Formal Verification of Function Blocks for Industrial Process Automation

Wolfgang A. Halang and Bernd J. Krämer
Faculty of Electrical Engineering
FernUniversität
58084 Hagen, Germany
{wolfgang.halang|bernd.kraemer}@fernuni-hagen.de

Sias Mostert
Department of Electrical and Electronic Engineering
University of Stellenbosch
7600 Stellenbosch, South Africa
mostert@eng.sun.ac.za

Abstract
This work was carried out in the framework of the German-South African scientific co-operation project 39.6.11.A.6. E Formal Verification of the VDI/VDE 3696 Function Block Set. Its goal is to formally specify the function blocks defined in the German engineering guideline VDI/VDE 3696 Manufacturer Independent Configuration of Digital Control Systems, and to formally prove implementations correct against these specifications. Thus, the dependability of safety related process control software is to be enhanced. In this report, several typical function blocks as selected from the guideline VDI/VDE 3696 are formally specified in the specification language OBJ and rigorously verified employing Higher Order Logic.

1 Introduction

The quality of software intensive systems primarily depends on the quality of their specifications and the software development process used for their production [49]. The main purpose of specifications is to define precisely the intended operation of a system in its operational environment. Specifications are subject to communication and contractual regulations between customers, developers, and certification authorities. Specifications define prescriptions for subsequent design, implementation and validation tasks, and they facilitate system evolution. Owing to the enormous costs resulting from erroneous and incomplete requirement specifications and designs [8], system developers are more and more put under pressure to replace or supplement informal specification and design techniques mostly used in practice by more reliable methods. Above all, this applies to safety and security sensitive applications, which have to meet high demands of robustness, reliability and correctness. Safety critical systems must often function the first time they are used, because they can neither be tested in their real environments and nor can they be phased in [41, 42]. The German catalogue of safety and security criteria for information technology [53], for example, assigns system components processing highly sensitive data to quality classes, for which security policies and system functions have to be formally specified and verified.

Formal specifications rely on some underlying mathematical logic and, thus, have well defined semantics. Their advantages over informal specification techniques as mostly used in practice
are abstraction, precision, structuring, and manipulability. Abstraction improves the comprehensi- mility of specifications and gives scope for design decisions and alternative implementations. Precision is necessary to avoid incorrect or ambiguous interpretations and to exclude inconsistencies, as inconsistent requirements cannot be satisfied by any implementation. Structuring concepts help in mastering the complexity of a development task and they support re-usability and adaptability of programmable components. Manipulability refers to the potential of specifications to enable the use of computers upon their construction, verification, validation, and systematic re-use. Essential characteristics and benefits of formal methods are further elaborated in [51], and several myths on formal methods are rectified in [29].

This report presents a rigorous, formal specification based development process for standard and application specific function blocks used in safety critical control and automation applications. The overall development process is introduced in the next section. All steps it includes and associated techniques and tools are then illustrated in subsequent sections using three examples. A discussion of the techniques presented and their relation to other software development paradigms such as automatic program construction and transformational software development conclude this report.

2 Specification Based Function Block Development

The development of application programs for Programmable Logic Controllers (PLCs) is largely a software development process including typical steps such as requirements capture, requirements specification, coarse and fine grain design, and implementation. These construction steps are usually interleaved with validation activities relying on reviews, informal plausibility analysis, consistency checking, and program testing. Although being advocated for in programming for more than three decades, rigorous correctness proofs are still rare in industry, but a growing awareness of their importance is witnessed, for example, in [30] and [7].

Formal semantics of graphically represented function block networks were introduced in [28] based on the formal specification language ṢORAS\(^1\) [39]. A graphical design method for constructing function block diagrams from a collection of pre-defined function blocks was presented and analysis techniques and tools applying to ṢORAS specifications were used to reason about dynamic properties of complex process control applications. In the sequel, we are concerned with the development of single standard functions blocks as they are used for vendor independent configuration of process control applications. Figure 1 depicts a rigorous function block development process which heavily relies on formal methods. It includes several major steps:

1. **Requirements specification** — produces a formal description of critical requirements of a function block and critical assumptions about the behaviour of its operational environment. This step will be illustrated in Section 3.

2. **Design** — leads to an architectural view in terms of pre-defined component blocks and their connections; it also provides a black-box view of the functional behaviour of individual function blocks without referring to their internal structure. In Section 4 the example introduced earlier is extended by a formal behaviour specification.

3. **Specification verification** — is concerned with the inner consistency and completeness of the design specification and its conformity with critical requirements; inductive proof techniques and case analysis serve as basic mechanisms to perform this step.

4. **Specification testing** — requires executable specifications; they provide implementation independent prototypes whose behaviour can be observed and checked against the ex-

\(^1\) ṢORAS is a registered trademark of GMD.
pectations of developers or interested third parties to gain further confidence about their adequacy. Instead of operating on concrete data, symbolic testing uses terms built from the operations of the specification as symbolic inputs and produces other terms as symbolic outputs using the axioms of the specification as re-write rules. Section 6 illustrates this step.

5. *Program construction* — refers to the manual derivation of programs from validated specifications. The program text implementing the behaviour of a function block is written in Structured Text (ST). The syntax of a subset of ST and an example are contained in Section 7.1.

6. *Program verification* — is enabled, because the ST code is decorated with pre- and post-conditions specifying program behaviour in terms of expected properties of input data and the effect of computations on output data and program states. In Section 7 Dijkstra’s weakest pre-condition approach is applied to the sample program designed in the previous step. Hoare style proof rules, which are closely related to the weakest pre-condition technique, are applied to verify a second example which is introduced in Section 8.

7. *Program testing* — is a classical and well known validation technique. It can be used to verify simple standard function blocks which have a finite and manageable number of states by exhaustive testing. Function blocks implementing $n$-ary Boolean functions with $n$ an arbitrary finite number are typical candidates for verification by exhaustive program testing. This technique is so obvious that there is no need to illustrate it here.

![Diagram of the rigorous function block development process](image_url)

Figure 1: Rigorous function block development process
The algebraic specification language OBJ [20] and its support system OBJ3 system [23] play a major rôle in the proposed process. OBJ is employed for formalising requirements, capturing designs, and stating pre- and post-conditions of ST programs. OBJ3 is used to perform all manipulations of OBJ specifications necessary to reason about properties of specifications and programs, verify their correctness, or execute them symbolically. Both language and system were successfully applied in other experiments to specify and verify hardware [22, 48] and software [17]. In Section 9 we then present an alternative approach to formal development of function blocks. A simple timer is formally developed using a higher order logic formalism [24] to specify its intended behaviour and the HOL verifier [25] to prove the correctness of its implementation.

It is obvious that, compared to a conventional development process focusing on program construction and program testing only, some extra effort is necessary to develop formal requirements, designs, and specifications and relate them by formal proof techniques. But this effort is justified by a number of reasons: standard function blocks will be used in hundreds or thousands of control applications and their correct operation is often crucial to satisfy hard safety and reliability requirements of entire systems. Specification development and verification provide increased certainty about the consistency between specification and program, contribute to a better understanding of requirements and designs, and reduce the costs for error detection and removal that would otherwise occur in later development stages. Formal interface specifications also enhance the systematic re-use of function blocks [52] and verified interface specifications extend the re-use of designs to the re-use of associated proofs and, thus, support modular proof techniques [22]. An elaborate discussion of the value of formal methods in safety critical software development is, for example, contained in [5, 46].

3 Formalising Requirements

The process of developing formal specifications from informally described requirements includes three major steps [26, 14]:

1. solicitate and understand user requirements,
2. translate these requirements into a formal specification, and
3. try to show that the specification is an accurate model of the user requirements.

The problem of effective requirements elicitation, analysis, and formalisation is, in general, still a major subject of research in the area of requirements engineering (see, e.g., [43, 19, 18].) We shall not further discuss this area and simply assume that appropriate methods are available for a selected application domain. Systematic approaches to specification formalisation are described in [14] for sequential systems and in [40, 45] for concurrent and distributed systems.

3.1 Problem Statement

Our first function block example is taken from the guideline [50] defining standard function blocks to be used in vendor independent descriptions of process control programs. This function block, called nonlin, approximates a non-linear static characteristic \( f \) by a polygon \( f' \), which is given by \( n \) base points defining the edges of the polygon. Function block \( \text{nonlin} \), once instantiated with an arbitrary finite positive number \( n \), receives \( 2n + 1 \) more inputs to set the \( n \) base point co-ordinates and one abscissa, and provides the corresponding ordinate as output. The functionality of \( \text{nonlin} \) is shown below and examples for \( f \) and \( f' \) are given in Figure 2.
Table 1: Parameters of function block nonlin

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Meaning</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Input Value</td>
<td>Num</td>
</tr>
<tr>
<td>N</td>
<td>Number of Base Points</td>
<td>Num</td>
</tr>
<tr>
<td>(some positive integer n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U1</td>
<td>U-Value of Base Point 1</td>
<td>Num</td>
</tr>
<tr>
<td>V1</td>
<td>V-Value of Base Point 1</td>
<td>Num</td>
</tr>
<tr>
<td>U2</td>
<td>U-Value of Base Point 2</td>
<td>Num</td>
</tr>
<tr>
<td>V2</td>
<td>V-Value of Base Point 2</td>
<td>Num</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Un</td>
<td>U-Value of Base Point n</td>
<td>Num</td>
</tr>
<tr>
<td>Vn</td>
<td>V-Value of Base Point n</td>
<td>Num</td>
</tr>
</tbody>
</table>

Outputs

| V       | Result Value                   | Num  |

Figure 2: Non-linear function and its polygon approximation
As usual the function block is regularly invoked in every basic cycle. Every time a new value is expected on input \( U \) and the corresponding output \( f'(U) \) occurs on output \( V \). For \( U < U1 \) it is required that \( V \) becomes \( V1 \) and for \( U > Un \) the output value is \( Vn \).

