the unproved conditions:

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

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

```
Condition: Case n - m ≤ 1 in LOOP (Partition)
Assumptions:
  a[m_0..m] ≤ a[m]
  a[m] < a[n..n_0]
  0 ≤ m_0 ≤ m < n ≤ n_0 ≤ N
  0 ≤ n - m
  n - m ≤ 1
ImPLY:
  permutation(a_0, a)
  a_0[0..m_0] = a[0..m_0]
  a_0[n_0..N] = 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

- $permutation(a_0, a)$
- $a_0[0..m_0] = a[0..m_0]$
- $a_0[n_0..N] = a[n_0..N]$

It remains to replace *magic* with executable program statements that give a transition back to LOOP. After some sketches on paper one might suggest something similar to the following:

```
if a[m + 1] < a[m] → Swap(a, m, m + 1);
                    m := m + 1;
                    LOOP
[] a[m + 1] > a[m] → Swap(a, m + 1, n);
                    n := n - 1;
                    LOOP
[] a[m + 1] = a[m] → m := m + 1;
                    LOOP
```

We have assumed an implementation of Swap. The definition of Swap is given below.

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 $n - m > 1$ in LOOP (Partition)

Assumptions:

```
a[m_0..m] ≤ a[m]
a[m] < a[n..n_0]
permutation(a_0, a)
a_0[0..m_0] = a[0..m_0]
a_0[n_0..N] = a[n_0..N]
0 ≤ m_0 ≤ m < n ≤ n_0 ≤ N
0 ≤ n - m
n - m > 1
```

ImPLY:

Assumptions:

```
a[m + 1] > a[m]
permutation(a, a_2)
a[m + 1] = a_2[n]
a[n] = a_2[m + 1]
a[0..N] = a_2[0..N] except m + 1, n
```

ImPLY:

```
a_2[m] < a_2[n - 1..n_0]
a_0[n_0..N] = a_2[n_0..N]
```

Assumptions:

```
a[m + 1] > a[m]
```

ImPLY:

```
n < N
```

We can read that the validity checker is unable to prove the case $a[m + 1] > a[m]$ correct. It cannot prove that the segment $a[n_0..N]$ satisfies the assertions. It also fails to prove that $n < N$ for the precondition of Swap. Hence the condition tells us that there might be something wrong with the call to Swap. By looking closer at it we can see that n ought to be replaced by $n - 1$ in $Swap(a, m + 1, n)$.

If we ask our tool to verify the implementation after correcting the mistake we get assured that Partition is correct with respect to its pre- and postcondition.

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

```
procedure Swap ( valres a: Int[0..N]; const m, n: Int )
PRE 0 ≤ m < N ∧ 0 ≤ n < N
PRE permutation(a_0, a)
POST permutation(a_0, a)
POST (a_0[m] = a[n]) ∧ (a_0[n] = a[m])
POST a_0[0..N] = a[0..N] except m, n
[] T → a := a[m ↦ a[n]][n ↦ a[m]]; POST
```

$a_0[0..N) = a[0..N)$ except m, n is an abbreviation of
 $(\forall i : \text{Int} \bullet i \neq n \wedge i \neq m \wedge 0 \leq i < N \Rightarrow a_0[i] = a[i])$

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 of `Swap`. 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:

$$\begin{aligned} \top &\Rightarrow \text{partition}(a, a) \\ \text{partition}(a, b) &\Rightarrow \text{partition}(b, a) \\ \text{partition}(a, b) \wedge \text{partition}(b, c) &\Rightarrow \text{partition}(a, c) \end{aligned}$$

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:

$$\begin{aligned} \square \top &\rightarrow a := a[m \mapsto a[n]][n \mapsto a[m]]; \\ &\text{[permutation}(a_0, a)]; \\ &\text{POST} \end{aligned}$$

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

C. Quick Sort

Quick sort [17] 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.

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

```
procedure QSort ( valres a: Int[0..N); const m, n: Int )
  PRE 0 ≤ m ≤ n ≤ N
  PRE permutation(a0, a)
  POST permutation(a0, a)
  POST sorted a[m..n)
  POST a0[0..m) = a[0..m) ∧ a0[n..N) = a[n..N)
```

where *sorted* $a[m..n)$ is an abbreviation of

$$(\forall i : \text{Int} \bullet m < i < n \Rightarrow a[i-1] \leq a[i])$$

2) *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:

```
var p : Int
□ n - m ≤ 1 → POST
□ n - m > 1 →
  Partition(a, m, n, p);
  QSort(a, m, p);
  QSort(a, p + 1, n);
  POST
```

Before we can check the correctness of `QSort` 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:

```
procedure QSort ( 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 two assertions just before `POST`:

$$\begin{aligned} &\{ 0 < p < N \Rightarrow a[p-1] \leq a[p] \}; \\ &\{ 0 < p+1 < N \Rightarrow a[p] \leq a[p+1] \}; \\ &\text{POST} \end{aligned}$$

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 these assertions. Actually the assertions cannot be proved without a definition of *permutation*. Hence we cannot expect our tool to prove it. The result is still useful since the problem has now been pinpointed to a small proof that can be done by paper and pen or in a theorem prover.

V. 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 them 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 verified.

A problem with developing programs using invariant based programming is that it requires programmers to have a good understanding of logical reasoning. This fact does not make our approach impractical, but merely 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` [18] and `PVS` [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 support invariant based programming.

REFERENCES

- [1] R.-J. Back, "Invariant based programming revisited," TUCS (Turku Centre for Computer Science), Turku, Finland, Tech. Rep. 661, 2005.
- [2] —, "Program construction by situation analysis," Computing Centre, University of Helsinki, Helsinki, Finland, Research Report 6, 1978.
- [3] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT*, vol. 8, pp. 174–186, 1968.
- [4] —, "Notes on structured programming," in *Structured Programming*, O.-J. Dahl, C. A. R. Hoare, and E. W. Dijkstra, Eds., Academic Press, New York, 1972.
- [5] —, *A Discipline of Programming*. Prentice-Hall, 1976.
- [6] J. C. Reynolds, "Programming with transition diagrams," in *Programming Methodology*. Berlin: Springer Verlag, 1978.
- [7] R.-J. Back, "Exception handling with multi-exit statements," in *6th Fachtagung Programmiersprachen und Programmentwicklungen*, ser. Informatik Fachberichte, H. J. Hoffmann, Ed., vol. 25. Darmstadt: Springer-Verlag, 1980, pp. 71–82.
- [8] —, "Invariant based programs and their correctness," in *Automatic Program Construction Techniques*, W. Biermann, G. Guiho, and Y. Kodratoff, Eds., no. 223-242. MacMillan Publishing Company, 1983.
- [9] M. H. Van Emden, "Programming with verification conditions," *IEEE Transactions on Software Engineering*, vol. SE-5, 1979.
- [10] K. R. M. Leino and G. Nelson, "An extended static checker for modula-3," in *Proceedings of the 7th International Conference on Compiler Construction*, ser. Lecture Notes in computer Science, vol. 1383, 1998, pp. 302–305.
- [11] D. L. Detslefs, K. R. M. Leino, G. Nelson, and B. Saxe, James, "Extended static checking," Compaq Systems Research Center, Tech. Rep. 159, 1998.
- [12] G. Nelson, "Techniques for program verification," Ph.D. dissertation, Stanford University, 1980.
- [13] D. Detslefs, G. Nelson, and J. Saxe, "Simplify: A theorem prover for program checking," HP Labs, Palo Alto, CA, Tech. Rep. HPL-2003-148, 2003.
- [14] R.-J. Back, V. Bos, and J. Eriksson, "Tool support for structural calculational proofs," technical report to be published in 2005, TUCS (Turku Centre for Computer Science), Turku, Finland.
- [15] G. Van Rossum and F. L. J. Drake, *The Python Tutorial — An Introduction to Python*. Network Theory Ltd., 2003.
- [16] A. Kaldewij, *Programming: The Derivation of Algorithms*. Prentice Hall Europe, 1990.
- [17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.
- [18] M. J. C. Gordon and T. F. Melham, *Introduction to HOL, A Theorem Proving Environment for Higher Order Logic*. Cambridge: Cambridge University Press, 1993.
- [19] S. Owre, J. M. Rushby, and N. Shankar, "Pvs: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Saratoga, NY: Springer-Verlag, 1992, pp. 748–752.