The Modula-2 Proving System MOPS

Th. Kaiser, B. Fischer, W. Struckmann

Informatik-Bericht Nr. 2000-01
Juni 2000

Abteilung Programmierung
Technische Universität Braunschweig
Gaußstraße 11
D-38092 Braunschweig/Germany

The Modula-2 Proving System MOPS

Th. Kaiser, B. Fischer, W. Struckmann
Institut für Software
Abteilung Programmierung
Technische Universität Braunschweig
Gaußstraße 11
D-38092 Braunschweig/Germany
`struck@ips.cs.tu-bs.de`

Abstract

In this report we describe the MOdula-2 Proving System MOPS. It is a Hoare-calculus based program verification system for a large subset of the programming language Modula-2 which uses VDM-SL as specification language. The main goal of MOPS is to demonstrate the feasibility and viability of a Hoare-style verification system for a real imperative programming language, including pointers, arrays, and other data structures. MOPS also provides support for the modular and partial verification of large systems.

We demonstrate MOPS with some example verifications. While the first two examples are rather small, the third one consists of a series of increasingly sophisticated quicksort-versions which include the median-of-three pivot selection strategy as well as the use of selection sort and bubblesort for small subarrays.

Contents

1. Introduction **2**

2. Calculus **3**

3. Specification and verification **4**
- 3.1. Specification of elementary control structures 4
- 3.2. Specification of array operations 5
- 3.3. Specification of record operations 6
- 3.4. Specification of pointer operations 7
- 3.5. Specification of functions and procedures 8

4. Modular verification **10**

5. Usage of the MOPS-system **13**

6. Examples **15**
- 6.1. Gaussian sum formula 15
- 6.2. Sorting algorithms 16
  - 6.2.1. Bubblesort 16
  - 6.2.2. Quicksort: base algorithm 17
  - 6.2.3. Quicksort: variation 1 25
  - 6.2.4. Quicksort: variation 2 26
  - 6.2.5. Quicksort: variation 3 28
- 6.3. Compression and decompression: The LZW algorithm 28

7. Conclusions **28**

References **28**

A. Gaussian sum formula **31**

B. Sorting algorithms **32**
- B.1. Bubblesort 32
  - B.1.1. The specified Modula-2 program 32
  - B.1.2. Proof obligations 33
- B.2. Quicksort: base algorithm 36
  - B.2.1. The Modula-2 program 36
  - B.2.2. The specified program 38
  - B.2.3. Proof obligations 44
- B.3. Quicksort: variation 1 52
- B.4. Quicksort: variation 2 53
- B.5. Quicksort: variation 3 53

1. Introduction

Almost all computer programs contain errors, at least initially. The traditional approach to discover these errors is testing. However, since testing can only be used to show the presence of errors but not their *absence*, other approaches as *program verification* are pursuit. Program verification is an exact, formal method to prove for all possible inputs the consistency between the specification of a program and its implementation. It is obviously closely related to the formal specification of software: the correctness proof for a program is done relative to its formal specification which should thus capture the informal requirements sufficiently.

A *verification system* automates parts of the verification task. The architecture of verification systems usually comprises two different tiers, a predicate transformer or *verification condition generator*, and a prover. The verification condition generator takes the program and the specification and computes a set of logical expressions called *proof obligations*. These are then proven or *discharged*, either automatically, by the prover, or manually, by the software engineer. If all obligations are discharged the program is proven correct with respect to the specification (assuming that the underlying calculus is sound). However, the failure to discharge an obligation does not always mean that the program contains an error. It may also indicate that the specification is incomplete or not adequate, or that the prover is too weak.

The reason for the two-tiered architecture is purely pragmatic. Any specification language which is expressive enough to capture "interesting" requirements (and thus to describe "interesting" programs) is undecidable. Hence, any prover is too weak for a fully automatic system. In contrast to that, the generation of verification conditions is decidable and a fully automatic verification condition generator can be implemented, even for real programming languages.

The *Modula Proving System* (MOPS) is a Hoare-calculus based program verification system for a large subset of the programming language Modula-2 which uses VDM-SL [7] as specification language. The main goal of MOPS is to demonstrate the feasibility and viability of a Hoare-style verification system for a real imperative programming language, including pointers, arrays, and other data structures. MOPS also provides support for the modular and partial verification of large systems and includes hooks for specification-based code reuse systems as for example NORA/HAMMR [4]. Finally, MOPS demonstrates the combination of a verification system with an etablished specification language which exists outside the verification system itself.

MOPS is built according to the two-tiered architecture outlined above and comprises a weakest precondition predicate transformer and a rather weak rewrite-based prover; however, stronger off-the-shelf provers can be incorporated relatively easy. The predicate transformer used in MOPS supports only proofs of partial correctness, i.e., reasoning about termination cannot be done within MOPS. However, this allows us to use a simpler calculus and also yields simpler proof obligations.

MOPS essentially follows the more traditional approach to verify programs after the implementation is completed instead of developing proof and program hand-in-hand, as for example advocated by the KIV-system [14]. However, we believe that the traditional approach is better suited for the incremental or even partial verification of large systems as the users can easily restrict the verification to the critical parts of a system.

The current version of MOPS supports almost the entire Modula-2 programming language as defined in [17], including pointers and data structures. The only language constructs not yet supported are variant record types, procedure types, and procedures as parameters, i.e., higher-order procedures cannot be verified. The verification of REAL-arithmetics is idealized and ignores possible rounding errors. Modula-2 also relies heavily on the use of standard libraries, e.g., for input/output, systems programming, and parallel programming. MOPS does not provide specific support for most of these modules but programs built on top of them can be verified as usual (except for input/output) after these modules have been re-specified using the modular verification techniques described in section 4.

This report describes the program verification system MOPS. In Section 2 the main ideas of the underlying calculus of MOPS are introduced. Section 3 explains how the different constructs of Modula-2 are specified within MOPS. Section 4 deals with the concept of modular verification. Section 5 is a short user's guide. Then, in section 6 some programs and their verifications are presented. While the first two examples are rather small, the third one—quicksort—consists of a series of increasingly sophisticated versions which include the median-of-three pivot selection strategy as well as the use of discrete (selection) sort and bubblesort for small subarrays. As a final example the well known LZW compression and decompression algorithms [18, 19, 16] are given. To be precise, the description here is rather short, the full version can be found in the literature. These collection demonstrate, we hope, that MOPS is suitable to verify "production quality" library components. The appendix contains the source code for some of these programs.

2. Calculus

MOPS is built upon the Hoare-calculus. Its theoretical foundations and the fundamental verification algorithms based on this calculus can be found in, e.g., [1, 2, 6]. We extended these foundations into a calculus for the programming language Modula-2 by adding further proof rules and extending the underlying logic. Adding new statements to the language means adding new proof rules to the calculus. This is relatively straightforward and as long as the new rules are sound and the statements are disjoint from the core, the extended calculus remains obviously sound. Adding data types, however, extends the underlying logic and can easily compromise its soundness. This problem has been dealt with in the literature, e.g., [3, 10].

The starting point for the axioms and proof rules for the verification of arrays, records and pointers has been the proof system given in [10]. For MOPS, this system was extended to support explicit memory deallocation via the `DISPOSE`-procedure in the Modula-2 system module. Obviously, pointers introduce the same *aliasing problem* as arrays, i.e., a memory location can be addressed by different names. The main idea in [10] is to treat all pointers of a particular type as a single dynamic array and thus to handle pointer aliasing with the same mechanism as array aliasing. This approach, however, critically relies on Modula-2's pointer discipline which guarantees that two pointers refer to the same memory location only if one of them has—directly or indirectly—been assigned to the other. It can thus not be applied to languages as C which allow pointer arithmetics. The complete axioms and proof rules for this approach are given in [8].

Hoare-style calculi are usually defined over the classical, two-valued predicate calculus. This implies that expressions are always assumed to be defined which in turn requires all semantic functions to be total. Since MOPS uses VDM-SL as specification language, it is natural to base the calculus on the logic LPF (Logic of Partial Functions) underlying VDM-SL. This does not affect the verification condition generator; however, the proof obligations are now LPF-formulae. Semantically, this provides an encapsulation of all partiality reasoning within the proof theory for LPF or an off-the-shelf translation from LPF to the classical predicate calculus. Moreover, partial correctness becomes a stronger result than in the classical case as it implies the absence of run-time errors caused by application of partial functions to arguments outside their domain, e.g., division by zero.

Intuitively, our calculus should be sound and relatively complete with respect to LPF; we expect the formal proofs to be straightforward adaptations from the classical proofs in the literature. Obviously, however, the calculus is not relatively complete with respect to the classical predicate logic.

3. Specification and verification

MOPS supports the verification of arbitrary program segments and not only, e.g., procedures or modules. This precludes considering the implementation as the final refinement of a specification module as for example done in KIV but requires a direct embedding of the VDM-SL specification into the Modula-2 code. Syntactically, this is achieved by enclosing the VDM-SL expressions within formal comments (`(*{` and `}*)`) such that the annotated program can still be compiled and executed by any Modula-2 compiler. MOPS thus assumes the syntactic correctness of the Modula-2 program. Since the VDM-SL specification can be extracted from the annotated program automatically and shown consistent using external tools, MOPS also assumes the syntactic correctness and internal consistency of the VDM-SL specification. Such embedding approaches date back at least to the ANNA-system [9] and have also been used in the specification languages in the Larch-tradition, e.g., in the Penelope-system [5].

3.1. Specification of elementary control structures

The specification and verification of statement sequences, `if`-, `case`-, and the various loop-statements is rather straightforward. Note, there is no `goto`-statement in Modula-2.

MOPS uses `entry`/`exit`-tags as shown below to mark the verification segments; these can be nested to break large proofs into manageable pieces. Loop invariants, which must be provided as usual in Hoare-style calculi, and additional `assert`-tags are used to aid the proof construction. Joint scoping allows the specification to refer to program variables but not vice versa.

```
...
(*{ entry sum_loop
     pre  sum = 0
     post sum = n * (n+1) div 2       }*)
(*{ loopinv sum = ((i - 1) * i) div 2 }*)
FOR i := 1 TO n DO
    sum := sum + i;
```

```
END;
(*{ exit sum_loop }*)
...
```

Verification segments also provide convenient hooks for specification-based retrieval as the `pre/post`-pair already comprises the crucial part of a retrieval query. By changing the `entry`-tag into the VDM-SL operation signature `sum_loop(n:int) ext rw sum:int` a retrieval system as NORA/HAMMR [4] (which also uses VDM-SL as specification language) can extract a full query and search a library for semantically matching, verified components. This allows a smooth integration of reuse without compromising program correctness, thus reducing the overall verification effort.

The main problem of embedding an existing specification language into a verification system (as opposed to defining a specialized behavioral interface specification language) is to define a suitable translation between the constructs of the implementation and specification languages. Fortunately, VDM-SL's meta-language heritage makes this task easier and most constructs (e.g., base types) can be mapped in a rather straightforward way.

3.2. Specification of array operations

In their paper *Verification of Array, Record and Pointer Operations in Pascal* [10] Luckham and Suzuki discuss an extension of the Hoare calculus to handle complex data types. Obviously, Hoare's assignment axiom is not sufficient for this case.

The main idea to handle structured data types is to treat them as a unit. To change an element of an array or a records means to change the entire array or record. For example, the assignment of an specific element $A[i]$ of an array has no consequence to the element $A[j]$ when using the assignment axiom of Hoare because these elements are syntactically different. Following Luckham und Suzuki, the assignment of $A[i]$ changes the array as a whole. In fact, $A[j]$ may also be changed in case $i = j$. For a further discussion of the extension of the assignment axiom see [8].

An array type of Modula-2 is represented in VDM-SL by a sequence type. Multi-dimensional arrays are modeled by sequences of sequences. A sequence is a finite map whose domain is a subset of the natural numbers. This is described by the following type invariant:

1.0 $Sequence = \mathbb{N}_1 \xrightarrow{m} X$

.1 $\mathsf{inv}\ s \triangleq \exists\, n \in \mathbb{N} \cdot \mathsf{dom}\ s = \{1, \ldots, n\}$

An immediate consequence is the precondition of a selection using an index i of a sequence $s \in X^*$:

$$s \in X^* \land 1 \le i \le \text{len } s \;\Rightarrow\; s(i) \in X$$

Transferring this precondition to an array of Modula-2 means that every index i used to select $A[i]$ in an array A must be an element of the domain of the array. In VDM-SL the selection of an element i of a sequence A is written as $A(i)$.

Example 1: In the following specified Modula-2 program the values of the variables `i` and `j` are undefined. Thus, the execution of this program will lead to a runtime error.

```
MODULE ArrayTest;

VAR a    : ARRAY[1..10],[2..3] OF CARDINAL;
    i, j : CARDINAL;

BEGIN

    (*{ entry arrayBsp post a(i)(j) = 4 }*)

    a[i,j] := 4;

    (*{ exit arrayBsp }*)

END ArrayTest.
```

Using the above type invariant the MOPS-system will generate one proof obligation:

```
Proof obligation in lines 8:9-9:46:
i >= 1 and i <= 10 and  j >= 2 and j <= 3
```

Thus, the program cannot be proven correct w.r.t. this specification because it cannot be guaranteed that at the beginning of the sequence the value of `i` is in the range `1..10` and the value of `j` is in the range `2..3`. □

3.3. Specification of record operations

A record type of Modula-2 is modeled in VDM-SL by a composition type. Therefore,

```
TYPE T = RECORD
           a1 : T1;
           a2 : T2;
             ...
           an : Tn;
         END;
```

is represented in VDM-SL by

```
T::a1 : T1
   a2 : T2
     ...
   an : Tn
END;
```

The selection of a component `a1` of a record `t` is written as `t.a1` both in Modula-2 and VDM-SL.