### 3.2 Requirements Specifications

A software component can be characterised by different kinds of requirements including functional, qualitative, technical, and economical requirements [14]. Functional requirements deal with the behaviour of a system and its environment. Qualitative requirements may define performance and timing attributes, and consider safety, reliability or security requirements. Technical requirements might concern the implementation environment, tools to be used in the development process, and interfaces to existing components to be observed. Economical requirements mainly deal with cost/benefit aspects, but also include legal restriction.

The focus of interest for the application domain considered here is on functional requirements. Often they include safety requirements, but low complexity of individual function blocks obviates the need for their separation. In the sequel we shall particularly deal with requirements about

- inputs to function blocks and constraints on these inputs,
- functions performed by function blocks, and
- outputs and other effects of the function blocks.

But before we can formally state these functional requirements, we have to translate the concepts of the problem domain into appropriate constructs of the chosen specification formalism OBJ. We shall now demonstrate this step for function block \texttt{nonlin}.

### 3.3 OBJ

The basic building blocks of OBJ are two kinds of modules: objects and theories. The difference between them is largely a semantic one\(^2\), which is of no interest here as we only use objects. Modules are named entities that encapsulate data of specific sorts. These data can only be manipulated through the operations provided by a module. The sorts and operations of a module are introduced by declarations which are preceded by the keywords \texttt{sort} and \texttt{op}, respectively. A sort \( s \) may be a subset or another sort \( s' \), written \texttt{subsort \( s < s' \))} in OBJ. Intuitively, this means that data of sort \( s \) can be used in an argument position of an operation that expects data of sort \( s' \). In this case the OBJ3 interpreter implicitly coerces data of sort \( s \) to data of sort \( s' \). Operation declarations have the form \texttt{op \( f : s_1 \ldots s_n \rightarrow s \))}, where \( f \) denotes the name of the operation, \( s_1, \ldots, s_n \) denote the sorts of arguments, and \( s \) represents the sort of results produced by \( f \). For \( n = 0 \), \( f \) declares a constant of sort \( s \). The properties of operations are implicitly defined by mutually relating the behaviour of different operations that are visible within a module. This is achieved by means of equations and conditional equations. Equations have the form \texttt{eq \( l = r \))}, where \( l \) and \( r \) are terms formed from operations and variables of a module; both terms must have coercible sorts. 

defterm Conditional equations are written \texttt{cq \( l = r \) IF \( c \))}, and \( c \) must be a term of sort \texttt{Bool}. Variables contained in (conditional) equations have to be declared prior to their use. Variable declarations are written \texttt{var \( i : s \)) or \texttt{vars \( i_1 \ldots i_m : s \))}. Modules can also import specifications of other modules in different ways\(^3\). This is achieved by listing a module name after a specific keyword such as \texttt{EXTENDING} or \texttt{PROTECTING}.

---

\(^2\)Objects denote initial order sorted algebras, while theories denote varieties.

\(^3\)The semantic differences between the different forms of specification imports allowed in OBJ are explained in [20].

---

6
OBJ provides a collection of built-in objects including \texttt{BOOL}, \texttt{INT}, and \texttt{PROC}. They specify the laws of Boolean algebra, integer arithmetic, and propositional calculus, respectively. Furthermore, there are two comparison operators \("\_\_!=\_\_\)" and \("\_\_=/\_\_\)" with the signature \("S S \rightarrow \text{Bool}\) and a conditional choice construct \texttt{if\_then\_else\_fi} with the signature \("\text{if\_then\_else\_fi}: \text{Bool} S S \rightarrow S\) for every sort \(S\). The operators denote syntactic equality and inequality of two terms, respectively.

Object \texttt{SIMPLE-NAT} below illustrates the OBJ style of specification.

\begin{verbatim}
obj SIMPLE-NAT is
  sorts Nat NzNat Zero .
  subsorts Zero NzNat < Nat .
  op 0 : \rightarrow Zero .
  op s_ : Nat \rightarrow NzNat .
  vars N M : Nat . vars N' M' : NzNat .
  eq N + 0 = N .
  eq (s N) + (s M) = s s (N + M) .
  eq d(0,N) = N .
  eq d(s N, s M) = d(N,M) .
endo
\end{verbatim}

The keywords enclosed in square brackets associate specific mathematical properties such as associativity or commutativity to binary operators and, thus, control the OBJ3 interpreter to apply corresponding re-write rules if necessary.

### 3.4 Functional Requirements of \texttt{nonlin}

Functional behaviour of a function block is naturally represented by OBJ operations, while types of input and output data are represented by sorts. As function blocks are regularly invoked, they synchronously process streams \(\alpha(0),\alpha(1),\ldots\) of time varying data. Thus, each input and output of sort \(S\) can be modelled by a function \(\alpha\) from \texttt{Time} to \(S\).

As the number of base points of \texttt{nonlin} instances depends on the value \(n\) of input \(N\) upon function block instantiation, the base point co-ordinates are modelled by two functions \texttt{ui} and \texttt{vi} and parameterised by an index ranging over the set \(\{1,\ldots,n\}\).

\begin{verbatim}
obj INPUT is
  protecting TIME .
  protecting NUM .
  op u : Time \rightarrow \text{Num} .
  op n : \rightarrow Nat .
  op ui : Nat Time \rightarrow \text{Num} .
  op vi : Nat Time \rightarrow \text{Num} .
endo
\end{verbatim}

\begin{verbatim}
obj OUTPUT is
  protecting NUM .
  protecting TIME .
  op v : Time \rightarrow \text{Num} .
endo
\end{verbatim}
Object **NUM** defines rational numbers with appropriate arithmetic and comparison operations on them. Its specification is adapted from object **RAT** contained in the library of sample specifications provided with the OBJ distribution tape. Object **TIME** simply provides distinguishable clock ticks:

```OBJ
obj TIME is sort Time .
  op 0    : -> Time .
  op tick : Time -> Time .
endo
```

The sorts used in operation declarations restrict the domains of admissible input and output values and, thus, partly protect a function block from unintended uses, because only those connections are meaningful that connect inputs and outputs of related sorts. For instance, an environment of use that admits the provision of negative integers on input \( \mathbb{N} \) would be rejected at construction time already due to sort inconsistencies [28].

Functional requirements explicitly expressed in the problem statement given before concern the treatment of out-of-range accesses:\(^4\)

\[
\forall u : \text{Num} \bullet u < u_1 \Rightarrow v = v_1
\]
\[
\forall u : \text{Num} \bullet u > u_n \Rightarrow v = v_n
\]

This can be stated as follows in OBJ:

```OBJ
obj REQ-OUT-OF-RANGE is
  protecting INPUT .
  protecting OUTPUT .
  var T : Time .
  cq v(T) = v1(1,T) if u(T) < u1(1,T) .
  cq v(T) = v1(n,T) if u(T) > u1(n,T) .
endo
```

For each input \( u \) respecting the range of well defined base points we must ensure that the corresponding output value \( v \) is \( f'(u) \). Instead of defining \( f' \) explicitly, we observe that it is sufficient to consider just that section of the polygon whose \( u \)-co-ordinates enclose input \( u \). We rely on the 2-point definition of straight lines. To avoid undefined results, we have to require first that all \( u \)-co-ordinates are presented in ascending order, i.e., for all indices \( i \) between 1 and \( n \) the value of \( u_i \) must be smaller than the value of the subsequent \( u \)-co-ordinate, which is stated more formally as follows:

\[
\forall i : \text{Nat} \bullet (1 \leq i < n \bullet u_i < u_{i+1})
\]

Now we can formally state the third requirement for output \( v \) as follows:

\[
\forall u : \text{Num} \text{ such that } \exists i \in \{1, \ldots, n-1\} \text{ and } u_i \leq u \leq u_{i+1} \text{ holds}.
\]

Point \((u,v)\) must lie on the straight line connecting the points \((u_1,v_1)\) and \((u_{i+1},v_{i+1})\), which is defined by

\[
\frac{v_{i+1} - v_i}{u_{i+1} - u_i} = \frac{v - v_i}{u - u_i}
\]

\(^4\)We use the convention that lower case letters written in italics, such as \( u \), denote values on inputs denoted by corresponding capital letters, such as \( U \).
Hence, the formalised requirement is all
\[
\forall u : \text{Num} \quad \bullet \quad (\exists i \in \{1, \ldots, n - 1\} \quad \bullet \quad u_i \leq u \leq u_{i+1} \quad \And \quad v = \frac{u_{i+1} - u_i}{u_{i+1} - u_i}(u - u_i) + u_i).
\]

Requirement 3 ensures that \( v \) is well defined for all base points.

To validate this requirement, we perform a plausibility check with the base point co-ordinates. They have the special property that they both lie on \( f \) and \( f' \). Let \( u = u_i \) for some \( i \in \{1, \ldots, n - 1\} \). Then we observe that \( v = u_i \). For \( i = n - 1 \) and \( u = u_n \) we obtain \( v = v_n \).