Example 2: The proof obligations generated during the verification of the following specified Modula-2 program are all reduced to `true` applying the reduction rules. Therefore, in this example the verification is carried out completely automatically.

```
MODULE RecordTest;

VAR x : RECORD
            y : RECORD
                    a: ARRAY[1..5] OF CARDINAL;
                END;
            z : CARDINAL;
        END;

BEGIN

    (*{ entry mix pre  true
                  post x.y.a(1) = 0 }*)

    x.z        := 1;
    x.y.a[x.z] := 0;

    (*{ exit mix }*)

END RecordTest.
```

□

3.4. Specification of pointer operations

The problem of the Hoare calculus handling different names for the same variable, *aliasing*, arises also when using pointers. The main idea here is to model the pointers as a dynamic array. The *reference class* P#T contains all pointers of the type "pointer to T." For a further discussion of the extension of the assignment axiom see [8].

To dereference a pointer Q of a reference class D the construct $D \subset Q \supset$ is introduced. The allocation of memory extends the reference class which is described by $D \cup \{Q\}$. The reference predicate *PointerTo(X, D)* has been defined to express that a pointer X is an element of a reference class D.

In MOPS only one reference class named *POINTER* is available. Therefore, to dereference a pointer x one uses $POINTER(x)$. To express the extension of the reference class by the new element x there is the construct $Add(POINTER, x)$.

Example 3: The specified Modula-2 program

```
MODULE PointerTest;

  FROM Storage IMPORT New, Dispose;
  FROM InOut   IMPORT WriteString, WriteLn;

  VAR x : POINTER TO CARDINAL;

BEGIN

  (*{ entry NewTest pre  true
                    post PointerTo(x, POINTER) }*)

  New (x);

  (*{ exit NewTest }*);


  (*{ entry AssignTest pre  PointerTo(x, POINTER)
                       post PointerTo(x, POINTER) and
                            POINTER(x) = 4            }*)

  x^ := 4;

  (*{ exit AssignTest }*);


  (*{ entry DisposeTest pre      PointerTo(x, POINTER)
                        post not PointerTo(x, POINTER) }*)

  Dispose (x);

  (*{ exit DisposeTest }*);

END PointerTest.
```

is verified completely by MOPS. □

3.5. Specification of functions and procedures

In VDML-SL functions and operations can be specified. These specifications may be implicit by giving a pre- and a postcondition or explicit by describing an algorithm. In both cases the precondition is optional. Functions compute their result using their arguments. These arguments cannot be changed during the computation. Operations cause a change in the global state by altering the value of external variables. These external variables have to be declared in a state definition.

In Modula-2 there is a `PROCEDURE` construct which is a combination of the VDM-SL constructs function and operation. A VDM-SL function corresponds to a Modula-2 procedure with a result and call-by-value-parameters. A function is not allowed to have side effect on global variables.

So, at first sight it looks easy to establish the connection between procedures in Modula-2 and function and procedures in VDM-SL. However, things are slightly more complicated.

A Modula-2 `PROCEDURE` with a return value and call-by-value-parameters only but without side effects can be specified via a VDM-SL function.

> **Example 4:** In the following fragment the Modula-2 procedure `inc` is specified by the explicit definition of a VDM-SL function:
>
> ```
> (*{ functions
>         inc : nat -> nat
>         inc (n) == n + 1  }*)
>
> PROCEDURE inc (x : CARDINAL) : CARDINAL;
> BEGIN
>     RETURN x + 1
> END inc;
> ```
>
> □

A procedure without call-by-reference-parameters but with side effects on global variables corresponds to an operation in VDM-SL. The global variables of a Modula-2 program and their values implicitly form a state.

> **Example 5:** In the following fragment the procedure `setX` has a side effect on the global variable `x`. The definition of an operation specifies this effect:
>
> ```
>     VAR x : CARDINAL;
>
>     (*{ operations
>            setX ()
>            ext wr x
>            post   x = 4 }*)
>
>     PROCEDURE setX ();
>     BEGIN
>         x := 4;
>     END setX;
> ```
>
> □

Call-by-reference parameters have no direct correspondence in VDM-SL; they require generating a (local) state containing the call-by-reference-parameters. Therefore, to change the value of a parameter means to change a state variable which can be specified using an operation.

> **Example 6:** The procedure `sum` has the call-by-reference-parameter `y`:

```
PROCEDURE sum (a, b : CARDINAL; VAR y : CARDINAL);
BEGIN
    y := a + b
END sum;
```

Viewing `y` as a state variable the procedure can be specified as described above:

```
PROCEDURE sum (a, b : CARDINAL; VAR y : CARDINAL);
(*{ post y = a + b }*)
```

□

In VDM-SL state definitions cannot be nested. Also, the definition of an operation is only meaningful w.r.t. a state definition. A possible solution to this problem is the specification of procedures with call-by-reference-parameters using the body of an operation. This specification has to follow immediately the Modula-2 declaration of the procedure. Thus, it is not possible to separate the specification from the declaration of the procedure.

4. Modular verification

Large systems are inevitably split into several separate modules and MOPS supports the verification of such modular systems. Procedure specifications can be separated from their corresponding implementations by including them into the definition modules only. The implementations are then verified against their definitions. Client modules which import a specified procedure automatically import the associated function specification and thus need to verify only the particular call. Thus, the verification can be modularized. Figure 1 illustrates this concept.

Example 7: The procedure `inc` is declared in the definition module and specified by a VDM-SL function:

```
DEFINITION MODULE Increment;
PROCEDURE inc (x: CARDINAL) : CARDINAL;

(*{ functions
        inc : nat -> nat
        inc (n) == n + 1   }*)

END Increment.
```

`inc` is programmed in the corresponding implementation module:

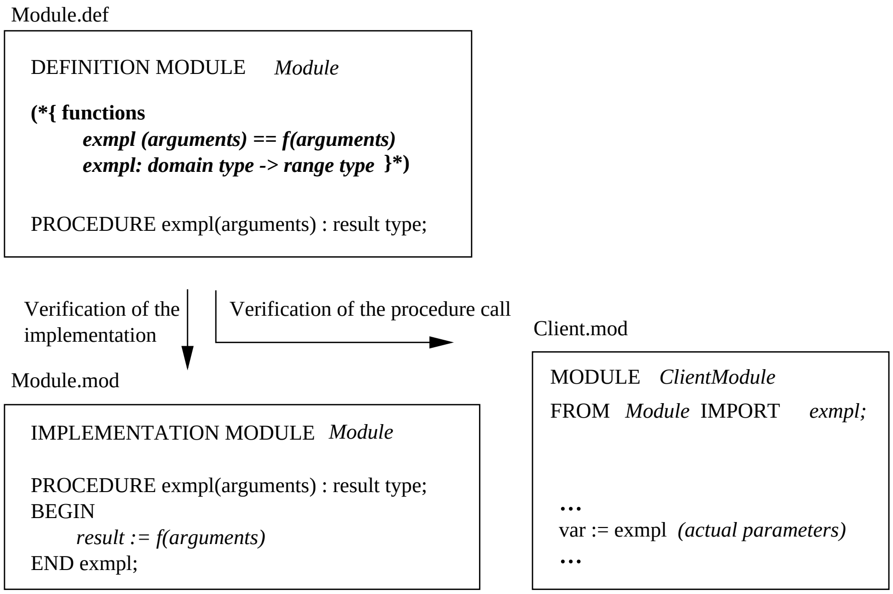

Figure 1: *Modular Verification*

```
IMPLEMENTATION MODULE Increment;
   PROCEDURE inc (x : CARDINAL) : CARDINAL;
   BEGIN
      RETURN x + 1;
   END inc;

BEGIN
END Increment.
```

The verification of the implementation module by the MOPS-system generates the following verification conditions which are completely reduced to `true` by the rewriting rules. Note that `result` is an auxiliary variable which holds the result of the `return`-statement.

```
Proof obligation in lines 3:4-5:7:
 false => true

Proof obligation in lines 3:4-5:7:
 true => 1 + x = 1 + x

Proof obligation in lines 3:4-5:7:
 1 + x = result => true
```

The module `Client` imports the procedure `inc`:

```
MODULE Client;

FROM Increment IMPORT inc;

VAR y : CARDINAL;

BEGIN
   y := 1;

   (*{ entry main1 pre  y = 1
                   post y = 2 }*)
   y := inc (y);

   (*{ exit main1 }*)

END Client.
```

The verification of the function call in the module `Client.mod` can be done independent of the verification of the implementation of `inc` in the module `Increment.mod`. The verification condition

```
Proof obligation in lines 10:8-11:36:
 1 = y => inc(y) = 2
```

is generated. Its validity is immediate. □

If a procedure contains no call-by-reference parameters, its specification can be separated entirely from the Modula-2 declaration, even beyond the file boundary of the definition module, and moved into a completely seperated specification file containing a pure VDM-SL module. The correspondence of these files is guaranteed by extending the Modula-2 naming conventions (see figure 2). This allows a subsequent specification of existing modules, e.g., standard library modules, without any changes to the definition modules. This is required for the timestamp-based module consistency mechanism employed by most Modula-2 compilers.

In MOPS, a Modula-2 client module can import arbitrary objects from arbitrary other modules. In particular, it can also access symbols from pure VDM-SL modules which are not associated with any definition or implementation modules. Hence, VDM-SL can be used as shared language to define theories supporting the verification (see figure 3).

Example 8: In the VDM-SL file *functiondefs* the function `sum` is defined:

```
functions
    sum : nat * nat  -> nat
    sum (a, b) == a + b
```

The function is imported by the specification part of the following Modula-2 program and is used to specify the sequence *main*:

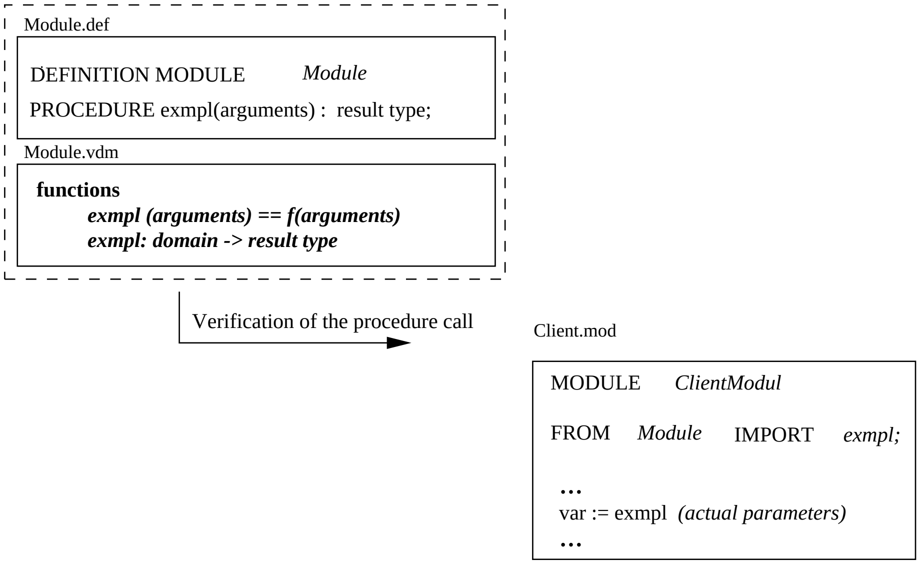

Figure 2: *Subsequent specification*

```
MODULE VDMImport;
(*{ imports
        from functiondefs
             functions  sum }*)

VAR x, y, z : CARDINAL;

BEGIN
     x := 1; y := 2;

     (*{ entry main post z = sum (x,y) }*)

     z := x + y;

     (*{ exit main }*)

END VDMImport.
```

□

5. Usage of the MOPS-system

As we have seen, MOPS is a verification system for programs written in a subset of the programming language Modula-2 and specified in VDM-SL. Given a specified program MOPS will generate all verification conditions needed to prove the partial correctness of the program with respect to the specification. As the VDM-SL expressions are completely embedded as comments, the specified program can be translated by any Modula-2

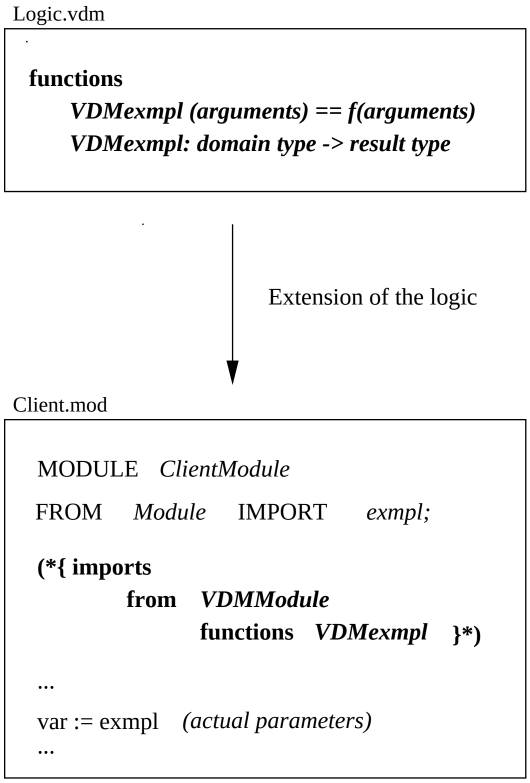

Figure 3: *Extension of the logic by VDM-SL import*

compiler. As already pointed out, the MOPS system assumes the syntactical correctness of the Modula-2 program and the consistency of the VDM-SL specification. The MOPS-system is implemented in the functional programming language SML. [12]

The verification of a specified Modula-2 program is started by the SML function call

```
val conds = MOPS.verify "filename";
```

Then, the verification conditions will be collected in the file *filename.vc* and a protocol of the verification will be written in *filename.proof*. This function call summarises the single steps of the verification as explained below.