In OBJ we formulate Requirement 4 as follows:

```plaintext
obj REQ-V is
  protecting REQ-OUT-OF-RANGE .
  var T : Time . var I : Nat .
  var U : Num .
  cq v(T) = (((vi(I + 1,T) + - vi(I,T)) / (ui(I + 1,T) + - ui(I,T))) * (U + - ui(I,T))) + vi(I,T) .
  if *** Condition of Req. 3 and 4
  1 <= n and I < n and
  ui(I,T) < ui(I + 1,T) and *** Req. 3
  *** Condition of Req. 4
  ui(I,T) <= u(T) and u(T) <= ui(I + 1,T) .
endo
```

We notice that the conditions in the three conditional equations
\[
\begin{align*}
  u(T) &< ui(I + 1, T) \\
  u(T) &> ui(n, T) \\
  ui(I, T) &\leq u(T) \text{ and } u(T) \leq ui(I + 1, T)
\end{align*}
\]
given for function \( v \) in objects REQ-OUT-OF-RANGE and REQ-V are logically complete. We also recognise that the definition of function \( v \) in object REQ-V is not constructive in the sense that it would define a computation rule for index \( i \). This is subject of the design step described in the following section.

4 Design Specification of Function Block nonlin

Design specifications will define abstract computation rules for all operations computing the outputs of a function block. We use a property oriented style of operation definition using equations and conditional equations to relate operation applications to each other. To the resulting specifications we can apply both specification verification and symbolic testing techniques to increase our confidence in the adequacy and correctness of our design before taking further construction steps.

Again we demonstrate this step with function block nonlin. First, we give a definition of function \( f' \), which takes two indices \( i \) and \( j \), the actual time \( t \), and \( u \) as inputs. Starting with the index values \( 1 \) and \( n \), we recursively narrow the range of consideration with respect to \( f' \) until the proper index \( i \) satisfying the condition of Requirement 4 is found. The algorithm defined in object fpolygon is illustrated in Figure 3 for the function and base points given in Figure 2.
Figure 3: Illustration of the recursive approximation algorithm defined in object `f_polygon`
obj FPOLYGON is
  protecting INPUT .
  protecting OUTPUT .
op f' : Nat Nat Time Num -> Num .
vars I J : Nat .
var T : Time . var U : Num .
cq f'(I,J,T,U) = (((vi(I + 1,T) + - vi(I,T)) / 
  (ui(I + 1,T) + - ui(I,T))) * 
  (U + - ui(I,T)))) + vi(I,T)
  if J == I + 1 .
cq f'(I,J,T,U) = f'(I + 1,J - 1,T,U)
  if (U >= ui(I + 1,T)) and (ui(p(J),T) >= U) 
  and J > I + 2 .
cq f'(I,J,T,U) = f'(I + 1,J,T,U)
  if (U >= ui(I + 1,T)) and (U >= ui(p(J),T)) 
  and J > I + 1 .
cq f'(I,J,T,U) = f'(I,J - 1,T,U)
  if (ui(I + 1,T) >= U) and (ui(p(J),T) >= U) 
  and J > s(0) .
endo
Based on this definition, a computation rule for v can be given, which concludes our design specification.

obj V is
  protecting FPOLYGON .
var T : Time .
cq v(T) = f'(1,n,T,u(T))
  if u(T) >= ui(1,T) and ui(n,T) >= u(T) .
cq v(T) = vi(1,T) if u(T) < ui(1,T) .
cq v(T) = vi(n,T) if u(T) > ui(n,T) .
endo

5 Specification Verification

In the previous section we have seen that OBJ, like other algebraic specification formalisms, allows the user to specify software components independently of any concrete representation of data and without reference to any particular implementation of operations. An important asset of these formalisms is their unique semantics and the underlying equational calculus [16, pp. 108–137]. This calculus admits proving theorems and equations from specifications and deriving terms from terms using equations as re-write rules. Both equational reasoning in theorem provers and equational computation in specification languages rely on term re-writing mechanisms. The central idea of re-writing is to impose a direction on the use of equations in proofs or symbolic computations.

OBJ3 is one of several re-writing systems that have been designed to compute and employ sets of re-write rules from equational axioms. Term re-write rules are used to convert specification expressions (terms) into their normal forms. To guarantee the uniqueness of such a normal form, i.e., to ensure that this form is independent of the rule application strategy, a set of re-write rules is required to be confluent. To guarantee that equational computation succeeds, a re-write rule system must further be terminating. Many re-write rule systems employ a variant of the
Knuth-Bendix completion algorithm to generate, if possible, a confluent and terminating re-write rule system [38]. OBJ3 directly interprets (conditional) equations as re-write rules from left to right, assuming that the set of rules thus obtained is confluent and terminating [37]. Moreover, it requires that all variables occurring on the right-hand side of an equation and in the condition of a conditional equation occur in the left-hand side, too. Thus, upon rule invocation all variables are bound. OBJ3’s term re-writing facility can be illustrated by a (slightly simplified) trace of the test “d(1, 3) = d(3, 1)” challenging the associativity of distance operation d defined in object SIMPLE-NAT in Section 3.3. OBJ> is the system prompt:

```
OBJ> set trace on.
OBJ> red d(1,3) == d(3,1).
reduce in SIMPLE-NAT : d(1,3) == d(3,1)
*************** rule *** d(s N,s M) = d(N,M)
d(s 0,s (s (s 0))) ---> d(0,s (s 0))
*************** rule *** d(0,N) = N
d(0,s (s 0)) ---> s (s 0)
*************** rule *** d(s N,s M) = d(N,M)
d(s (s (s 0)),s 0) ---> d(s (s 0),0)
*************** rule *** d(0,N) = N
d(s (s 0),0) ---> s (s 0)
*************** rule
s (s 0) == s (s 0) ---> true
rewrites : 5
result Bool : true
```

We observe that OBJ3 recursively applies one of the equations in the specification of SIMPLE-NAT as re-write rule to reduce the input term to its simplest normal form, which is true for this example.

5.1 Inductive Proofs with Equations

This section applies proof techniques developed in [22] to formally verify that a design specification written in OBJ satisfies its requirements and further propositions about conjectured properties of the design. These requirements or propositions are expressed as equational theorems in OBJ. The goal of a proof is to conclude the theorem in question from the given specification and a set of premisses, which are again equations. A proof consists of a sequence of re-write steps using premisses and specification equations. The re-write rule interpreter of OBJ3 can be employed to perform routine work automatically and the modularity concepts of the language help structuring user directed proofs in modules and hierarchies.

A powerful proof strategy for algebraic specifications of abstract data types is structural induction. The structural induction principle is set up analogously to the complete induction over integers. For showing that a property P is applicable to all data of a type, we must show that P applies to all constructor operations of an object. Intuitively, constructor operations are those operations that build up data of a sort. For example, constant 0 and the successor operation s of object SIMPLE-NAT are constructors, while d and _+_- are only derived operations whose behaviour must be specified in terms of the constructor operations. The theoretical basis for this proof technique and further examples are presented in [22].
5.2 Correctness of Function \( f' \)

To demonstrate that the definitions of the operations \( v \) and \( f' \) presented in objects \( V \) and \( f\text{polygon} \), respectively, are correct, we use induction over the length \( n \) of the polygon. It is sufficient to show that the operations are correctly defined for an elementary polygon of length one, which provides the induction basis, and then demonstrate that they are correct for a polygon of length \( n + 1 \) under the assumption that they are correct for some arbitrary length \( n \).

First we consider the induction base. As we want to perform ground term deduction, we consider the variables in requirement specifications as arbitrary constants. The justification for this approach is given in [22]. We also assume that Requirement 3 is satisfied.

```
=================================================================
obj VARS-BASE is extending V .
  op t : -> Time .
  op u : -> Num .
  eq n = 2 .
  eq ui(1,t) < ui(n,t) = true .  *** Req. 3
endo
=================================================================
```

For operation \( v \) we have to distinguish between three cases. Note that the first two properties proved below hold for an arbitrary number \( n > 1 \) of base points, too.

1. \( u < u_1 \):
   As the requirement to be verified is expressed as a conditional equation, we must assume the condition to be true. Hence we assert:

```
obj BASE-UNDER is extending VARS-BASE .
  eq u(t) < ui(1,t) = true .  
endo
```

and attempt to verify \( v(t) \):

```
OBJ> reduce in BASE-UNDER : v(t)
result v(s(0))
```

Note that OBJ3 generates the standard representation of natural numbers in terms of constant 0 and the successor operation \( s \).

2. \( u > u_n \):

```
obj BASE-OVER is extending VARS-BASE .
  eq u(t) > ui(n,t) = true .
endo
```

```
OBJ> reduce in BASE-OVER : v(t)
result v(s(s(0)))
```

13
3. \( u_1 \leq u \leq u_n \):

\[
\text{obj BASE-IN is extending VARS-BASE .} \\
\text{eq } u(t) >= u(1,t) \text{ and } u(n,t) >= u(t) .
\]
\text{end}

\[=\]

\[\text{reduce in BASE : } v(t) \]
\text{rewrites: 56}
\text{result } ((v_i(s(0),t) * (-u_i(s(0),t) + u_i(s(s(0),t)))) +
(u_i(s(s(0),t)) +
((u_i(s(s(0),t)) +
-((u_i(s(0,t) * v_i(s(s(0),t)))) +
-((u_i(s(0),t) * v_i(s(s(0),t)))) /
(-u_i(s(0),t) + u_i(s(s(0),t)))))
\]

\[=\]

which is equals to the expected result
\[
((v_i(n,t) - v_i(1,t)) / (u_i(n,t) - u_i(1,t))) *
(u_i(t) - u_i(1,t)) + v_i(1,t). 
\]

This reduction step is also an argument for the correctness of \( f' \) as it is needed to perform the reduction of \( v \).