The syntax tree of the specified program text will be constructed by the function call

```
val cu = MOPSParser.fparse "filename";
```

The call

```
val all = M2_Elaborate.prepareSymTab cu;
```

generates the Modula-2 as well as the VDM-SL symbol tables. Since the specification of a procedure in the source may be located before the implementation a second pass through the syntax tree is needed:

```
val (cu',mst,vst) = M2_Elaborate2.secondPrepare all;
```

This may alter the syntax tree. The generation of the verification conditions is done by the function call

```
val pol = MOPS_Verify.verify' (cu', mst, vst, aProtocolFilename);
```

Using a few rewrite rules the verification conditions may be simplified by the function call

```
val pol' = map MOPS_Simplify.normalize pol;
```

Finally,

```
map MOPS_Verify.showCond pol';
```

will transform the verification conditions into a text format.

6. Examples

In this section we will show some examples which have been specified and verified successfully with MOPS. First, we have a look at a small program containing just one loop—the Gaussian sum formula. Then we deal with "bubblesort" and four versions of "quicksort." The completely specified programs and the generated and proved verification conditions can be found in [8]. Finally, the LZW compression and decompression algorithms [16, 18, 19] are considered. Of course we cannot go into details here, they are in the references.

6.1. Gaussian sum formula

The Gaussian formula to compute the sum of the first n natural numbers

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

is implemented and specified in the module *SumUpToN* (cf. appendix A). The validity of the verification condition generated during the verification is proved.

At the end of the computation the value of the variable `sum` should be $\sum_{i=1}^{n} i$. Thus, the postcondition of the statement sequence is the Gaussian formula `sum = N * (N + 1) div 2`. The precondition `N >= 0` is redundant because the type of `N` is `CARDINAL`. As the simplification algorithm of MOPS uses no type information this specification is useful for the application of the rewrite rules.

During the i-th run of the loop the value of `sum` is $\sum_{j=1}^{i\text{-}1} j$. Furthermore, `i <= N + 1`. Thus, the loop invariant is formed as the conjunction of these two conditions.

Now we consider the verification conditions generated but not proved by MOPS.

```
Proof obligation in line 16:9-66:
   exists  X_7 : nat &
     1 + X_7 >= i and
     X_7 = N and
     ((i - 1 ) * i) div 2 = sum and
```

```
     X_7 >= i
=> (((1 + i) - 1) * (1 + i)) div 2 = i + sum and
   1 + N >= 1  + i
```

The expression in the end of the implication can be simplified:

$$\begin{aligned} & (((1+i)\text{-}1)*(1+i)) \text{ div } 2 \\ = \quad & (i*(1+i)) \text{ div } 2 \\ = \quad & (i*(i\text{-}1+2)) \text{ div } 2 \\ = \quad & ((i*(i\text{-}1))+2*i) \text{ div } 2 \\ = \quad & (i*(i\text{-}1)) \text{ div } 2+i \end{aligned}$$

Using the precondition

$$((i\text{-}1)*i) \text{ div } 2 = sum$$

this can be simplified to $sum + i = sum + i$. The remaining inequality $1 + N \geq 1 + i$ follows from

$$\text{X\_7} = N \wedge \text{X\_7} \geq i \quad \Rightarrow \quad N \geq i.$$

```
Proof obligation in line 16:9-66:
   exists  X_7 : nat &
     i > X_7 and
     X_7 = N and
     ((i - 1 ) * i) div 2 = sum and
     1 + X_7 >= i
=> ((1  + N) * N) div 2 = sum
```

Because of the precondition $\text{X\_7} = N$ the variable X_7 can be replaced by N in the inequalities $1 + \text{X\_7} \geq i$ and $i > \text{X\_7}$. Because $i \in \mathbb{N}$,

$$N + 1 \geq i > N \ \Rightarrow \ i = N + 1.$$

Replacing i by $N + 1$ in

$$((i\text{-}1)*i) \text{ div } 2 = sum$$

shows the validity of the rest of the implication:

$$(N*(N+1)) \text{ div } 2 = sum$$

6.2. Sorting algorithms

6.2.1. Bubblesort

Now we are going to illustrate these ideas by more complex examples. As a first one the reader should look at the completely specified bubblesort algorithm together with the verification conditions generated by MOPS as given in appendix B.1. We do not comment on this program. Instead we will concentrate on the more interesting quicksort algorithm.

6.2.2. Quicksort: base algorithm

Quicksort divides an array to be sorted in two parts and then sorts both parts recursively. One part contains all elements less then a *pivot element* and the other part all elements greater or equal than this special element. Of course, there is some freedom in choosing the pivot element.

Here, we present the base version and three variants of the quicksort-algorithm, including the median-of-three pivot selection strategy and the use of selection sort and bubblesort for small subarrays. The base algorithm which uses the "middle element" as the pivot is implemented and specified in the procedure *QuickSort*. (cf. appendix B.2)

The quicksort-implementations work on open arrays of element-records and sort by one of the record components. The base version consists of more than 300 lines of Modula-2 code and VDM-SL specification. MOPS generates 23 proof obligations and discharges 14 by plain rewriting. By encapsulation of the variation into separate verification segments, the number of emerging proof obligations for the variants can generally be kept small; however, MOPS does not provide any proof management.

In the specification the predicate *sorted* indicates that an array is sorted. Partitioning the array is realized using a `while`-loop. Starting with the lower and upper bounds of the indices quicksort looks for elements greater or less than the pivot element. This search is implemented using two `while`-loops. These loops terminate because there always exists an element with an index greater than *left* whose *key* component is greater than or equal to the pivot element. This is stated in the *containsElementGEQ* predicate

$$\begin{aligned}\text{containsElementGEQ} \quad &: \quad (\text{seq of Element}) \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{B} \\ \text{containsElementGEQ}(A, i, j, v) \quad &= \quad \exists \ \cdot p \in \mathbb{N} \cdot i \leq p \leq j \wedge A(p).\mathit{key} \geq v\end{aligned}$$

The *containsElementLEQ* predicate expresses an analogous proposition. After these loops the value of the variable *left* is the index of that element whose *key* component is greater than the pivot element and the value of *right* is the index of that element whose *key* component is less than the pivot element. If the partitioning is not yet finished these element will be swapped and the search is continued starting from these positions. The parts whose indices are less than *left* and greater than *right* suffice. The predicate *partitioned*

$$\begin{aligned}\text{partitioned} \quad &: \quad (\text{seq of Element}) \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{N} \to \mathbb{B} \\ \text{partitioned}(A, m, M, l, r, p) \quad &= \quad \forall\, k \in \mathbb{N} \cdot m \leq k < l \Rightarrow A(k).\mathit{key} \leq \mathit{pivot} \wedge \\ & \forall\, i \in \mathbb{N} \cdot r < i \leq M \Rightarrow A(i).\mathit{key} \geq \mathit{pivot}\end{aligned}$$

is therefore an invariant for the outer while loop. Now, we are going to (manually) prove the remaining verification conditions.

Sequence "choose_pivot"

```
Proof obligation in lines 61:13-66:70:
   max >= min and HIGH A >= max and min >= 0
```

```
=> HIGH A >= (max + min) div 2 and (max + min) div 2 >= 0  and
   containsElementGEQ(A,min,max,([A (max + min) div 2]).key) and
   containsElementLEQ(A,min,max,([A (max + min) div 2]).key)
```

The inequalities at the end of the implication follow from

$$max \geq min \wedge min \geq 0 \quad \Rightarrow \quad \frac{max + min}{2} \geq 0$$

and

$$HIGH(A) \geq max \wedge max \geq min \quad \Rightarrow \quad HIGH(A) \geq \frac{max + min}{2}.$$

It is

$$max \geq min \quad \Rightarrow max \geq \frac{max + min}{2} \geq min,$$

therefore using

$$p_1 = p_2 = \frac{max + min}{2}$$

the existential statements in *containsElementGEQ* and *containsElementLEQ* are valid.

Entering the outer loop

```
Proof obligation in lines 61:13-66:70:
   min >= 0   and min = left and
   max >= min and max = right and
   HIGH A >= max and
   containsElementGEQ(A,min,max,pivot) and
   containsElementLEQ(A,min,max,pivot)
=> left >= min   and right   >= min - 1  and
   max  >= right and 1 + max >= left and
   forall  j : nat &
     left > j and j > right => (([ A  j ]).key) = pivot and
   partitioned(A,min,max,left,right,pivot)
```

The inequalities at the right hand side follow from

$$\begin{aligned} min = left &\Rightarrow left \geq min \\ max = right &\Rightarrow max \geq right \\ max \geq min \wedge min = left &\Rightarrow 1 + max \geq left \\ max = right \wedge max \geq min &\Rightarrow right \geq min\text{-}1 \end{aligned}$$

The antecedents of the universal quantified statements in the *partitioned* predicate are all false. Thus, the implications are true:

$$\begin{aligned} left = min &\Rightarrow left > k \wedge k \geq min \equiv false \\ right = max \geq min = left &\Rightarrow left > j \wedge j > right \equiv false \\ right = max &\Rightarrow max \geq i \wedge i > right \equiv false \end{aligned}$$

Sequence "find_elements_to_swap"

```
Proof obligation in lines 110:17-128:70:
   min    >= 0   and max  >= right and
   HIGH A >= max and left >= min   and right > left and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot) and
   forall  j : nat &
     left > j and j > right => (([ A  j ]).key) = pivot and
   partitioned(A,min,max,left,right,pivot)
=> right >= min and max >= left and
   containsElementGEQ (A,left,max,pivot) and
   containsElementLEQ (A,min,right,pivot)
```

The inequalities are consequences of

$$\begin{aligned} max \geq right \wedge right > left \quad &\Rightarrow \quad max \geq left \\ right > left \wedge left \geq min \quad &\Rightarrow \quad right \geq min. \end{aligned}$$

The *containsElement* predicates contain $right > left$ in the antecedent. Because $right > left$ is part of the precondition these predicates are true.

Invariance of the loop invariant of the first inner while-loop

```
Proof obligation in lines 130:17-136:70:
   left   >= min   and min   >= 0  and
   max    >= right and max   >= left and
   HIGH A >= max   and right >= min and
   pivot > (([ A  left ]) . key) and
   containsElementLEQ(A,min,right,pivot)
   containsElementGEQ(A,left,max,pivot) and
   partitioned(A,min,max,left,right,pivot) and
=> max >= 1 + left and 1 + left >= min and
   partitioned(A,min,max,1 + left,right,pivot) and
   containsElementGEQ(A,1 + left,max,pivot)
```

With $left \geq min$, $left + 1 \geq min$ holds. The existential proposition of the *containsElementGEQ* predicate in the end of the implication follows from

$$pivot > A[left].key$$

together with

$$\begin{aligned} & \text{containsElementGEQ}(A, left, max, pivot) \\ \equiv \quad & \exists\, p \cdot \cdot\, max \geq p \geq left \wedge A[p].key \geq pivot. \end{aligned}$$

Furthermore, from these propositions we have $left < max$ and $max \geq left + 1$. Because of

$$\begin{array}{rl} & (\forall k \cdot left > k \wedge k \geq min \Rightarrow pivot > A[k].key) \wedge pivot > A[left].key \\ \Rightarrow & (\forall k \cdot left + 1 > k \wedge k \geq min \Rightarrow pivot > A[k].key) \end{array}$$

the validity of the *partitioned* predicate and the the verification condition are immediate.

Invariance of the loop invariant of the second inner while-loop

```
Proof obligation in lines 154:17-161:69:
   left >= min   and min >= 0  and
   max >= right  and max >= left and
   HIGH A >= max and right >= min and
   (([ A  left ]) . key) >= pivot and
   (([ A  right ]) . key) > pivot and
   containsElementLEQ(A,min,right,pivot)
   containsElementGEQ(A,left,max,pivot) and
   partitioned(A,min,max,left,right,pivot) and
=> right - 1  >= min and max >= right - 1 and
   partitioned(A,min,max,left,right - 1,pivot) and
   containsElementLEQ(A,min,right - 1,pivot)
```

>From $max \geq right$ we deduce $max \geq right\text{-}1$. The existential proposition of the *containsElementLEQ* predicate in the end of the implication is a consequence of

$$A[right].key > pivot$$

and

$$\begin{array}{rl} & \text{containsElementGEQ}(A, left, max, pivot) \\ \equiv & \exists\, p \cdot \cdot\, right \geq p \wedge p \geq min \wedge pivot \geq A[p].key. \end{array}$$

Furthermore, we have $right > min$ and thus $right\text{-}1 \geq min$. Because of

$$\begin{array}{rl} & (\forall i \cdot max \geq i \wedge i > right \Rightarrow A[i].key > pivot) \wedge A[right].key > pivot \\ \Rightarrow & (\forall i \cdot max \geq i \wedge i > right\text{-}1 \Rightarrow A[i].key > pivot) \end{array}$$

the *partitioned* predicate and the verification condition are valid.