The induction step is based on the assumption that the correctness of \( v \) and \( f' \) has been shown for some arbitrary \( n \geq 2 \). First we re-define some constants representing arbitrary but fixed values for relevant variables of the specification.

\[
\text{obj VARS-ASS is extending V .}
\]
\text{op } t \text{ : } \rightarrow \text{Time .}
\text{op } i \text{ : } \rightarrow \text{Nat .}
\text{op } u \text{ : } \rightarrow \text{Num .}
\text{eq } n >= 2 .
\text{end}

As the correctness of \( v \) for out-of-range accesses has been shown before, we consider only inputs to \( U \) that range between \( u_1 \) and \( u_n \) and assert the correctness of \( f' \) under the following assumptions:

\[
\text{obj ASS is extending VARS-ASS .}
\]
\text{eq } 1 <= i = \text{true .}
\text{eq } i < n = \text{true .}
\text{eq } u_i(1,t) <= u(t) = \text{true .}
\text{eq } u(t) <= u_i(n,t) = \text{true .}
\text{end}

Now we have to show that \( f' \) is also correctly defined for a polygon that is extended by one more base point obeying Requirement 3, i.e., \( u_n < u_{n+1} \). Two cases must be distinguished:

1. \( u_1 \leq u < u_n \):

This case is satisfied trivially, because \( u \) lies on that part of the polygon for which the property holds by the induction assumption.
2. \( u_n \leq u \leq u_{n+1} \):

That is, we assume that:

\[
\text{obj ASS2 is extending VARS-ASS .}
\]

\[
\text{eq n} \leq i < n + 1 = \text{true .}
\]

\[
\text{eq ui(n,t)} \leq u(t) = \text{true .}
\]

\[
\text{eq u(t)} \leq ui(n + 1,t) = \text{true .}
\]

\text{endo}

and obtain by reduction:

\[
\text{reduce in ASS2 : } f'(n,n + 1,t,u(t))
\]

\text{rewrites: 10}

\text{result Num:}

\[
((\text{vi}(n,t) \cdot (\text{-ui}(n,t) + \text{ui}(s(n),t))) +

((\text{-ui}(n,t) + (\text{u(t)})) \cdot (\text{-vi}(n,t) +

\text{vi}(s(n),t)))) / (\text{-ui}(n,t) + \text{ui}(s(n),t))
\]

\text{------------------------}

which reads as expected after some transformations:

\[
((\text{vi}(n+1,t) - \text{vi}(n,t)) / (\text{ui}(n+1,t) - \text{ui}(n,t))) \cdot

(\text{u(t)} - \text{ui}(n,t)) + \text{vi}(n,t).
\]

6 Specification Testing

Developing and formally verifying a specification for a function block are important steps in the PLC software development process. We must understand, however, that correctness proofs just relate mathematical abstractions of a given problem and, hence, cannot guarantee the adequacy of given requirements or design specifications. Symbolic testing based on executable specifications is, therefore, a useful supplement to formal verification, because the functional behavior of an abstract function block design can be observed under specific application conditions by using the specification itself as a test object or prototype. This prototyping approach helps to detect design errors at early stages that cannot be found by formal analysis and verification [36].

Algebraic specifications can be tested by re-writing a term into its normal form. In this way, the behavior of the system to be developed can be observed and evaluated in early development phases to correct design errors before further expensive development steps are taken. In comparison to program testing, specification testing has the additional advantage of testing entire classes of programs rather than exactly one instance of a large set of implementations. Design specifications can further be used to systematically derive test cases and comparative values for program tests [9, 34].

The following protocol of a test session with OBJ3 illustrates a number of test cases performed to evaluate the design specification for function block nonlin. Starting point of this examination was the function depicted together with a particular approximation in Figure 2. The test environment is defined by:

\[
\text{obj TEST-NONLIN is extending V .}
\]
\[
\text{op t : } \to \text{ Time .}
\]
\[
\text{eq n} = 25 .
\]
\[
\text{eq ui(1,t)} = - 2 .
\]
\[
\text{eq } v_i(1, t) = 1 .
\]
\[
\text{eq } u_i(2, t) = 0 .
\]
\[
\text{eq } v_i(2, t) = -1 .
\]
\[
\text{eq } u_i(3, t) = 3 .
\]
\[
\text{eq } v_i(3, t) = -1 .
\]
\[
\text{eq } u_i(4, t) = 7 .
\]
\[
\text{eq } v_i(4, t) = 1 .
\]

... 

end

The different objects in which \(\sim\) and \(v\) are reduced in the following test cases just provide different input values for \(\sim\). Our convention is that object \(Uu\) sets \(u(t) = u\) such as

\[\text{obj } U-1 \text{ is extending TEST-NONLIN .}
\]
\[
\text{eq } u(t) = -1 .
\]

end

We present eight test runs.

*** out of range
reduce in U-3 : u(t)
result NzInt: \(- (s (s (s (0))))\)
reduce in U-3 : v(t)
result NzNat: s (0)
reduce in U-1 : u(t)
result NzInt: \(- (s (0))\)
reduce in U-1 : v(t)
rewrites: 205
result Zero: (0)
reduce in U0 : u(t)
result Zero: (0)
reduce in U0 : v(t)
rewrites: 193
result NzInt: \(- (s (0))\)
reduce in U1 : u(t)
result NzNat: s (0)
reduce in U1 : v(t)
rewrites: 164
result NzInt: \(- (s (0))\)
reduce in U3 : u(t)
result NzNat: s (s (s (0)))
reduce in U3 : v(t)
rewrites: 304
result NzInt: \(- (s (0))\)
reduce in U4 : u(t)
result NzNat: s (s (s (s (0))))
reduce in U4 : v(t)
rewrites: 358
result NzNum: \(- (s (0)) / s (s (0))\)
reduce in U5 : u(t)
result NzNat: s (s (s (s (s (0)))))
reduce in U5 : v(t)
rewrites: 369
result Zero: (0)
reduce in U6 : u(t)
result NzNat: s (s (s (s (s (s (0)))))
reduce in U6 : v(t)
rewrites: 379
result NzNum: s (0) / s (s (0))

We observe that the test results agree with the illustration in Figure 2.

7 Program Verification

The aim of program verification is to increase one's confidence in the correct functioning of a piece of software or software controlled hardware. Two fundamental program verification techniques are Hoare style proof rules [32, 4] and Dijkstra's predicate transformer approach [15]. Both techniques, which have many similarities (see [3]), rely on logic assertions about values of program variables upon entry and exit of an individual statement or a whole program and are concerned with verifying the (partial) correctness of a program S with respect to a given specification (P, Q). P is called the pre-condition; it describes relevant properties of program variables on entry of S, while Q denotes the post-condition of S and describes the expected effect of computation on the program state assuming that the pre-condition P ensures termination of S. The notation

\[ \{P\} S \{Q\} \]

is used to claim that the program S is partially correct, i.e., if P holds before the execution of S and S terminates, then the final state of S will satisfy Q. For example, if S denotes a program implementing the behaviour of function block nonlin, Q might include the assertion that the value of output V satisfies its requirements. Two extreme assertions are true and false. The former is always satisfied. It holds at any place in a program, because it does not constrain the range of variables. In contrast, the assertion false never holds. It restricts the range of variables in such a way that no valid values exist at all.

Both calculi are the basis of many automatic verification systems. Automatic program verification employs so-called verification condition generators to derive assertions about how programs behave from program specifications or programs themselves. The validity of these assertions then has to be proved, manually or with the aid of a theorem prover. Verification condition generators rely on logical deduction systems. Such systems consist of a set of axioms and deduction rules that map a given set of well formed formulae into another set of well formed formulae. Deduction systems for equational logic, for example, which we referred to in the previous two sections, usually include rules for symmetry, transitivity, or substitution of equals for equals [16].

In this section we shall review and illustrate program verification for a simple subset of the ST language. After introducing the language subset and an ST program A that implements function block nonlin in the following section, the weakest pre-condition method will be employed to verify that A respects Requirements 1 to 4. Hoare’s approach is discussed in Section 8, where we consider some development steps of another function block.

7.1 A Subset of Structured Text: Simple ST

To simplify our discussion, we consider the following subset of ST expressed in Backus-Naur form:\(^5\):

\(^5\)Square brackets [ ] indicate optional occurrences of syntactic items.
FUNCTION_BLOCK NONLIN

VAR_INPUT
  U : NUM;  (* input value *)
  N : INT;  (* number of base points *)

VAR
  I : INT  (* local variable used to compute V *)

VAR_OUTPUT
  V : NUM;  (* result value *)

(* *** FB body ***)
IF U < UI[1]
  THEN V := VI[1]
ELSE
  IF UI[N] < U
  THEN V := VI[I]
  ELSE V := VI[I] + (U - UI[I]) * (VI[I+1] - VI[I]) / (UI[I+1] - UI[I])
END_VAR;

Variables, functions, and types in ST are just names; expressions include the usual Boolean and arithmetic operators. This sublanguage turned out to be sufficient for implementing the library of basic function block modules [50] used for process control problems in chemical industry.