Sequence “swap_elements”

```
Proof obligation in lines 174:17-192:70:
   left   >= min   and min   >= 0  and
   max    >= right and max   >= left and
```

```
   HIGH A >= max   and right >= min and
   (([ A  left ]) . key) >= pivot and
   pivot >= (([ A  right ]) . key) and
   containsElementLEQ(A,min,right,pivot)
   containsElementGEQ(A,left,max,pivot) and
   partitioned(A,min,max,left,right,pivot)
=> left > right =>
     1 + max >= left and right >= min - 1 and
     right > left =>
       containsElementGEQ(A,left,max,pivot) and
       containsElementLEQ(A,min,right,pivot) and
     forall  j : nat & left > j and  > right
        => (([ A  j ]) . key) = pivot and
   right >= left =>
     left >= 0       and max >= right - 1  and
     1 + left >= min and 1 + max >= 1  + left and
     HIGH A >= right and HIGH A >= left and
     right - 1  >= min - 1  and   right >= 0  and
     right - 1  > 1  + left =>
       containsElementGEQ(ArrayUpdate
                         (ArrayUpdate A left ([ A  right ]))
                          right ([ A left ]),1 + left,max,pivot) and
       containsElementLEQ(ArrayUpdate
                          (ArrayUpdate A left ([ A  right ]))
                          right ([ A  left ]),min,right - 1,pivot) and
     forall  j : nat &
       1 + left > j and j > right - 1
       => (([ (ArrayUpdate
              (ArrayUpdate A left ([ A  right ]))
              right ([ A  left ]))  j ]) . key) = pivot and
   partitioned(ArrayUpdate
             (ArrayUpdate A left ([ A  right ]))
              right ([ A  left ]), min, max,
              1 + left, right - 1, pivot)
```

The implication consists of two Further implications. The inequalities of the one follows from

$$max \geq left \quad \Rightarrow \quad 1 + max \geq left$$

and

$$right \geq min \quad \Rightarrow \quad right \geq min\text{-}1.$$

The *containsElement* predicates are (without a condition) part of the precondition. The universal quantified proposition is a consequence of the validity of the *partitioned* predicate

$$\begin{aligned} & \forall k \cdot left > k \wedge k \geq min \Rightarrow pivot \geq A[k].key \wedge \\ & \forall i \cdot max \geq i \wedge i > right \Rightarrow A[i].key \geq pivot \\ \Rightarrow \quad & \forall j \cdot left > j \wedge j > right \Rightarrow A[j].key = pivot, \end{aligned}$$

therefore the first implication holds. The inequalities of the second implication are shown by

$$\begin{array}{rcl}
\mathit{left} \geq \mathit{min} \wedge \mathit{min} \geq 0 & \Rightarrow & \mathit{left} \geq 0 \\
\mathit{max} \geq \mathit{right} & \Rightarrow & \mathit{max} \geq \mathit{right}\text{-}1 \\
\mathit{left} \geq \mathit{min} & \Rightarrow & 1 + \mathit{left} \geq \mathit{min} \\
\mathit{max} \geq \mathit{left} & \Rightarrow & 1 + \mathit{max} \geq 1 + \mathit{left} \\
\mathit{HIGH}(A) \geq \mathit{max} \wedge \mathit{max} \geq \mathit{right} & \Rightarrow & \mathit{HIGH}(A) \geq \mathit{right} \\
\mathit{HIGH}(A) \geq \mathit{max} \wedge \mathit{max} \geq \mathit{left} & \Rightarrow & \mathit{HIGH}(A) \geq \mathit{left} \\
\mathit{right} \geq \mathit{min} & \Rightarrow & \mathit{right}\text{-}1 \geq \mathit{min}\text{-}1 \\
\mathit{right} \geq \mathit{min} \wedge \mathit{min} \geq 0 & \Rightarrow & \mathit{right} \geq 0.
\end{array}$$

We have $\mathit{max} \geq \mathit{right}$ and $\mathit{right} > \mathit{right}\text{-}1 > \mathit{left} + 1$. After the evaluation of the *ArrayUpdate* expressions the new value of $A[\mathit{right}]$ is the old value of $A[\mathit{left}]$. According to the precondition

$$A[\mathit{left}].\mathit{key} \geq \mathit{pivot}$$

holds. Choosing $p = \mathit{right}$ in the definition of the predicate *containsElementGEQ* shows the validity of the existential quantified statement. Analogously choose $p = \mathit{left}$ for the predicate *containsElementLEQ*. The proposition

$$\begin{array}{ll}
& \forall k \cdot \mathit{left} + 1 > k \wedge k \geq \mathit{min} \\
\Rightarrow & \mathit{pivot} \geq (\mathrm{ArrayUpdate}\,(\mathrm{ArrayUpdate}\ A\ \mathit{left}\ A[\mathit{right}])\ \mathit{right}\ A[\mathit{left}])[k].\mathit{key}
\end{array}$$

follows from

$$\forall k \cdot \mathit{left} > k \wedge k \geq \mathit{min} \quad \Rightarrow \quad \mathit{pivot} \geq A[k].\mathit{key}$$

with

$$\mathit{pivot} \geq A[\mathit{right}].\mathit{key}$$

After the evaluation of the *ArrayUpdate* expressions the new value of $A[\mathit{left}]$ is the old value of $A[\mathit{right}]$. So

$$\begin{array}{ll}
& \forall i \cdot \mathit{max} \geq i \wedge i > \mathit{right}\text{-}1 \\
\Rightarrow & (\mathrm{ArrayUpdate}\,(\mathrm{ArrayUpdate}\ A\ \mathit{left}\ A[\mathit{right}])\ \mathit{right}\ A[\mathit{left}])[i].\mathit{key} \geq \mathit{pivot}.
\end{array}$$

Therefore the *partitioned* predicate is valid. The remaining universal quantified statement is a consequence of these two statements.

Sequence “recursion_left”

```
Proof obligation in lines 207:13-229:69:
   left    >= right and left   >= min   and
   min     >= 0     and max    >= right and
   1 + max >= left  and HIGH A >= max   and right >= min - 1 and
   right > left =>
     containsElementGEQ(A,left,max,pivot)  and
     containsElementLEQ(A,min,right,pivot) and
   forall  j : nat &
     left > j and j > right => (([ A  j ]) . key) = pivot and
   partitioned(A,min,max,left,right,pivot)
=> not right > INT(min) =>
     right > min =>
       sorted(A,min,right) and
   right > INT(min) =>
     HIGH A >= right and
     right  >= min and
     sorted(A,min,right) =>
       left    >= right and left   >= min   and
       min     >= 0     and max    >= right and
       1 + max >= left  and HIGH A >= max   and right >= min - 1 and
       right > left =>
         containsElementGEQ(A,left,max,pivot) and
         containsElementLEQ(A,min,right,pivot) and
       forall  j : nat &
         left > j and j > right => (([ A  j ]) . key) = pivot and
       partitioned(A,min,max,left,right,pivot)
```

Again, the major implication consists of two further implications. The first one is just an implication whose precondition is a negation of the precondition of the outer implication. Therefore, one of these precondition must be false. So the first statement is true.

The second implication contains another one with precondition *sorted*$(A, min, right)$. The conclusion of this implication is part of the precondition of the outer implication. Thus, the *sorted* implication holds. With the definition of the function `INT` in mind the inequality $right \geq min$ is part of the precondition. The remaining inequality $HIGH(A) \geq right$ follows from $HIGH(A) \geq max$ and $max \geq right$.

Sequence “recursion_right”

```
Proof obligation in lines 241:13-265:69:
   left     >= right and left   >= min   and
   min      >= 0     and  max   >= right and
   1  + max >= left  and HIGH A >= max   and right >= min - 1 and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot)and
   right > min => sorted(A,min,right) and
   forall  j : nat &
     left > j and j > right => (([ A  j ]) . key) = pivot and
```

```
   partitioned(A,min,max,left,right,pivot)
=> not INT(max) > left =>
     max > left =>
       sorted(A,left,max) and
   INT(max) > left =>
     max >= left and left >= 0 and
     sorted(A,left,max) =>
       left    >= right and left   >= min and
       min     >= 0     and max    >= right and
       1 + max >= left  and HIGH A >= max and
       right > left =>
         containsElementGEQ(A,left,max,pivot)  and
         containsElementLEQ(A,min,right,pivot) and
       right > min => sorted(A,min,right) and
       forall  j : nat &
         left > j and j > right => (([ A  j ]) . key) = pivot and
       partitioned(A,min,max,left,right,pivot)
```

This proof obligation is shown analogously.

Leaving the sequence "recursion_right"

```
Proof obligation in lines 241:13-265:69:
   left     >= right and left   >= min   and
   min      >= 0     and max    >= right and
   1  + max >= left  and HIGH A >= max   and right >= min - 1 and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot)
   right > min  => sorted(A,min,right) and
   max   > left => sorted(A,left,max) and
   forall  j : nat &
     left > j and j > right => (([ A  j ]) . key) = pivot and
   partitioned(A,min,max,left,right,pivot)
=> sorted(A,min,max)
```

>From *left* $\geq$ *right* and

$$\begin{aligned} \forall k \cdot left > k \geq min \Rightarrow pivot \geq A[k].key \\ \forall j \cdot left > j > right \Rightarrow A[j].key = pivot \\ \forall i \cdot max \geq i > right \Rightarrow A[i].key \geq pivot \end{aligned}$$

we know that the array is partioned in almost three parts where the element are less than, equal to, and greater than the pivot element. In the case $right > min$ and $max > left$ the validity of the *sorted* predicate follows so the array is sorted. In the case $right \leq min$ or $max \leq left$ the parts with elements less than and greater than the pivot element are empty. The elements of the remaining part are according to the precondition equal to the pivot element so the array is sorted in this case too.

All verification conditions have been shown valid. Thus, the implementation is partially correct with respect to the given specification. Because we have argued that the program will always terminate, this program is even totally correct.

6.2.3. Quicksort: variation 1

The number of the recursion steps during sorting is minimal if the sizes of the parts produced by the partition are "almost equal." Therefore, the pivot element is chosen best if the median of all elements is taken. An approximation is to choose the median of three elements. The following variation of the base algorithm uses this strategy. The pivot element will be the median of the elements on the left and the right border and the element in the middle. (Cf. appendix B.3)

For this variant, only the verification condition of the sequence *choose_ pivot* is different from the one in the base algorithm.

Sequence "choose_pivot"

```
Proof obligation in lines 61:13-66:70:
   max >= min and HIGH A >= max and min >= 0
=> not (([ A  min ]) . key) > (([ A  max ]) . key) =>
     not (([ A  (max + min) div 2 ]) . key) > (([ A  max ]) . key) =>
       not (([ A  min ]) . key) > (([ A  (max + min) div 2 ]) . key) =>
         min >= 0   and HIGH A >= max and
         max >= min and HIGH A >= (max + min) div 2 and
         (max + min) div 2 >= 0  and
         containsElementGEQ(A,min,max,([A (max + min) div 2]).key) and
         containsElementLEQ(A,min,max,([A (max + min) div 2]).key) and
       (([ A  min ]) . key) > (([ A  (max + min) div 2 ]) . key) =>
         min >= 0   and HIGH A >= min and
         max >= min and HIGH A >= max and
         containsElementLEQ(A,min,max,([ A  min ]) . key) and
         containsElementGEQ(A,min,max,([ A  min ]) . key) and
     (([ A  (max + min) div 2 ]) . key) > (([ A  max ]) . key) =>
       min >= 0   and HIGH A >= max and
       max >= min and max    >= 0   and
       containsElementLEQ(A,min,max,([ A  max ]) . key) and
       containsElementGEQ(A,min,max,([ A  max ]) . key) and
  (([ A  min ]) . key) > (([ A  max ]) . key) =>
    not (([ A  max ]) . key) > (([ A  (max + min) div 2 ]) . key) =>
      not (([ A  min ]) . key) > (([ A  (max + min) div 2 ]) . key) =>
        min >= 0   and HIGH A >= min and
        max >= min and HIGH A >= max and
        containsElementLEQ(A,min,max,([ A  min ]) . key) and
        containsElementGEQ(A,min,max,([ A  min ]) . key) and
      (([ A  min ]) . key) > (([ A  (max + min) div 2 ]) . key) =>
        min >= 0   and HIGH A >= (max + min) div 2 and
        max >= min and HIGH A >= max and
        (max + min) div 2 >= 0  and
        containsElementLEQ(A,min,max,([A (max + min) div 2]).key) and
        containsElementGEQ(A,min,max,([A (max + min) div 2]).key) and
```

```
  (([ A  max ]) . key) > (([ A  (max + min) div 2 ]) . key) =>
    min >= 0   and HIGH A >= max and
    max >= min and max    >= 0   and
    containsElementLEQ(A,min,max,([ A  max ]) . key) and
    containsElementGEQ(A,min,max,([ A  max ]) . key)
```

The correctness of the inequalities

$$\begin{aligned} max &\geq min \\ HIGH(A) &\geq max \\ HIGH(A) &\geq \frac{max+min}{2} \\ \frac{max+min}{2} &\geq 0 \\ min &\geq 0 \end{aligned}$$

follows either directly from the precondition or analogously to the proof of the base algorithm.

Both *min* and *max* as well as $\frac{max+min}{2}$ are in the range *min* ... *max* so the existential quantified propositions of the *containsElement* predicate is true when choosing one of these values.

Thus, this verification condition is valid. With this, the partial correctness of this variation with respect to the given specification is an immediate consequence of the correctness of the base algorithm.

6.2.4. Quicksort: variation 2

One more improvement of the algorithm is gained in stopping the recursion at time. In the following variation parts of the array of size “almost two” are sorted by swapping the elements. (appendix B.4)

In addition to the verification of the base algorithm three more conditions have to be proven.

Entering the Sequence “swap_sort”

```
Proof obligation in lines 32:13-36:46:
   max >= min and min >= 0 and HIGH A >= max
=> not max - min > 2
     => 2 >= max - min
```

The validity of this condition is a consequence of

$$\neg(max\text{-}min > 2) \quad \Rightarrow \quad 2 \geq max\text{-}min.$$

Sequence “swap_sort”

```
Proof obligation in lines 279:17-285:54:
   min    >= 0   and max >= min and
```

```
   HIGH A >= max and 2   >= max - min
=> not (([ A  min ]) . key) > (([ A  max ]) . key) =>
     (([ A  max ]) . key) >= (([ A  min ]) . key) and
   (([ A  min ]) . key) > (([ A  max ]) . key) =>
       HIGH A >= min and
       (([ A  min ]).key) >= (([[(ArrayUpdate
                                (ArrayUpdate A min ([A  max])) max
                                ([A  min]))  min]).key) and
       max >= 0
```

The final part of this implication consists of two other implications. The first follows from

$$\neg(A[min].key > A[max].key) \quad \Rightarrow \quad A[max].key \geq A[min].key$$

If $max \neq min$ the second expression can be simplified to

$$A[min].key \geq A[max].key$$

This holds because of the precondition $A[min].key > A[max].key$. In the case of $max = min$ the *ArrayUpdate* expressions do not alter the array so the inequality can be simplified to

$$A[min].key \geq A[min].key (= A[max].key)$$

This validity follows from the precondition $A[min].key > A[max].key$. The remaining inequalities are deduced from

$$\begin{aligned} HIGH(A) \geq max \wedge max \geq min &\quad \Rightarrow \quad HIGH(A) \geq min \\ max \geq min \wedge \mathrm{min} \geq 0 &\quad \Rightarrow \quad max \geq 0 \end{aligned}$$

Therefore, this verification condition is valid too.

Leaving the sequence “swap_sort”

```
Proof obligation in lines 279:17-285:54:
   min    >= 0   and max >= min and
   HIGH A >= max and 2   >= max - min and
   (([ A  max ]) . key) >= (([ A  min ]) . key)
=> sorted(A,min,max)
```

>From $max\text{-}min \leq 2$ we conclude that the part of the array under consideration consists of one or two elements. Because of

$$A[max].key \geq A[min].key$$

this subarray is sorted, so the verification condition has been shown.

6.2.5. Quicksort: variation 3

Another way of stopping the recursion earlier is to use a different sorting algorithm for parts whose size are small enough. In the following subarrays with an arbitrarily chosen size of at most 10 are sorted using *BubbleSort*. (appendix B.5)

The precondition of the sequence *Sort_ body* implies the precondition of the procedure call to *BubbleSort*. From the postcondition of the procedure call the postcondition of the sequence follows. Thus, the additional verification conditions holds.

The verification of this variant of the base algorithm can therefore be reduced to the verification of the base algorithm and the verification of *BubbleSort*.

6.3. Compression and decompression: The LZW algorithm

The most challenging program MOPS has been applied to so far is the LZW algorithm introduced by Lempel, Ziv and Welch in a sequence of papers [16, 18, 19]. To be precise, the LZW algorithm consists of a series of compression and decompression algorithms. One of these is used e. g. in the UNIX compress algorithm. The pair of the compression and decompression algorithm specified and verified with MOPS is the one in [16].

Because the specified program and the generated verification conditions are rather lengthy we do not present them here. Together with a description of the LZW algorithm they can be found in [11].

7. Conclusions

As opposed to other systems, e. g. the *Karlsruhe Interactive Verifier* (KIV, [13, 14, 15]), MOPS offers the possibility to verify existing software. KIV is a verification system based on algebraic specification and stepwise refinement. The development process in KIV starts with the specification of the planned software system using abstract data types. The specification language used is a first order subset of SPECTRUM. Furthermore, KIV needs the specification of the whole system whereas MOPS is able to specify and verify one or more selected parts of a program.

MOPS has deliberately been designed as a “small tool”. It combines established techniques as Hoare-style reasoning and specification-based reuse with established implementation and specification languages as Modula-2 and VDM-SL. This conceptual simplicity is—in our opinion—a major contribution of MOPS and makes it also suitable for educational purposes. Future work on MOPS includes the combination with fully automated theorem provers and the migration from the programming language Modula-2 to Java.

References

[1] K. R. Apt. Ten years of Hoare’s logic: A survey—part I. *ACM Trans. on Prog. Lang. and Systems*, 3:431–483, 1981.

[2] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer, New York, 1991.

[3] H. Bickel and W. Struckmann. The Hoare Logic of Data Types. Informatik-Bericht Nr. 95-04, Technische Universität Braunschweig, Februar 1995.

[4] B. Fischer, J. M. Ph. Schumann, and G. Snelting. Deduction-based software component retrieval. In *Automated Deduction - A Basis for Applications*, pages 265–292, Dordrecht, 1998. Kluwer.

[5] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada. *IEEE Trans. Software Engineering*, 16(9):1058–1075, 1990.

[6] B. Hohlfeld and W. Struckmann. *Einführung in die Programmverifikation.* BI-Wissenschaftsverlag, Mannheim, 1992.

[7] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall, New York, 2nd edition, 1990.

[8] Th. Kaiser. Behandlung von Datenstrukturen in einem VDM-basierten Prädikatentransformer für Modula-2, September 1998. Diplomarbeit, Technische Universität Braunschweig.

[9] D. Luckham, F. W. von Henke, B. Krieg-Brückner, and O. Owe. *ANNA - A Language for Annotating Ada Programs*, volume 260 of *Lect. Notes Comp. Sci.* Springer, 1987.

[10] D. C. Luckham and N. Suzuki. Verification of Array, Record and Pointer Operations in PASCAL. *ACM Trans. on Prog. Lang. and Systems*, 1(2):226–244, 1979.

[11] M. Nordmann and A. Reimers. Verifikation des LZW-Algorithmus, 2000. Studienarbeit, Technische Universität Braunschweig.

[12] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, 1991.

[13] W. Reif. The KIV-Approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, LNCS 1009, Berlin, 1995. Springer.

[14] W. Reif. Formale Methoden für sicherheitskritische Software – Der KIV-Ansatz. *Informatik Forsch. Entw.*, 14(4):193–202, 1999.

[15] W. Reif, G. Schellhorn, and K. Stenzel. Interactive Correctness Proofs for Software Modules Using KIV. In *Tenth Annual Conference on Computer Assurance*, IEEE press. NIST, Gaithersburg, MD, USA, 1995.

[16] T. Welch. A Technique for High Performance Data Compression. *IEEE Computer*, 17(6):8–19, 1984.

[17] N. Wirth. *Programmieren in Modula-2.* Springer, Berlin, 1991.

[18] J. Ziv and A. Lempel. A Universal Algorithm for Data Compression. *IEEE Trans. Information Theory*, 23:337–343, 1977.

[19] J. Ziv and A. Lempel. Compressions of Individuel Sequences Via Variable-Rate Coding. *IEEE Trans. Information Theory*, 24:530–536, 1978.

A. Gaussian sum formula

```
001: MODULE SumUpToN;
002:
003: FROM InOut IMPORT WriteString, WriteLn,
004:                   ReadCard, WriteCard;
005:
006: VAR sum, i, N : CARDINAL;
007:
008: BEGIN
009:     WriteString ("N = ");
010:     ReadCard (N);
011:
012:     (*{ entry sumFOR pre  N >= 0
013:                      post sum = N * (N + 1) div 2 }*)
014:     sum := 0;
015:
016:     (*{ loopinv
017:         sum = (i * (i - 1)) div 2 and i <= N + 1 }*)
018:     FOR i := 1 TO N DO
019:
020:         sum := sum + i;
021:
022:     END;
023:
024:     (*{ exit sumFOR }*);
025:
026:     WriteString ("Sum = "); WriteCard (sum, 4); WriteLn;
027:
028: END SumUpToN.
```

```
Proof obligation in lines 12:9-13:46:
  false => N >= 0


Proof obligation in lines 12:9-13:46:
  N >= 0  =>
  0  = (1  * (1  - 1 )) div 2 and
  1  <= 1  + N


Proof obligation in lines 12:9-13:46:
  summe = 15 => true


Proof obligation in line 16:9-66:
   exists  X_7 : nat &
     i <= 1  + X_7 and
     X_7 = N and
     ((i - 1 ) * i) div 2 = summe and
     i <= X_7
```

```
=> (((1  + i) - 1 ) * (1  + i)) div 2 = i + summe and
   1  + i <= 1  + N


Proof obligation in line 16:9-66:
   exists  X_7 : nat &
     i <= 1  + X_7 and
     X_7 = N and
     ((i - 1 ) * i) div 2 = summe and
     i > X_7
=> summe = N * (N + 1) div 2
```

B. Sorting algorithms

B.1. Bubblesort

B.1.1. The specified Modula-2 program

```
001: MODULE BubbleSortModule;
002:
003: (*  functions
004:         sorted : (seq of Element) * nat * nat -> bool
005:         sorted (a, i, j) ==
006:             forall p, q : nat &
007:             (p in set inds(a)) and (q in set inds(a)) and
008:             (i <= p) and (p <= q) and (q <= j)
009:             => a(p).key <= a(q).key        *)
010:
011: TYPE  Element = RECORD
012:                     key  : CARDINAL;
013:                     name : ARRAY [1..40] OF CHAR;
014:                 END;
015:
016: PROCEDURE BubbleSort (VAR A: ARRAY OF Element);
017:
018: VAR i, j : CARDINAL;
019:     hilf : Element;
020:
021: BEGIN
022:     (* entry Sort_Intern
023:         pre   HIGH(A) >= 0
024:         post  sorted (A, 0, HIGH(A)) *)
025:
026:     (* loopinv 0 <= i and i <= HIGH(A) and
027:                 HIGH(A) >= 0 and
028:                 sorted (A, HIGH(A) - i, HIGH(A)) *)
029:
030:     FOR i := 0 TO (HIGH(A) - 1) DO
031:
032:       (* loopinv 0 <= i and i <= HIGH(A) - 1 and
033:                    0 <= j and j <= HIGH(A) - i and
```

```
034:                   HIGH(A) >= 0 and
035:                   sorted(A, HIGH(A) - i, HIGH(A)) and
036:                   (forall k : nat &
037:                       (k >= 0) and (k < j) =>
038:                       A(k).key <= A(j).key)      *)
039:
040:       FOR j := 0 TO  (HIGH(A) - 1 - i) DO
041:
042:         (* entry swap
043:             pre 0 <= i and i <= HIGH(A) - 1 and
044:                 0 <= j and j <= HIGH(A) - 1 - i and
045:                 HIGH(A) >= 0 and
046:                 sorted(A, HIGH(A) - i, HIGH(A)) and
047:                 (forall k : nat &
048:                     (k >= 0) and (k < j) =>
049:                     A(k).key <= A(j).key)
050:
051:             post 0 <= i and i <= HIGH(A) - 1 and
052:                  0 <= j and j <= HIGH(A) - 1 - i and
053:                  HIGH(A) >= 0 and
054:                  sorted(A, HIGH(A) - i, HIGH(A)) and
055:                  (forall k : nat &
056:                      (k >= 0) and (k < j) =>
057:                      A(k).key <= A(j).key) and
058:                      A(j + 1).key >= A(j).key    *)
059:
060:         IF (A[j].key > A[j + 1].key) THEN
061:           hilf     := A[j + 1];
062:           A[j + 1] := A[j];
063:           A[j]     := hilf;
064:         END;
065:         (* exit swap *)
066:
067:       END;
068:     END;
069:    (* exit Sort_Intern *)
070:
071: END BubbleSort;
072:
073: BEGIN
074: END BubbleSortModule.
```