In the following iterative solution to function approximation we assume that

- U- and V-co-ordinates of base points are kept in two arrays UI[1.. N] and VI[1.. N] of Num,
- the elements of UI are stored in ascending order, i.e., UI[I] < UI[J] for I < J, and
- UI[I] and VI[I] correspond to each other, i.e., VI[I] = f'(UI[I]).

The ST program describes our algorithm, call it A, in a way adapted from the guideline [50].
THEN $V := V_I[N]$
ELSE
    $I := 1$;
    WHILE $UI[I+1] < U$ AND $U$
        DO $I := I + 1$
    END;
    $V := fp(I, U)$;
FI
END_FUNCTION_BLOCK

where $fp(i, U)$ is used as an abbreviation for the following arithmetic expressions:

$$((VI[I + 1] - VI[I])/(UI[I + 1] - UI[I])) * (U - UI[I]) + VI[I].$$

Each ST program uses a finite set of variables representing input or output data and local variables. In assertions we refer to the values of program variables to reason about the program state. It is, however, important to make a clear distinction between program variables, the values they represent, and the logical variables used in pre- and post-conditions. The latter refer to values of program variables in a particular syntactic context. Program variables may be bound to different values in different syntactic contexts, while logical variables always represent one and the same value. At each point of program execution the program state at this point can be characterized by a function $\sigma$ associating each variable $x$ to a particular value $\sigma(x)$ of the type of that variable.

Statements generally change the association of values with states and, thus, change the program state. They map states into new states. In assertions we, therefore, consider program states immediately before and after statement executions, but not in between. Since the value $\nu(e)$ of an expression $e$ depends on the current values of the variables it contains, we can also determine $\nu(e)$ only with respect to some state $\sigma$. Extending this idea to statements and entire programs, the meaning of a program $S$ can then be interpreted as a function from some initial state to a final state. This function is determined by the composition of state changes effected by the constituent statements of $S$. For the sake of readability, we use, however, a somewhat sloppy notation in the sequel and simply write $x$ in assertions where we actually mean $\sigma(x)$ and $e$ instead of $\nu(e)(\sigma)$.

7.2 Weakest Pre-conditions

Dijkstra's predicate transformer approach introduced in 1976 [15] allows us to derive assertions for individual statements from global assertions relating pre- and post-conditions of an entire program such as nonlin and construct a correctness proof for it from correctness proofs of its constituent statements. The core method provides a calculus to compute the weakest pre-condition of $S$ with respect to $Q$, written $wp(S, Q)$, defining states from which program $S$ could execute and terminate in states meeting $Q$. The calculus can be presented as a collection of axioms, which can be employed to derive pre-conditions from the required post-conditions [3].

Although it is not known how to give a complete set of axioms for expressive programming languages, this assertional technique is applicable for the language subset presented above, and this subset turned out to be sufficient for implementing the behaviour of basic function blocks used in a wide range of process control problems [28]. Below we introduce that subset of Dijkstra's rules we need in the proof of the correctness of algorithm $A$ defined in program
nonlin. A detailed account of the calculus of weakest pre-conditions and many examples are contained in [15] and [3].

Excluded miracle:

\[ wp(S, false) = false \]

There is no initial state that guarantees the execution of \( S \) to result in the impossible state.

Statement composition:

\[ wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q)) \]

The weakest pre-condition for a sequence of two statements \( S_1 \) and \( S_2 \) is found by calculating the weakest pre-condition that must hold for \( P_2 \) to terminate and satisfy \( Q \) and then computing the weakest pre-condition that will force \( P_1 \) to terminate in a state that meets the weakest pre-condition of \( P_2 \) with respect to \( Q \). Note that \( S_1 \) and \( S_2 \) can be statement sequences \( (S_{i_1}; \ldots; S_{i_n}) \) themselves; but as statement composition is associative, we can omit parentheses.

Assignment statements:

\[ wp(x := e, Q) = Q{x/e} \]

If \( Q \) is expressed in terms of variable \( x \) and we want \( Q \) to hold after assigning \( e \) to \( x \), this will only be true if we substitute all free occurrences of \( x \) in \( Q \) by \( e \), which is denoted by \( Q{x/e} \).

Conditional statements:

\[ wp(IF \; B \; THEN \; S_1 \; ELSE \; S_2 \; FI, Q) = \]

\((B \; and \; wp(S_1, Q)) \; or \; (not \; B \; and \; wp(S_2, Q))\)

The weakest pre-condition has to consider the two possible paths through the conditional statement.

If the else clause is omitted, the axiom simplifies to:

\[ wp(IF \; B \; THEN \; S \; FI, Q) = (B \; and \; wp(S, Q)) \; or \; (not \; B \; and \; Q). \]

Iteration:

\[ wp(WHILE \; B \; DO \; S \; END, Q) = \exists k : \text{Nat} \bullet P_k \]

To understand the weakest pre-condition rule for while-loops, we must recall that their intended meaning is a sequence \( S; S; \ldots; S \) of zero or more executions of the loop body \( S \).

If the required post-condition is \( Q \) and the number of iterations is \( k = 0 \), it is obvious that the weakest pre-condition \( P_0 \) must be

\[ P_0 = \text{not} \; B \; and \; Q. \]

Condition \( B \) must initially be false as otherwise the body must have been executed at least once.

Now, if we assume that the body is iterated exactly once, i.e., \( k = 1 \), we see that the execution of \( S \) must start from a state in which \( B \) holds; furthermore, the post-condition of this state must ensure that \( Q \) and the condition that \( S \) is never executed again, which is \( P_0 \), hold. Hence, the pre-condition of a single iteration through a while-loop is:

\[ P_1 = B \; and \; wp(S, P_0). \]

Iterating twice over the loop body must begin in a state in which \( B \) holds, and the second iteration is enabled after executing \( S \) once. The latter, however, is equivalent to \( P_1 \) so that we get

\[ P_2 = B \; and \; wp(S, P_1). \]
Repeating this line of argumentation leads us to a sequence of pre-conditions of the form

\[ P_k = B \text{ and } wp(S, P_{k-1}) \quad (\text{for } k \geq 1) \]

which can be generalised to:

\[ wp(\text{WHILE } B \text{ DO } S \text{ END}, Q) = \exists k : \text{Nat} \cdot P_k. \]

### 7.3 Correctness of Program nonlin

Our objective is now to prove that the approximation algorithm encoded in the body of function block nonlin is correct with respect to the requirements defined in Section 3.4. For convenience of presentation we use a mathematical notation of our algorithm, which is shown in Figure 4, and simply call it \( A \). As before we set

\[ fp(i, u) = \frac{v_{i+1} - v_i}{u_{i+1} - u_i} = \frac{v - v_i}{u - u_i}. \]

IF \( u < u_1 \)

THEN \( v := v_1 \)

ELSE

IF \( u_n < u \)

THEN \( v := v_n \)

ELSE

\[ i := 1; \]

WHILE \( u_{i+1} < u \)

DO \( i := i + 1 \)

END;

\( v := fp(i, u) \)

FI

FI

Figure 4: Approximation algorithm \( A \)

The correctness proof consists in the calculation of \( wp(A, Q) \), where \( Q \) is the post-condition:

\[
Q = (u < u_1 \text{ and } v = v_1) \text{ or } \quad (u > u_n \text{ and } v = v_n) \text{ or } \quad (u_1 \leq u \text{ and } u \leq u_n \text{ and } v = fp(i, u))
\]

A closer inspection of the rules in Section 7.2 reveals that the calculation of weakest pre-conditions of statements often depends on weakest pre-conditions of inner statements, as in the case of conditional and while statements, or it relies on weakest pre-conditions of subsequent statements as in statement compositions. Figure 5 illustrates the structure of the calculation process shown below by decorating the statements of \( A \) with their pre-conditions. This figure also introduces some abbreviations for constituent statements of \( A \), to which we refer in the recursive correctness proof:

\[
wp(A, Q) = wp(\text{IF } u < u_1 \text{ THEN } v := v_1 \text{ ELSE } S_1 \text{ END}, Q)
\]

\[ = (u < u_1 \text{ and } Q[v/v_1]) \text{ or } \quad (u_1 \leq u \text{ and } wp(S_1, Q)) \]
\{wp(A, Q)\}
\[
\text{IF } u < u_1 \\
\text{THEN } v := v_1 \\
\text{ELSE}
\begin{align*}
\{wp(S_1, Q)\} \\
\text{IF } u_n < u \\
\text{THEN } v := v_n \\
\text{ELSE}
\{wp(S_2, Q)\} \\
\{wp(S_3, wp(S_4, wp(S_5, Q)))\}
\end{align*}
\]
\[
\begin{align*}
S_1 & \{ i := 1; \\
& \{wp(S_4, wp(S_5, Q))\} \\
S_2 & \{ \text{WHILE } u_{i+1} < u \text{ DO } i := i + 1 \} \\
& \{wp(S_5, Q)\} \\
S_5 & \{ v := fp(i, u) \}
\end{align*}
\]
\end{align*}
\]
\]}{Q}

Figure 5: Algorithm A decorated with weakest preconditions

\[= (u < u_1 \text{ and } v_1 = v_1) \text{ or } (u_1 \leq u \text{ and } wp(S_1, Q))\]
\[= (u < u_1 \text{ and } \text{true}) \text{ or } (u_1 \leq u \text{ and } wp(S_1, Q))\]
\[= u < u_1 \text{ or } (u_1 \leq u \text{ and } wp(S_1, Q))\]