B.1.2. Proof obligations

```
Proof obligation in lines 22:9-24:44:
  false => HIGH A >= 0

Proof obligation in lines 22:9-24:44:
   HIGH A >= 0
=> HIGH A >= 0 and  0 <= HIGH A and
   0 <= 0  and
```

```
   sorted(A, HIGH A - 0 , HIGH A)

Proof obligation in lines 22:9-24:44:
   sorted(A, 0, HIGH A) => true

Proof obligation in lines 26:9-28:56:
   exists  X_2 : nat &
     HIGH A >= 0       and i <= X_2 and
     i      <= HIGH A  and 0 <= i and
     HIGH A - 1 = X_2 and
     sorted(A, HIGH A - i, HIGH A)
=> HIGH A >= 0 and i <= HIGH A - 1 and
   0  <= i  and 0  <= HIGH A - i and
   0  <= 0  and
   forall  k : nat &
     k >= 0 and k < 0
     => (([ A  k ]) . key) <= (([ A  0  ]) . key) and
   sorted(A, HIGH A - i, HIGH A)

Proof obligation in lines 26:9-28:56:
   exists  X_2 : nat &
     HIGH A >= 0 and i > X_2 and
     i <= HIGH A  and 0  <= i and
     HIGH A - 1 = X_2 and
     sorted(A, HIGH A - i, HIGH A)
=> sorted(A, 0, HIGH A)

Proof obligation in lines 32:11-38:57:
   exists  X_3 : nat &
     HIGH A >= 0 and j <= X_3 and
     j <= HIGH A - i and 0 <= i and
     i <= HIGH A - 1 and 0 <= j and
     (HIGH A - 1) - i = X_3 and
     forall  k : nat &
       k >= 0 and k < j
       => (([ A  k ]) . key) <= (([ A  j ]) . key) and
     sorted(A, HIGH A - i, HIGH A)
=> HIGH A >= 0 and j <= (HIGH A - 1) - i and
   i <= HIGH A - 1 and  0  <= j and
   0  <= i and
   forall  k : nat &
     k >= 0 and k < j
     => (([ A  k ]) . key) <= (([ A  j ]) . key) and
   sorted(A, HIGH A - i, HIGH A)

Proof obligation in lines 32:11-38:57:
   exists  X_3 : nat &
     HIGH A >= 0 and j > X_3 and
     j <= HIGH A - i and
     i <= HIGH A - 1 and
     0  <= j and 0 <= i and
     (HIGH A - 1) - i = X_3 and
```

```
     forall  k : nat &
       k >= 0 and k < j
       => (([ A  k ]) . key) <= (([ A  j ]) . key) and
     sorted(A, HIGH A - i, HIGH A)
=> HIGH A >= 0 and 1  + i <= HIGH A and
   0  <= 1  + i and
   sorted(A, HIGH A - (1 + i), HIGH A)

Proof obligation in lines 42:13-58:57:
   HIGH A >= 0 and j <= (HIGH A - 1) - i and
   i <= HIGH A - 1 and 0  <= j and
   0  <= i and
   forall  k : nat &
     k >= 0 and k < j
     => (([ A  k ]) . key) <= (([ A  j ]) . key) and
   sorted(A, HIGH A - i, HIGH A)
=> not (([ A  j ]) . key) > (([ A  1  + j ]) . key) =>
     HIGH A >= 0 and
     (([ A  1  + j ]) . key) >= (([ A  j ]) . key) and
     j <= (HIGH A - 1) - i and  i <= HIGH A - 1 and
     0  <= j and 0  <= i and
     forall  k : nat &
       k >= 0 and k < j
       => (([ A  k ]) . key) <= (([ A  j ]) . key) and
     sorted(A, HIGH A - i, HIGH A) and
   (([ A  j ]) . key) > (([ A  1  + j ]) . key) =>
     j >= 0  and j >= 0  and
     1  + j >= 0  and 1  + j >= 0  and
     HIGH (ArrayUpdate
            (ArrayUpdate A 1  + j ([ A  j ]))
          j ([ A  1  + j ])) >= 0 and
    (([ (ArrayUpdate
        (ArrayUpdate A 1  + j ([ A  j ]))
        j ([ A  1  + j ]))  1  + j]) . key) >=
    (([ (ArrayUpdate
        (ArrayUpdate A 1  + j ([ A  j ]))
        j ([ A  1  + j ]))  j ]) . key) and
    j <= HIGH A and
    j <= HIGH (ArrayUpdate A 1  + j ([ A  j ])) and
    j <= (HIGH (ArrayUpdate
               (ArrayUpdate A 1  + j ([ A  j ]))
               j ([ A  1  + j ])) - 2) - i and
    i <= HIGH (ArrayUpdate
              (ArrayUpdate A 1  + j ([ A  j ]))
              j ([ A  1  + j ])) - 2 and
    1  + j <= HIGH A and 1  + j <= HIGH A and
    0  <= j and  0  <= i and
    forall  k : nat &
      k >= 0 and k < j
      => (([ (ArrayUpdate
             (ArrayUpdate A 1  + j ([ A  j ]))
             j ([ A  1  + j ]))  k ]) . key) <=
```

```
         (([ (ArrayUpdate
             (ArrayUpdate A 1  + j ([ A  j ]))
             j ([ A  1  + j ])) j ]) . key) and
    sorted(ArrayUpdate
          (ArrayUpdate A 1  + j ([ A  j ]))
          j ([ A  1  + j ]),
          (HIGH (ArrayUpdate
                (ArrayUpdate A 1  + j ([ A  j ]))
                j ([ A  1  + j ])) - 1 ) - i,
          HIGH (ArrayUpdate
               (ArrayUpdate A 1  + j ([ A  j ]))
               j ([ A  1  + j ])) - 1 )

Proof obligation in lines 42:13-58:57:
   HIGH A >= 0 and
   (([ A  1  + j ]) . key) >= (([ A  j ]) . key) and
   j <= (HIGH A - 1) - i and i <= HIGH A - 1 and
   0  <= j and 0  <= i and
   forall  k : nat &
     k >= 0 and k < j
     => (([ A  k ]) . key) <= (([ A  j ]) . key) and
   sorted(A, HIGH A - i, HIGH A)
=> HIGH A >= 0 and i <= HIGH A - 1 and
   1  + j <= HIGH A - i and
   0  <= i and 0  <= 1  + j and
   forall  k : nat &
     k>= 0 and k < 1  + j
     => (([ A  k ]) . key) <= (([ A  1  + j ]) . key) and
   sorted(A, HIGH A - i, HIGH A)
```

B.2. Quicksort: base algorithm

B.2.1. The Modula-2 program

```
001: MODULE QuickSortModule;

035:
036: (* ------------------------------------------------------------ *)
037: (* QuickSort                                                    *)
038: (* ------------------------------------------------------------ *)
039: (* Sorts the Array A of Elements using the QuickSort-Algorithm. *)
040: (* QuickSort is a Divide-and-Conquer-Algorithm. A will be       *)
041: (* partitioned in two parts, which will seperately be sorted.   *)
042: (* ------------------------------------------------------------ *)
043: PROCEDURE QuickSort (VAR A: ARRAY OF Element);
044:
045:     (* -------------------------------------------------------- *)
046:     (* Sort                                                     *)
047:     (* -------------------------------------------------------- *)
048:     (* Sorts the [min, max]-part of the array A                 *)
049:     (* -------------------------------------------------------- *)
```

```
PROCEDURE Sort (min, max : CARDINAL);

VAR pivot       : CARDINAL;
    left, right : INTEGER;
    swap        : Element;

BEGIN

    (* ---------------------------------------------------- *)
    (* Choosing an element, which determines the partition: *)
    (* here, simply the element in the middle of the array  *)
    (* ---------------------------------------------------- *)
    pivot := A [(min + max) DIV 2].key;
    left  := min;
    right := max;

    (* ---------------------------------------------------- *)
    (*                    DIVIDE ...                        *)
    (* ---------------------------------------------------- *)
    (* Partitioning [min ... max]:                          *)
    (* ---------------------------------------------------- *)
    WHILE (right > left) DO

        (* ------------------------------------------------ *)
        (* Finding an element with a key greater/equal      *)
        (* pivot                                            *)
        (* ------------------------------------------------ *)
        WHILE A[left].key < pivot DO
            left := left + 1;
        END;

        (* ----------------------------------------------- *)
        (* Finding an element with a key less/equal pivot  *)
        (* ----------------------------------------------- *)
        WHILE A[right].key > pivot DO
            right := right - 1;
        END;

        (* ------------------------------------------------ *)
        (* Swap the elements, if in wrong order             *)
        (* ------------------------------------------------ *)
        IF (left <= right) THEN
            swap     := A[left];
            A[left]  := A[right];
            A[right] := swap;

            left  := left  + 1;
            right := right - 1;
        END;

    END;
```

```
        (* --------------------------------------------------- *)
        (* If there is a less/equal-pivot-part, sort it.       *)
        (* --------------------------------------------------- *)
        IF INT(min) < right THEN
            Sort (min, right);
        END;

        (* --------------------------------------------------- *)
        (* If there is a greater/equal-pivot-part, sort it.    *)
        (* --------------------------------------------------- *)
        IF INT(max) > left THEN
            Sort (left, max);
        END;

        (* --------------------------------------------------- *)
        (*             ... AND CONQUER                         *)
        (* --------------------------------------------------- *)

    END Sort;

BEGIN

    Sort (0, HIGH(A));

END QuickSort;

BEGIN
END QuickSortModule.
```

B.2.2. The specified program

```
MODULE QuickSortModule;

TYPE  Element = RECORD
                    key  : CARDINAL;
                    name : ARRAY [1..40] OF CHAR;
                END;

(* functions
        sorted : (seq of Element) * nat * nat -> bool
        sorted (a, i, j) ==
            forall p, q : nat &
            (p in set inds(a)) and (q in set inds(a)) and
            (i <= p) and (p <= q) and (q <= j)
            => a(p).key <= a(q).key;

        containsElementGEQ:
        (seq of Element) * nat * nat * nat -> bool
        containsElementGEQ (A, i, j, v) ==
            (exists p : nat &
               p >= i and p <= j and A(p).key >= v);
```

```

        containsElementLEQ:
        (seq of Element) * nat * nat * nat -> bool
        containsElementLEQ (A, i, j, v) ==
            (exists p : nat &
               p >= i and p <= j and A(p).key >= v);

        partitioned:
        (seq of Element) * nat * nat * int * int * nat -> bool
        partitioned (A, min, max, left, right, pivot) ==
            (forall k : nat & (min <= k) and (k < left)
                => A(k).key <= pivot) and
            (forall i : nat & (right <  i) and (i <= max)
                => A(i).key >= pivot) *)

(* ------------------------------------------------------------ *)
(* QuickSort                                                    *)
(* ------------------------------------------------------------ *)
(* Sorts the Array A of Elements using the QuickSort-Algorithm. *)
(* QuickSort is a Divide-and-Conquer-Algorithm. A will be       *)
(* partitioned in two parts, which will seperately be sorted.   *)
(* ------------------------------------------------------------ *)
PROCEDURE QuickSort (VAR A: ARRAY OF Element);

    (* -------------------------------------------------------- *)
    (* Sort                                                     *)
    (* -------------------------------------------------------- *)
    (* Sorts the [min, max]-part of the array A                 *)
    (* -------------------------------------------------------- *)
    PROCEDURE Sort (min, max : CARDINAL);

    VAR pivot       : CARDINAL;
        left, right : INTEGER;
        swap        : Element;

    BEGIN
        (* entry Sort_body
            pre   0 <= min and min <= max and max <= HIGH(A)
            post  sorted (A, min, max) *)

        (* entry choose_pivot
            pre   0 <= min and min <= max and max <= HIGH(A)
            post  0 <= min and min <= max and max <= HIGH(A) and
                  left  = min and right = max and
                  containsElementGEQ (A, left, max, pivot) and
                  containsElementLEQ    (A, min, right, pivot) *)
        (* ---------------------------------------------------- *)
        (* Choosing an element, which determines the partition: *)
        (* here, simply the element in the middle of the array  *)
        (* ---------------------------------------------------- *)
        pivot := A [(min + max) DIV 2].key;
        left  := min;
```

```
        right := max;
        (* exit choose_pivot *);

        (* loopinv
            0       <= min   and max   <= HIGH(A) and
            min     <= left  and left  <= max + 1 and
            min - 1 <= right and right <= max     and
            partitioned (A, min, max, left, right, pivot) and
            (forall j : nat & (right <   j) and (j <  left)
               => A(j).key = pivot) and
            (right > left =>
                  containsElementGEQ (A, left, max, pivot) and
                  containsElementLEQ (A, min, right, pivot)) *)
        (* ---------------------------------------------------- *)
        (*                    DIVIDE ...                        *)
        (* ---------------------------------------------------- *)
        (* Partitioning [min ... max]:                          *)
        (* After the execution of the loop, the following holds *)
        (* (each part of the partition may be empty)            *)
        (*     [min ... right] [ ... ] [left ... max]           *)
        (*        <= pivot     = pivot    >= pivot              *)
        (*                                                      *)
        (* In every step, an element with a key greater and an  *)
        (* element less than pivot is searched (using left and  *)
        (* right). If they are in the wrong order (left <=      *)
        (* right), they will be swaped. So in every step, the   *)
        (* [min...left]-part will contain only element withs    *)
        (* keys less/equal pivot and the [right...max] part     *)
        (* only elements with keys greater/equal pivot. In other*)
        (* words, the partitioned-predicate holds. Furthermore, *)
        (* as a consequence of it, for all elements with an     *)
        (* index greater than right and less than left the key  *)
        (* must be = pivot. If left >= right, there are no more *)
        (* elements to swap, so the partition is done.          *)
        (* ---------------------------------------------------- *)
        WHILE (right > left) DO

            (* entry find_elements_to_swap
                pre   0    <= min   and max  <= HIGH(A) and
                      min  <= left  and right <= max and
                      left <  right and
                      partitioned (A,min,max,left,right,pivot) and
                      (forall j : nat & (right < j) and (j < left)
                        => A(j).key = pivot) and
                      (right > left =>
                         containsElementGEQ (A,left,max,pivot) and
                         containsElementLEQ (A,min,right,pivot))

                post  0   <= min   and max   <= HIGH(A) and
                      min <= left  and left  <= max and
                      min <= right and right <= max and
                      A(left).key  >= pivot and
```

```
                      A(right).key <= pivot and
                      partitioned (A,min,max,left,right,pivot) and
                      containsElementGEQ (A, left, max, pivot) and
                      containsElementLEQ (A, min, right,pivot) *)

            (* loopinv
                 0   <= min   and max   <= HIGH(A) and
                 min <= left  and left  <= max and
                 min <= right and right <= max and
                 partitioned (A, min, max, left, right, pivot) and
                 containsElementGEQ (A, left, max, pivot) and
                 containsElementLEQ (A, min, right, pivot) *)
            (* ------------------------------------------------ *)
            (* Finding an element with a key greater/equal      *)
            (* pivot                                            *)
            (* The loop terminates because...                   *)
            (*   - in the first step of the outer loop, there   *)
            (*     is at least the pivot element,               *)
            (*   - in every further step there has been in      *)
            (*     the preceeding step an element greater/equal *)
            (*     and one less/equal and they have been        *)
            (*     swapped, so this loop will stop, if left is  *)
            (*     the index of this element.                   *)
            (* The containsElementGEQ-predicate holds.          *)
            (* ------------------------------------------------ *)
            WHILE A[left].key < pivot DO
                left := left + 1;
            END;

            (* loopinv
                 0   <= min   and max   <= HIGH(A) and
                 min <= left  and left  <= max and
                 min <= right and right <= max and
                 A(left).key  >= pivot and
                 partitioned (A, min, max, left, right, pivot) and
                 containsElementGEQ (A, left, max, pivot) and
                 containsElementLEQ (A, min, right, pivot)    *)
            (* ----------------------------------------------- *)
            (* Finding an element with a key less/equal pivot  *)
            (*                                                 *)
            (* The loop terminates because of argument similar *)
            (* to the left-loop, so the containsElementLEQ-    *)
            (* predicate holds.                                *)
            (* ----------------------------------------------- *)
            WHILE A[right].key > pivot DO
                right := right - 1;
            END;
            (* exit find_elements_to_swap *);

            (* entry swap_elements
                pre   0   <= min   and max  <= HIGH(A) and
                      min <= left  and left  <= max and
```

```
                  min <= right and right <= max and
                  A(left).key  >= pivot and
                  A(right).key <= pivot and
                  partitioned (A,min,max,left,right,pivot) and
                  containsElementGEQ (A, left, max, pivot) and
                  containsElementLEQ (A, min, right, pivot)

            post  0   <= min   and max  <= HIGH(A) and
                  min <= left  and left <= max + 1 and
                  min - 1 <= right and right <= max and
                  partitioned (A,min,max,left,right,pivot) and
                  (forall j : nat & (right < j) and (j < left)
                    => A(j).key = pivot) and
                  (right > left =>
                   containsElementGEQ (A,left,max,pivot) and
                   containsElementLEQ (A,min,right,pivot)) *)
        (* ------------------------------------------------ *)
        (* Swap the elements, if in wrong order             *)
        (* ------------------------------------------------ *)
        IF (left <= right) THEN
            swap     := A[left];
            A[left]  := A[right];
            A[right] := swap;

            left  := left  + 1;
            right := right - 1;
        END;
        (* exit swap_elements *)
    END;

    (* entry recursion_left
        pre  0       <= min   and max   <= HIGH(A) and
             min     <= left  and left  <= max + 1 and
             min - 1 <= right and right <= max     and
             left    >= right and
             partitioned (A, min, max, left, right, pivot) and
             (forall j : nat & (right < j) and (j < left)
                => A(j).key = pivot) and
             (right > left =>
               containsElementGEQ (A, left, max, pivot) and
               containsElementLEQ (A, min, right, pivot))

        post 0       <= min   and max   <= HIGH(A) and
             min     <= left  and left  <= max + 1 and
             min - 1 <= right and right <= max     and
             left    >= right and
             partitioned (A, min, max, left, right, pivot) and
             (forall j : nat & (right < j) and (j < left)
                => A(j).key = pivot) and
             (min < right => sorted(A, min, right)) and
             (right > left =>
               containsElementGEQ (A, left, max, pivot) and
```

```
                   containsElementLEQ (A, min, right, pivot)) *)
        (* --------------------------------------------------- *)
        (* If there is a less/equal-pivot-part, sort it.       *)
        (* Otherwise, the recursion stops. At least for an     *)
        (* array that contains just one element, the less/equal*)
        (* part will be empty.                                 *)
        (* --------------------------------------------------- *)
        IF INT(min) < right THEN
            Sort (min, right);
        END;
        (* exit recursion_left *);

        (* entry recursion_right
            pre  0       <= min   and max   <= HIGH(A) and
                 min     <= left  and left  <= max + 1 and
                 min - 1 <= right and right <= max     and
                 left    >= right and
                 partitioned (A, min, max, left, right, pivot) and
                 (forall j : nat & (right < j) and (j < left)
                    => A(j).key = pivot) and
                 (min < right => sorted(A, min, right)) and
                 (right > left =>
                   containsElementGEQ (A, left, max, pivot) and
                   containsElementLEQ (A, min, right, pivot))

            post 0       <= min   and max   <= HIGH(A) and
                 min     <= left  and left  <= max + 1 and
                 min - 1 <= right and right <= max     and
                 left    >= right and
                 partitioned (A, min, max, left, right, pivot) and
                 (forall j : nat & (right < j) and (j < left)
                    => A(j).key = pivot) and
                 (min < right => sorted(A, min, right)) and
                 (max > left  => sorted(A, left, max))  and
                 (right > left =>
                   containsElementGEQ (A, left, max, pivot) and
                   containsElementLEQ (A, min, right, pivot)) *)
        (* --------------------------------------------------- *)
        (* If there is a greater/equal-pivot-part, sort it.    *)
        (* The recursion terminates because of the same        *)
        (* argument as for the left-recursion.                 *)
        (* --------------------------------------------------- *)
        IF INT(max) > left THEN
            Sort (left, max);
        END;
        (* exit recursion_right *)

        (* --------------------------------------------------- *)
        (*             ... AND CONQUER                          *)
        (* --------------------------------------------------- *)
        (* The less/equal-pivot- and the greater/equal-pivot-  *)
        (* part of the min-max-array are sorted, so the whole  *)
```

```
281:         (* array is sorted.                                  *)
282:         (* ------------------------------------------------- *)
283:         (* exit Sort_body *)
284:     END Sort;
285:
286: BEGIN
287:     (* entry QuickSort_body
288:         pre   HIGH(A) >= 0
289:         post  sorted (A, 0, HIGH(A)) *)
290:
291:     Sort (0, HIGH(A));
292:
293:     (* exit QuickSort_body *)
294: END QuickSort;
295:
296: BEGIN
297: END QuickSortModule.
```