The simplifications in the first clause of the disjunction are due to the laws of propositional calculus and inconsistencies between conditions in \(Q[v/v_1]\) and the condition \(u < u_1\). Using again the rule of conditional statements and performing similar simplifications, we obtain:

\[
wp(A, Q) = u < u_1 \text{ or } (u_1 \leq u \text{ and wp(IF } u_n < u \text{ THEN } v := v_n \text{ ELSE } S_2 \text{ END, Q}))
\]
\[= u < u_1 \text{ or } (u_1 \leq u \text{ and ((}u_n < u \text{ and } Q[v/v_n]) \text{ or } (u \leq u_n \text{ and } wp(S_2, Q))))\]
\[= u < u_1 \text{ or } (u_1 \leq u \text{ and } u_n < u) \text{ or } (u_1 \leq u \text{ and } u \leq u_n \text{ and } wp(S_2, Q))\]

We observe that:

\[
wp(S_2, Q) = wp(S_3; S_4; S_5, Q) = wp(S_3, wp(S_4, wp(S_5, Q)))
\]
Hence, to complete the calculation of Condition 5 (which is equivalent to Condition 6), we have to determine the weakest pre-conditions of $S_3$, $S_4$, and $S_5$. We apply the assignment rule to calculate the post-condition of the while statement:

$$wp(S_5, Q) = wp(v := fp(i, u), Q)$$
$$= Q[v/fp(i, u)]$$
$$= (u < u_1 \text{ and } fp(i, u) = v_1) \text{ or }$$
$$\quad (u > u_n \text{ and } fp(i, u) = v_n) \text{ or }$$
$$\quad (u_1 \leq u \text{ and } u \leq u_n \text{ and } fp(i, u) = fp(i, u))$$
$$= (u < u_1 \text{ and } fp(i, u) = v_1) \text{ or }$$
$$\quad (u > u_n \text{ and } fp(i, u) = v_n) \text{ or }$$
$$\quad (u_1 \leq u \text{ and } u \leq u_n)$$

(7)

We refer to Condition 7 by $W$. Using $W$ and the rule of statement composition, we can compute the weakest pre-condition of the while statement:

$$wp(S_4, wp(S_5, Q)) = wp(\text{WHILE } u_{i+1} < u \text{ DO } i := i + 1 \text{ END }, W)$$

We recall the procedure to calculate the pre-condition of while statements inductively:

$$P_0 = \text{not } (u_{i+1} < u) \text{ and } W$$
$$= u \leq u_{i+1} \text{ and } W$$
$$= u \leq u_{i+1} \text{ and }$$
$$\quad ((u < u_1 \text{ and } fp(i, u) = v_1) \text{ or }$$
$$\quad \quad (u > u_n \text{ and } fp(i, u) = v_n) \text{ or }$$
$$\quad \quad (u_1 \leq u \text{ and } u \leq u_n)$$
$$= (u < u_{i+1} \text{ and } u < u_1 \text{ and } fp(i, u) = v_1) \text{ or }$$
$$\quad (u \leq u_{i+1} \text{ and } u > u_n \text{ and } fp(i, u) = v_n) \text{ or }$$
$$\quad (u \leq u_{i+1} \text{ and } u \leq u_n)$$

$$P_1 = u_{i+1} < u \text{ and } wp(i := i + 1, P_0)$$
$$= u_{i+1} < u \text{ and } P_0[i/i + 1]$$
$$= u_{i+1} < u \text{ and }$$
$$\quad ((u \leq u_{i+2} \text{ and } u < u_1 \text{ and } fp(i + 1, u) = v_1) \text{ or }$$
$$\quad \quad (u \leq u_{i+2} \text{ and } u > u_n \text{ and } fp(i + 1, u) = v_n) \text{ or }$$
$$\quad \quad (u \leq u_{i+2} \text{ and } u_1 \leq u \text{ and } u \leq u_n))$$
$$= (u_{i+1} < u \text{ and } u \leq u_{i+2} \text{ and } u < u_1 \text{ and } fp(i + 1, u) = v_1) \text{ or }$$
$$\quad (u_{i+1} < u \text{ and } u \leq u_{i+2} \text{ and } u > u_n \text{ and } fp(i + 1, u) = v_n) \text{ or }$$
$$\quad (u_{i+1} < u \text{ and } u \leq u_{i+2} \text{ and } u_1 \leq u \text{ and } u \leq u_n)$$

$$P_2 = u_{i+1} < u \text{ and } wp(i := i + 1, P_1)$$
$$= (u_{i+2} < u \text{ and } u \leq u_{i+3} \text{ and } u < u_1 \text{ and } fp(i + 2, u) = v_1) \text{ or }$$
$$\quad (u_{i+2} < u \text{ and } u \leq u_{i+3} \text{ and } u > u_n \text{ and } fp(i + 2, u) = v_n) \text{ or }$$
$$\quad (u_{i+2} < u \text{ and } u \leq u_{i+3} \text{ and } u_1 \leq u \text{ and } u \leq u_n)$$

23
Note that $u_{i+1} < u$ is implied by condition $u_{i+2} < u$, Requirement 3, which requires $u_{i+1} < u_{i+2}$, and the transitivity of the “$<$” operator.

$$P_k = \begin{cases} 
    u_{i+k} < u \text{ and } u \leq u_{i+k+1} \text{ and } u < u_1 \text{ and } f_p(i + k, u) = v_1) \text{ or} \\
    u_{i+k} < u \text{ and } u \leq u_{i+k+1} \text{ and } u > u_n \text{ and } f_p(i + k, u) = v_n) \text{ or} \\
    u_{i+k} < u \text{ and } u \leq u_{i+k+1} \text{ and } u_1 \leq u \text{ and } u \leq u_n 
\end{cases}$$

From this we conclude that the weakest pre-condition of $S_5$ is:

$$wp(S_4, W) = \exists k \cdot k \geq 0 \text{ and } u_{i+k} < u \text{ and } u \leq u_{i+k+1} \text{ and}$$

$$\begin{cases} 
    (u < u_1 \text{ and } f_p(i, u) = v_1) \text{ or} \\
    (u > u_n \text{ and } f_p(i, u) = v_n) \text{ or} \\
    (u_1 \leq u \text{ and } u \leq u_n) 
\end{cases}$$

which simplifies to:

$$wp(S_4, W) = \begin{cases} 
    1 \leq i \text{ and } i < n \text{ and } u_1 \leq u \text{ and } u \leq u_n 
\end{cases}$$

(8)

Now we return to Condition 6:

$$wp(S_2, W) = wp(S_2, wp(S_4, W))$$

$$= wp(i := 1, 1 \leq i \text{ and } i < n \text{ and } u_1 \leq u \text{ and } u \leq u_n)$$

$$= 1 \leq 1 \text{ and } 1 < n \text{ and } u_1 \leq u \text{ and } u \leq u_n$$

$$= \text{true and } 1 < n \text{ and } u_1 \leq u \text{ and } u \leq u_n$$

$$= 1 < n \text{ and } u_1 \leq u \text{ and } u \leq u_n$$

This yields the post-condition we need to complete the computation of the weakest pre-condition of algorithm $A$ from the intermediate result achieved in Condition 5.

$$wp(A, Q) = \begin{cases} 
    u < u_1 \text{ or} \\
    (u_1 \leq u \text{ and } u_n < u) \text{ or} \\
    (u_1 \leq u \text{ and } u \leq u_n \text{ and } wp(S_2, Q)) 
\end{cases}$$

$$= u < u_1 \text{ or}$$

$$\begin{cases} 
    (u_1 \leq u \text{ and } u_n < u) \text{ or} \\
    (u_1 \leq u \text{ and } u \leq u_n \text{ and } wp(S_2, Q)) 
\end{cases}$$

$$= \begin{cases} 
    u < u_1 \text{ or} \\
    (u_1 \leq u \text{ and } u_n < u) \text{ or} \\
    (u_1 \leq u \text{ and } u \leq u_n \text{ and } 1 < n) 
\end{cases}$$

(9)

Remembering that a general requirement for the design of function block nonlin was that the number of base points is larger than one, because otherwise no useful approximation algorithm can be defined, the weakest pre-condition of $A$ guarantees that the algorithm will terminate and satisfy $Q$, independent of the initial state the computation started from.

8 A Second Example

This section introduces a second function block example to demonstrate two further verification techniques: specification verification by exhaustive specification testing and program verification by means of Hoare rules.
8.1 Problem Statement

The following example is again taken from the guideline [50]. The function block check serves to perform process data supervision. Its functionality and intended behaviour may be stated informally as follows:

There are eight inputs referring to
DIGX: a digitised and normalised signal produced from
analogue raw data by an external A/D converter,
XB: begin of measuring range,
XE: end of measuring range,
CF: channel fault indicator,
UL: upper alarm limit,
LL: lower alarm limit,
UAE: upper alarm limit enabled,
LAE: lower alarm limit enabled
and four outputs producing
X: a numerical measuring value derived from DIGX,
UA: an alarm signal on range overflow,
LA: an alarm signal on range underflow,
COND: a notification about the processing condition

Inputs on X and CF are mandatory, while the other inputs are constant parameters that are preset for each application or assume the following default values: CF = false, UAE = LAE = true, XB = LL = minum, and XE = UL = maxum.