B.2.3. Proof obligations

```
Proof obligation in lines 57:13-59:46:
  false => max <= HIGH A and 0 <= min and min <= max

Proof obligation in lines 57:13-59:46:
   max <= HIGH A and 0 <= min and min <= max
=> max <= HIGH A and 0 <= min and min <= max

Proof obligation in lines 57:13-59:46:
  sorted(A,min,max) => true

Proof obligation in lines 61:13-66:70:
   max <= HIGH A and 0  <= min and min <= max
=> min <= max and max <= HIGH A and
   (max + min) div 2 <= HIGH A and (max + min) div 2 >= 0 and
   0 <= min and min = min and max = max and
   containsElementGEQ(A,min,max,([ A  (max + min) div 2 ]) . key) and
   containsElementLEQ(A,min,max,([ A  (max + min) div 2 ]) . key)

Proof obligation in lines 61:13-66:70:
   max <= HIGH A and 0  <= min   and
   min  = left   and max = right and min <= max and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right,pivot)
=> left <= 1  + max and min     <= left and
   max  <= HIGH A   and min - 1 <= right and
   0    <= min      and right   <= max and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
```

```
      containsElementLEQ(A,min,right,pivot)
    forall  j : nat &
      right < j and j < left => (([ A  j ]) . key) = pivot and
    partitioned(A,min,max,left,right,pivot)

Proof obligation in lines 76:13-85:70:
    right    <= max   and left <= 1  + max and
    min      <= left  and max  <= HIGH A   and
    min - 1  <= right and 0    <= min      and right > left and
    right > left =>
      containsElementGEQ(A,left,max,pivot) and
      containsElementLEQ(A,min,right,pivot) and
    forall  j : nat &
      right < j and j < left => (([ A  j ]) . key) = pivot and
    partitioned(A,min,max,left,right,pivot)
=> min <= left and max <= HIGH A and
    0   <= min  and left < right  and right <= max and
    right > left =>
      containsElementGEQ(A,left,max,pivot) and
      containsElementLEQ(A,min,right,pivot) and
    forall  j : nat &
      right < j and j < left => (([ A  j ]) . key) = pivot and
    partitioned(A,min,max,left,right,pivot)

Proof obligation in lines 76:13-85:70:
    left <= 1  + max and min      <= left  and
    max  <= HIGH A   and min - 1  <= right and
    0    <= min      and not right > left  and right <= max and
    right > left =>
      containsElementGEQ(A,left,max,pivot) and
      containsElementLEQ(A,min,right,pivot) and
    forall  j : nat &
      right < j and j < left => (([ A  j ])  . key)  = pivot and
    partitioned(A,min,max,left,right,pivot)
=> right    <= max   and left <= 1  + max and
    min      <= left  and max  <= HIGH A   and
    min - 1  <= right and 0    <= min      and left >= right and
    right > left =>
      containsElementGEQ(A,left,max,pivot) and
      containsElementLEQ(A,min,right,pivot) and
    forall  j : nat &
      right < j and j < left => (([ A  j ])  . key)  = pivot and
    partitioned(A,min,max,left,right,pivot)

Proof obligation in lines 110:17-128:70:
    min <= left and max  <= HIGH A and
```

```
   0   <= min  and left < right   and right <= max and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot) and
   forall  j : nat &
     right < j and j < left => (([ A  j ]) . key)  = pivot and
   partitioned(A,min,max,left,right,pivot)
=> left <= max  and min <= right  and 0     <= min and
   min  <= left and max <= HIGH A and right <= max and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right,pivot) and
   partitioned(A,min,max,left,right,pivot)
Proof obligation in lines 110:17-128:70:
   right <= max    and left <= max and
   min   <= right  and min  <= left and
   max   <= HIGH A and 0    <= min and
   (([ A left ]) . key)   >= pivot and
   (([ A  right ]) . key) <= pivot and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right,pivot) and
   partitioned(A,min,max,left,right,pivot)
=> right <= max    and left <= max and
   min   <= right  and min  <= left and
   max   <= HIGH A and 0    <= min and
   (([ A left ]) . key)   >= pivot and
   (([ A  right ]) . key) <= pivot and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right,pivot) and
   partitioned(A,min,max,left,right,pivot)

Proof obligation in lines 130:17-136:70:
   left <= max  and min   <= right and
   min  <= left and max   <= HIGH A and
   0    <= min  and right <= max and
   (([ A  left ]) . key) < pivot and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right,pivot) and
   partitioned(A,min,max,left,right,pivot)
=> min <= right  and min <= 1  + left and
   max <= HIGH A and 1  + left <= max and
   0  <= min     and right <= max and
   containsElementGEQ(A,1 + left,max,pivot) and
   containsElementLEQ(A,min,right,pivot) and
   partitioned(A,min,max,left + 1,right,pivot)

Proof obligation in lines 130:17-136:70:
```

```
   left <= max    and min <= right and min   <= left and
   max  <= HIGH A and 0   <= min   and right <= max and
   not (([ A  left ]) . key) < pivot and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right,pivot) and
   partitioned(A,min,max,left,right,pivot)
=> right <= max  and left <= max    and min <= right and
   min   <= left and max  <= HIGH A and 0   <= min and
   (([ A  left ]) . key) >= pivot and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right,pivot) and
   partitioned(A,min,max,left,right,pivot)

Proof obligation in lines 154:17-161:69:
   (([ A  right ]) . key) > pivot and
   (([ A  left ]) . key) >= pivot and
   right <= max    and left <= max and
   min   <= right  and min  <= left and
   max   <= HIGH A and 0    <= min and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right,pivot) and
   partitioned(A,min,max,left,right,pivot)
=> left       <= max       and min <= left and
   min        <= right - 1 and max <= HIGH A and
   right - 1  <= max       and 0   <= min and
   (([ A  left ]) . key) >= pivot and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right - 1,pivot) and
   partitioned(A,min,max,left,right - 1,pivot)

Proof obligation in lines 154:17-161:69:
   right <= max    and left <= max and
   min   <= right  and min  <= left and
   max   <= HIGH A and 0    <= min and
   (([ A  left ]) . key) >= pivot and
   not (([ A  right ]) . key) > pivot and
   containsElementLEQ(A,min,right,pivot)
   containsElementGEQ(A,left,max,pivot) and
   partitioned(A,min,max,left,right,pivot) and
=> left <= max  and max   <= HIGH A and
   min  <= left and min   <= right  and
   0    <= min  and right <= max and
   (([ A  left ]) . key)  >= pivot and
   (([ A  right ]) . key) <= pivot and
   containsElementLEQ(A,min,right,pivot) and
   containsElementGEQ(A,left,max,pivot) and
```

```
   partitioned(A,min,max,left,right,pivot) and

Proof obligation in lines 174:17-192:70:
   right <= max    and left <= max and
   min   <= right  and min  <= left and
   max   <= HIGH A and 0    <= min and
   (([ A  left ]) . key) >= pivot and
   (([ A  right ]) . key) <= pivot and
   containsElementGEQ(A,left,max,pivot) and
   containsElementLEQ(A,min,right,pivot) and
   partitioned(A,min,max,left,right,pivot)
=> not left <= right =>
     left <= 1  + max and min     <= left and
     max  <= HIGH A   and min - 1 <= right and
     0    <= min      and right   <= max and
     right > left =>
       containsElementGEQ(A,left,max,pivot) and
       containsElementLEQ(A,min,right,pivot) and
     forall  j : nat &
       right < j and j < left  => (([ A  j ]) . key)  = pivot and
     partitioned(A,min,max,left,right,pivot)
   left <= right =>
     right >= 0 and left >= 0  and
     left >= 0  and right <= HIGH A and
     right <= HIGH (ArrayUpdate A left ([ A  right ])) and
     left <= HIGH A and left <= HIGH A and
     min <= 1  + left and right >= 0 and
     max <= HIGH (ArrayUpdate
                 (ArrayUpdate A left ([ A  right ]))
                 right ([ A  left ])) and
     1  + left <= 1  + max and right - 1  <= max and
     min - 1  <= right - 1  and 0  <= min and
     right - 1  > 1  + left =>
       containsElementGEQ(ArrayUpdate
                         (ArrayUpdate A left ([ A  right ]))
                          right ([ A left ]),1 + left,max,pivot) and
       containsElementLEQ(ArrayUpdate
                         (ArrayUpdate A left ([ A  right ]))
                          right ([ A  left ]),min,right - 1,pivot) and
     forall  j : nat &
       right - 1  < j and j < 1  + left
       => (([ (ArrayUpdate
              (ArrayUpdate A left ([ A  right ]))
              right ([ A  left ]))  j ]) . key) = pivot and
     partitioned(ArrayUpdate
                (ArrayUpdate A left ([ A  right ]))
```

```
                 right ([ A  left ]),
                 min, max, 1 + left, right - 1, pivot)

Proof obligation in lines 174:17-192:70:
   left <= 1  + max and min      <= left and
   max  <= HIGH A   and min - 1  <= right and
   0    <= min      and right    <= max and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot) and
   forall  j : nat &
     right < j and j < left => (([ A  j ]) . key)  = pivot and
   partitioned(A,min,max,left,right,pivot)
=> left <= 1  + max and min    <= left and
   max  <= HIGH A  and min - 1 <= right and
   0    <= min      and right  <= max
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot) and
   forall  j : nat &
     right < j and j < left => (([ A  j ]) . key)  = pivot and
   partitioned(A,min,max,left,right,pivot)

Proof obligation in lines 207:13-229:69:
   right    <= max   and left <= 1  + max and
   min      <= left  and max  <= HIGH A   and
   min - 1  <= right and 0    <= min      and left >= right and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot) and
   forall  j : nat &
     right < j and j < left => (([ A  j ]) . key)  = pivot and
   partitioned(A,min,max,left,right,pivot)
=> not INT(min) < right =>
     right   <= max   and left <= 1  + max and
     min     <= left  and max  <= HIGH A and
     min - 1 <= right and 0    <= min and
     right > left =>
       containsElementGEQ(A,left,max,pivot) and
       containsElementLEQ(A,min,right,pivot) and
     min < right => sorted(A,min,right) and
     forall  j : nat &
       right < j and j < left => (([ A  j ]) . key)  = pivot and
     partitioned(A,min,max,left,right,pivot) and
     left >= right and
   INT(min) < right =>
```

```
     min <= right and  0  <= min and
     sorted(A,min,right) =>
       right <= max and left <= 1  + max and
       min <= left and max <= HIGH A and
       min - 1  <= right and 0  <= min and
       right > left =>
         containsElementGEQ(A,left,max,pivot) and
         containsElementLEQ(A,min,right,pivot) and
       min < right =>
         sorted(A,min,right) and
       forall  j : nat &
         right < j and j < left => (([ A  j ]) . key)  = pivot and
       partitioned(A,min,max,left,right,pivot) and
       left >= right and
     right <= HIGH A

Proof obligation in lines 207:13-229:69:
   right    <= max   and left <= 1  + max and
   min      <= left  and max  <= HIGH A   and
   min - 1  <= right and 0    <= min      and left >= right
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot) and
   min < right =>
     sorted(A,min,right) and
   forall  j : nat &
     right < j and j < left => (([ A  j ]) . key)  = pivot and
   partitioned(A,min,max,left,right,pivot)
=> right    <= max   and left <= 1  + max and
   min      <= left  and max  <= HIGH A   and
   min - 1  <= right and 0    <= min      and left >= right and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot) and
   min < right => sorted(A,min,right) and
   forall  j : nat &
     right < j and j < left => (([ A  j ]) . key)  = pivot and
   partitioned(A,min,max,left,right,pivot)

Proof obligation in lines 241:13-265:69:
   right    <= max   and left <= 1  + max and
   min      <= left  and max  <= HIGH A   and
   min - 1  <= right and 0    <= min      and left >= right and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot) and
```

```
   min < right => sorted(A,min,right) and
   forall  j : nat &
     right < j and j < left => (([ A  j ]) . key)  = pivot and
   partitioned(A,min,max,left,right,pivot)
=> not INT(max) > left =>
     right    <= max   and left <= 1  + max and
     min      <= left  and max  <= HIGH A and
     min - 1  <= right and 0    <= min and
     right > left =>
       containsElementGEQ(A,left,max,pivot) and
       containsElementLEQ(A,min,right,pivot) and
     max > left  => sorted(A,left,max) and
     min < right => sorted(A,min,right) and
     forall  j : nat &
       right < j and j < left => (([ A  j ]) . key)  = pivot and
     partitioned(A,min,max,left,right,pivot)
     left >= right and
   INT(max) > left =>
     max <= HIGH A and 0  <= left and
     sorted(A,left,max) =>
       right    <= max   and left <= 1  + max and
       min      <= left  and max  <= HIGH A and
       min - 1  <= right and 0    <= min and
       right > left =>
         containsElementGEQ(A,left,max,pivot) and
         containsElementLEQ(A,min,right,pivot) and
       max > left  => sorted(A,left,max) and
       min < right => sorted(A,min,right) and
       forall  j : nat &
         right < j and j < left => (([ A  j ]) . key)  = pivot and
       partitioned(A,min,max,left,right,pivot) and
       left >= right and
     left <= max

Proof obligation in lines 241:13-265:69:
   right <= max      and left <= 1  + max and
   min <= left       and max <= HIGH A and
   min - 1  <= right and 0  <= min and
   right > left =>
     containsElementGEQ(A,left,max,pivot) and
     containsElementLEQ(A,min,right,pivot) and
   max > left  => sorted(A,left,max) and
   min < right => sorted(A,min,right) and
   forall  j : nat &
     right < j and j < left => (([ A  j ]) . key)  = pivot and
   partitioned(A,min,max,left,right,pivot) and
```

```
   left >= right
=> sorted(A,min,max)

Proof obligation in lines 279:9-281:
  false => HIGH A >= 0

Proof obligation in lines 279:9-281:44:
   HIGH A >= 0
=> 0  <= HIGH A and 0  <= 0  and
   sorted(A, 0, HIGH A) =>
     sorted(A, 0, HIGH A) and
     HIGH A <= HIGH A

Proof obligation in lines 279:9-281:44:
  sorted(A, 0, HIGH A) => true
```

B.3. Quicksort: variation 1

```
(*{ entry choose_pivot
    pre   0 <= min and min <= max and max <= HIGH(A)
    post  0 <= min and min <= max and max <= HIGH(A) and
          left  = min and right = max and
          containsElementGEQ (A, left, max, pivot) and
          containsElementLEQ (A, min, right, pivot)    }*)
(* ---------------------------------------------------- *)
(* Choosing an element, which determines the partition: *)
(* here, the middle of three is chosen                  *)
(* ---------------------------------------------------- *)
IF A[min].key > A[max].key THEN
    IF A[max].key > A[(min + max) DIV 2].key THEN
        pivot := A[max].key;
    ELSE
        IF A[min].key > A[(min + max) DIV 2].key THEN
            pivot := A[(min + max) DIV 2].key;
        ELSE
            pivot := A[min].key;
        END
    END
ELSE
    IF A[(min + max) DIV 2].key > A[max].key THEN
        pivot := A[max].key;
    ELSE
        IF A[min].key > A[(min + max) DIV 2].key THEN
            pivot := A[min].key;
        ELSE
            pivot := A[(min + max) DIV 2].key;
        END
    END
END;
```

```
        left  := min;
        right := max;
        (*{ exit choose_pivot }*);
```

B.4. Quicksort: variation 2

```
        (*{ entry Sort_body
            pre   0 <= min and min <= max and max <= HIGH(A)
            post  sorted (A, min, max) }*)

        IF (max - min > 2) THEN


        ELSE
            (*{ entry swap_sort
                pre   0   <= min and max - min <= 2 and
                      min <= max and max <= HIGH(A)

                post  0   <= min and max - min <= 2 and
                      min <= max and max <= HIGH(A) and
                      A(min).key <= A(max).key }*)
            (* ----------------------------------------------- *)
            (* For an Array with one or two element, recursion *)
            (* is not efficient. A conditional swap takes less *)
            (* time.                                           *)
            (* ----------------------------------------------- *)
            IF (A[min].key > A[max].key) THEN
                swap   := A[min];
                A[min] := A[max];
                A[max] := swap;
            END;
            (*{ exit swap_sort }*)
        END;


        (*{ exit Sort_body }*)
    END Sort;
```

B.5. Quicksort: variation 3

```
MODULE QuickSortModule;

(* ------------------------------------------------------------ *)
(* QuickSort                                                    *)
(* ------------------------------------------------------------ *)
PROCEDURE QuickSort (VAR A: ARRAY OF Element);

    (* -------------------------------------------------------- *)
    (* BubbleSort                                               *)
    (* -------------------------------------------------------- *)
```

```
(* Sorts the [l, r]-part of the array A using the           *)
(* BubbleSort-Algorithm.                                    *)
(* -------------------------------------------------------- *)
PROCEDURE BubbleSort (l, r : CARDINAL);
BEGIN
    (*{ entry BubbleSort_body
        pre   HIGH(A) >= r and r >= l and l >= 0
        post  sorted (A, l, r) }*)

    (*{ exit BubbleSort_body }*)
END BubbleSort;


(* -------------------------------------------------------- *)
(* Sort                                                     *)
(* -------------------------------------------------------- *)
PROCEDURE Sort (min, max : CARDINAL);

BEGIN
    (*{ entry Sort_body
        pre   0 <= min and min <= max and max <= HIGH(A)
        post  sorted (A, min, max) }*)

    IF (max - min > 10) THEN

    ELSE
        BubbleSort (min, max)
    END;
    (*{ exit Sort_body }*)
END Sort;
```

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 96-03

96-03	C. Lindig, G. Snelting	Modularization of Legacy Code Based on Mathematical Concept Analysis
96-04	J. Adámek, J. Koslowski, V. Pollara, W. Struckmann	Workshop Domains II (Proceedings)
96-05	F.-J. Grosch	A Syntactic Approach to Structure Generativity
96-06	E. H. A. Gerbracht, W. Struckmann	Zur Diskussion elementarer Funktionen aus algorithmischer Sicht
96-07	H.-D. Ehrich	Object Specification
97-01	A. Zeller	Versioning Software Systems through Concept Descriptions
97-02	K. Neumann, R. Müller	Implementierung von Assertions durch Oracle7-Trigger
97-03	G. Denker, P. Hartel	Troll – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics
97-04	F.-J. Grosch	M - eine typisierte, funktionale Sprache für das Programmieren-im-Grossen
97-05	J. Küster Filipe	Putting Synchronous and Asynchronous Object Modules together: an Event-Based Model for Concurrent Composition
97-06	J. Küster Filipe	A categorical Hiding Mechanism for Concurrent Object Systems
97-07	G. Snelting, U. Grottker, M. Goldapp	VALSOFT Abschlussbericht
98-01	J. Krinke, G. Snelting	Validation of Measurement Software as an application of Slicing and Constraint Solving
98-02	S. Petri, M. Bolz, H. Langendörfer	Transparent Migration and Rollback for Unmodified Applications in Workstation Clusters
98-03	M. Cohrs, E. H. A. Gerbracht, W. Struckmann	DISKUS - Ein Programm zur symbolischen Diskussion reeller elementarer Funktionen
98-04	C. Lindig	Analyse von Softwarevarianten
98-05	Gregor Snelting, Frank Tip	Reengineering Class Hierarchies Using Concept Analysis
98-06	Juliana Küster Filipe	On a Distributed Temporal Logic for Modular Object Systems
98-07	J. Schönwälder, M. Bolz, S. Mertens, J. Quittek, A. Kind, J. Nicklisch	SMX - Script MIB Extensibility Protocol Version 1.0
98-08	C. Heimann, S. Lauterbach, T. Förster	Entwurf und Implementierung eines verteilten Ansatzes zur Lösung langrechnender Optimierungsprobleme aus dem Bereich der Ingenieurwissenschaften
99-01	A. Zeller	Yesterday, my program worked. Today, it does not. Why?
99-02	P. Niebert	A Temporal Logic for the Specification and Verification of Distributed Behaviour
99-03	S. Eckstein, K. Neumann	Konzeptioneller Entwurf mit der Unified Modeling Language
99-04	T. Gehrke, A. Rensink	A Mobile Calculus with Data
00-01	T. Kaiser, B. Fischer, W. Struckmann	The Modula-2 Proving System MOPS