In each computation of X from DIGX the numerical range limits XB and XE are used to normalise and delimit the measuring value X. If a channel fault is indicated on input CF, the X value of the previous computation cycle is delivered. The inputs UL and LL define an upper and lower limit at which an alarm is raised via output UA or LA, respectively, provided that input UAE or LAE, respectively, is enabled. Output COND indicates one out of three possible states according to the actual conditions and the following increasing priorities:

<table>
<thead>
<tr>
<th>Value</th>
<th>indicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>channel fault occurred,</td>
</tr>
<tr>
<td>2</td>
<td>measuring value above upper limit,</td>
</tr>
<tr>
<td>3</td>
<td>measuring value below lower limit</td>
</tr>
</tbody>
</table>

8.2 Functional Requirements of check

We choose bit sequences to represent digitised data.

```
obj BIT is sort Bit .
op 0 : -> Bit .
op 1 : -> Bit .
endo
```

Operation bits2int converts bit sequences into positive integers.

---

6The original problem description is a little bit more complicated. The omissions made here are, however, irrelevant for our discussion.
obj BITS is sort Bits .
    protecting BIT .
    protecting INT .
    subsort Bit < Bits .
    op __ : Bit Bits -> Bits .
    op bits2int : Bits -> Nat .
    op |!| : Bits -> Nat .
    var N : Nat .
    eq 2` 0 = 1 .
    cq 2` N = 2` (N - 1) * 2 if N > 0 .
    var B : Bit .
    var BS : Bits .
    eq | B | = 1 .
    eq | B BS | = 1 + | BS | .
    eq bits2int(0) = 0 .
    eq bits2int(1) = 1 .
    eq bits2int(B BS) = bi(B) * 2` | BS | + bi(BS) .
end

The default values for numerical inputs are defined by an extension of the built-in object INT:

obj NUM is protecting INT .
    op minnum : -> Int .
    op maxnum : -> Int .
end

The functionality of inputs is obvious:

obj INPUT is protecting INT .
    protecting BITS .
    protecting TIME .
    op digx : Time -> Bits .
    op xb : Time -> Int .
    ...
    op lae : Time -> Bool .
end

As the outputs of computations may refer to results of previous computation cycles, we assume that each output is initialised with a suitable value to ensure that all computations of check are well defined. When trying to initialise output cond, we realise that nothing was said about the state in which none of the stated faults had occurred. Obviously the ok indication was overseen and, therefore, it is encoded as value 0 to the output specification.

obj OUTPUT is protecting NUM .
    protecting BITS .
    protecting TIME .
    op x : Time -> Int .
    op ua : Time -> Bool .
    op la : Time -> Bool .
    op cond : Time -> Nat .
\begin{verbatim}
esq x(0) = minnum .
esq \text{ua}(0) = false .
esq \text{la}(0) = false .
esq \text{cond}(0) = 0 .
end
\end{verbatim}

To keep the presentation short, we focus on a few requirements which are amenable to exhaustive case analysis. We conjecture

\begin{align*}
\text{cond}(\text{tick}(T)) &= 0 \text{ if not } \text{cf}(T) \text{ and not } \text{ua}(T) \text{ and not } \text{la}(T) \\
\text{cond}(\text{tick}(T)) &= 1 \text{ if } \text{cf}(T) \\
\text{cond}(\text{tick}(T)) &= 2 \text{ if not } \text{cf}(T) \text{ and } \text{ua}(T) \tag{10} \\
\text{cond}(\text{tick}(T)) &= 3 \text{ if not } \text{cf}(T) \text{ and not } \text{ua}(T) \text{ and } \text{la}(T)
\end{align*}

to hold for all times greater than 0 as \text{cond} is set to 0 by default. We notice that the definition of this requirement is incomplete, because it depends on the definitions of the output functions \text{ua} and \text{la}. This will be the subject of the design step described in the following section.

### 8.3 Design of check

Relying on the variable definition:

\begin{verbatim}
var T : Time.
\end{verbatim}

the outputs \text{UA} and \text{LA} are defined by

\begin{align*}
\text{eq } \text{ua}(\text{tick}(T)) &= (\text{ul}(\text{tick}(T)) < x(\text{tick}(T))) \text{ and } \text{uae}(\text{tick}(T)). \\
\text{eq } \text{la}(\text{tick}(T)) &= (x(\text{tick}(T)) < \text{ll}(\text{tick}(T))) \text{ and } \text{lae}(\text{tick}(T)).
\end{align*}

Informally, the first equation states that \text{ua} is true if and only if both the actual value of \text{x} exceeds the upper range limit \text{ul} and the input \text{uae} is true at the same time.

The output \text{x} is defined by:

\begin{verbatim}
cq(x)(\text{tick}(T)) = \text{xb}(\text{tick}(T)) + \text{bits2int}((\text{digt}(\text{tick}(T)))) 
* (\text{x}(\text{tick}(T)) - \text{xb}(\text{tick}(T))) 
\text{if not } \text{cf}(\text{tick}(T)). \\
cq(x)(\text{tick}(T)) = \text{x}(T) \text{ if } \text{cf}(\text{tick}(T)).
\end{verbatim}

A graphical illustration of the black-box abstraction we developed in this section is depicted in Figure 6.

### 8.4 Correctness of Functions \text{cond}, \text{ua}, and \text{la}

Requirement 10 is proved by case analysis which examines all possible states of \text{check} by specification testing. We conjecture that 4·2³ cases cover a complete proof. The 2³ base cases depend on the three \text{Bool}-valued inputs \text{cf}, \text{uae}, and \text{lae}. The expected results are described by the following table.

The variation in the fifth, sixth, and seventh row results from the definitions of functions \text{ua} and \text{la}, which depend on the values of inputs \text{UA} and \text{LA} and the value of function \text{x}(T) as expressed by the following propositions:

\begin{align*}
\text{not } \text{uae}(T) \implies \text{not } \text{ua}(T) \\
\text{not } \text{lae}(T) \implies \text{not } \text{la}(T) \\
\text{uae}(T) \text{ and } (\text{ul}(T) < \text{x}(T)) \implies \text{ua}(T) \\
\text{lae}(T) \text{ and } (\text{x}(T) < \text{ll}(T)) \implies \text{la}(T)
\end{align*}
Figure 6: Abstract functionality of function block check

Table 2: States of function block check

<table>
<thead>
<tr>
<th>cond(i)</th>
<th>cf(T)</th>
<th>uae(T)</th>
<th>lae(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>0, 2, or 3</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>0 or 2</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>0 or 3</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
The first two propositions can be verified directly by reduction. Constant \( t \) denotes an arbitrary time greater than 0.

---

OBJ> reduce (not uae(t)) implies (not ua(t)) .
result Bool: true
OBJ> reduce (not lae(t)) implies (not la(t)) .
result Bool: true
---

The third proposition is verified under the assertion that \( x(t) \) exceeds the upper range limit \( ul(t) \), which we include as axiom

\[
ul < m \times q + Xb
\]

in the OBJ specification and, then, verify the proposition under the assumption \( uae(t) = \text{true} \):

---

OBJ> reduce (uae(t) and (ul(t) < x(t))) implies ua(t)
result Bool: true
---

Similarly the last proposition can be proved, but is omitted here.

We complete the proof of operation \texttt{cond} by

1. defining eight different input vectors reflecting the combinations of values on the inputs \( CF, UAE \) and \( LAE \); note that time \( t' \) serves to refer to the value of \( x \) from the previous computation cycle, which occurs in cases of a channel fault:

\begin{verbatim}
obj CHECK-BEHAV-ASS-INPUT is
    protecting CHECK-BEHAV-ASS .
cps t1 t2 t3 t4 t5 t6 t7 t8 t' : -> Time .
cps Ul L1 : -> Int .
eq t1 = tick(t') .
eq t2 = tick(t') .
...
eq t8 = tick(t') .
*** case 1
    eq cf(t1) = true .
    eq uae(t1) = true .
    eq ul(t1) = Ul .
    eq lae(t1) = true .
    eq 11(t1) = L1 .
*** case 2
    eq cf(t2) = true .
    eq uae(t2) = true .
    eq ul(t2) = Ul .
    eq lae(t2) = false .
    eq 11(t2) = L1 .
...
*** case 8
\end{verbatim}
eq cf(t1) = false.
eq uae(t1) = false.
eq ul(t1) = Ul.
eq lae(t1) = false.
eq ll(t1) = Ll.
endo

2. assuming that the previous value of x lies within the range limits:

obj CHECK-BEHAV-ASS-QLDX is
  protecting CHECK-BEHAV-ASS-INPUT.
eq Ul < x(t') = false.
eq x(t') < Ll = false.
endo

and

3. using OBJ3 with four different combinations of assumptions about the value of x(T) with respect to the range limits Ul and Ll for T = t1, ... , t8:

(a) Lower limit violated:

obj CHECK-BEHAV-ASS-LIM1 is
  using CHECK-BEHAV-ASS-QLDX.
eq Ul < m * q + Xb = false.
eq m * q + Xb < Ll = true.
endo

Reduction yields:

=================================
cond(t1).
result NzNat: 1
cond(t2).
result NzNat: 1
cond(t3).
result NzNat: 1
cond(t4).
result NzNat: 1
*** lower limit violated and lae
cond(t5).
result NzNat: 3
*** lower limit violated and not lae
cond(t6).
result Zero: 0
*** lower limit violated and lae
cond(t7).
result NzNat: 3
*** lower limit violated and not lae
cond(t8).
result Zero: 0
=================================

(b) Upper limit exceeded:
Proof by case analysis as before under the assumption:
obj CHECK-BEHAV-ASS-LIM2 is
    protecting CHECK-BEHAV-ASS-OLDX .
    eq U1 < m * q + Xb = true .
    eq m * q + Xb < L1 = false .
end

(c) x within range limits:

obj CHECK-BEHAV-ASS-LIM3 is
    protecting CHECK-BEHAV-ASS-OLDX .
    eq U1 < m * q + Xb = false .
    eq m * q + Xb < L1 = false .
end

Reduction in the proper context yields:

===============================================
cond(t1) .
result NzNat: 1
...
cond(t4) .
result NzNat: 1
***> not cf
cond(t5) .
result Zero: 0
...
cond(t8) .
result Zero: 0
===============================================

(d) x beyond upper and below lower range limit (which is not really possible because
xb(T) <= xe(T) for all inputs):

obj CHECK-BEHAV-ASS-LIM4 is
    protecting CHECK-BEHAV-ASS-OLDX .
    eq U1 < m * q + Xb = true .
    eq m * q + Xb < L1 = true .
end

===============================================

Reduction yields:

===============================================
***> cf
cond(t1) .
result NzNat: 1
...
cond(t4) .
result NzNat: 1
***> not cf and ua
cond(t5) .
result Zero: 2
***> not cf and ua
cond(t6) .
result Zero: 2
***> not cf and not ua and la
cond(t7) .

31
result Zero: 3
cond(t8) .

>>> not cf and not ua and not la
result Zero: 0

=========================================

8.5 ST Program Implementing Function Block check

The following ST program, we call it check, is intended to implement the behaviour of function block check:

FUNCTION_BLOCK CHECK

VAR_INPUT
    DIGX : BITS; (* digital value *)
    XB : INT; (* begin of measuring range *)
    ...
    LL : INT; (* lower alarm level *)
END_VAR;

VAR_OUTPUT
    X : INT; (* resulting physical value *)
    ...
END_VAR;

VAR
    OLDC : INT; (* previous condition *)
END_VAR;

IF NOT CF THEN X := XB + BITS_TO_INT(DIGX) * (XE - XB)
FI;

IF X > UL
    THEN UA := TRUE
    ELSE IF X < LL
        THEN UA := FALSE
    FI
FI;

IF X < LL
    THEN LA := TRUE
    ELSE IF X > UL
        THEN LA := FALSE
    FI
FI;

IF NOT (CF OR UA OR LA)
    THEN COND := 0;
    OLDC := 0
ELSE
    IF CF
        THEN COND := 1
    ELSE
        IF UA
            THEN COND := 2
        ELSE
            IF LA

32
THEN COND := 3
FI
FI
FI;
IF COND <> 0LDC
   THEN 0LDC := COND
FI
FI
END_FUNCTION_BLOCK

8.6 Hoare Style Proof Rules

In this section we define a collection of proof rules for the different types of constructs of simple ST. The meaning of declarations is omitted, because we assume that the programs we consider are well typed, and the domains of values denoted by program variables are disjoint so that they can be lumped together in a single value domain.

We use the notation

\[ \phi_1, \phi_2, \ldots, \phi_n \rightarrow \phi \]

to state that “\( \phi \) is provable if \( \phi_1 \) and \( \phi_2 \) and \ldots and \( \phi_n \) are provable”. For \( n = 0 \) we have an axiom stating that “\( \phi \) is provable”, and we simply write

\[ \phi. \]

The following axiom and five deduction rules establish a proof system sufficient to construct formal proofs of the total correctness of programs written in simple ST. The validity of these rules is examined in [4] and will be taken for granted here.

The assignment rule

\[ \text{[ASS]} \quad \{Q[x/e]\}x := e\{Q\} \]

allows us to derive the weakest pre-condition \( Q[x/e] \) from a given post-condition \( Q \).

The sequential composition rule

\[ \text{[SEQ]} \quad \{P\}S_1\{Q\}, \{Q\}S_2\{R\} \rightarrow \{P\}S_1;S_2\{R\} \]

means that if the execution of \( S_1 \) under pre-condition \( P \) satisfies post-condition \( Q \) and if the execution of \( S_2 \) under pre-condition \( Q \) satisfies post-condition \( R \), then we can conclude that the execution of the statement sequence \( S_1;S_2 \) under pre-condition \( P \) satisfies \( R \).

The conditional statement rule occurs in two forms:

\[ \text{[COND1]} \quad \{P \text{ and } C\}S_1\{Q\}, \quad \{P \text{ and not } C\}S_2\{Q\} \rightarrow \{P\}\text{IF } C \text{ THEN } S_1 \text{ ELSE } S_2 \text{ FI}\{Q\} \]

and

\[ \text{[COND2]} \quad \{P \text{ and } C\}S\{Q\}, \quad P \text{ and not } C \Rightarrow Q \rightarrow \{P\}\text{IF } C \text{ THEN } S \text{ FI}\{Q\} \]

One more rule, called the rule of consequence, is useful to relate program pieces whose pre- and post-conditions are not identical, but related:

\[ \text{[CONS]} \quad P \Rightarrow P', \quad \{P'\}S\{Q'\}, \quad Q' \Rightarrow Q \rightarrow \{P\}S\{Q\} \]

\(^7\)Termination of simple ST programs is guaranteed, hence we can speak about total correctness.
Correctness proofs are typically constructed by composing deduction rules and axioms to proof trees. Each axiom $\phi$ of a given set of deduction rules is considered a proof tree with root $\phi$. Furthermore, if $\Theta_1, \ldots, \Theta_n$ are proof trees with roots $\phi_1, \ldots, \phi_n$ and there is a rule
\[
\frac{\phi_1 \cdots \phi_n}{\phi}
\]
in the set of rules, then
\[
\frac{\Theta_1 \cdots \Theta_n}{\phi}
\]
is also a proof tree with root $\phi$.

8.7 Correctness of Program check

Using the proof technique of the previous section, we now construct a proof that our sample program \texttt{check} satisfies post-condition $Q$ under the pre-condition
\[
P = \text{true},
\]
where $Q$ is the conjunction
\[
\text{OLDC} = \text{COND and (COND = 0 and not UA and not LA and not CF) or (COND = 1 and CF) or (COND = 2 and not CF and UA) or (COND = 3 and not CF and not UA and LA)}.
\]

This post-condition is actually stronger than Requirement 10 and the design specification for \texttt{cond} defined in Section 3. Hence, if we succeed to verify that \texttt{check} satisfies $Q$, we can deduce by means of rule CONS that \texttt{check} also satisfies Requirement 10 due to the fact that
\[
(a \text{ and } b) \text{ implies } a
\]
holds in propositional calculus, where $a$ stands for Requirement 10 and $b$ for the expression $\text{OLDC} = \text{COND}$, which we added to Requirement 10 in postcondition $Q$.

Again we shall use OBJ to express assertions about ST programs.

The goal of program verification is to verify a program in a finite number of steps by applying appropriate proof rules to the statements of that program. In the stepwise proof we proceed as follows:

1. Start from the post-condition and follow the program path in backward direction.

2. For each statement $S$ on this path an appropriate proof rule is applied to transform the post-condition of this statement into a weakest pre-condition. This pre-condition is then the post-condition of the preceding statement.

That is, we derive pre- and post-conditions for the statements constituting \texttt{check} using appropriate proof rules. For example, for a conditional statement
\[
S = \text{IF C THEN } S_1 \text{ ELSE } S_2 \text{ FI}
\]
with pre-condition $P$ and post-condition $R$ we can derive pre- and post-conditions for $S_1$ and $S_2$ by means of rule COND1:
\[
\{P \text{ and } C\} \quad S_1 \quad \{R\} \quad (11)
\]
\[
\{P \text{ and not } C\} \quad S_2 \quad \{R\} \quad (12)
\]

If we can find a proof for Assertion 11 and 12, we also have a proof for $\{P\}S\{R\}$.  

34
3. The leaves in the proof tree thereby constructed are assignments; for each assignment specification \( P \{ x := e \{ Q \} \) we can derive a pre-condition \( Q \{ x/e \} \) using the assignment axiom; finally, we have to show that the derived pre-condition is implied by the specified pre-condition, i.e., \( P \Rightarrow Q \{ x/e \} \) must hold.

This procedure is also applied by automatic verification condition generators.

In the proof below we restrict ourselves to the statement computing output \( \text{COND} \) and state variable \( \text{OLDC} \). This program part is depicted in Figure 7, where we also introduce some abbreviations for substatements to ease our discussion.

![Figure 7: Structure of program \( S \) computing \( \text{COND} \) and \( \text{OLDC} \)](image)

To prove that \( S \) satisfies its specification, we first construct a proof tree using rules defined in the preceding section. This tree is depicted in Figure 8. The root of this tree is the assertion to be verified, while the leaves of the tree are the verification conditions that must be proved. Assertions \( T \) and \( R \) are defined by:

\[
T = (\text{COND} = 0 \land \text{not UA} \land \text{not LA} \land \text{not CF}) \lor \\
(COND = 1 \land \text{CF}) \lor \\
(COND = 2 \land \text{not CF} \land \text{UA}) \lor \\
(COND = 3 \land \text{not CF} \land \text{not UA} \land \text{LA})
\]

and

\[
R = T \land (\text{COND} = 0).
\]

Hence, we have that

\[
Q = T \land (\text{COND} = \text{OLDC}).
\]

The verification conditions occurring in the proof tree in Figure 8 are the following, reading the tree from left to right in a depth-first strategy:

1. \((\text{not (CF or UA or LA)}) \Rightarrow R \{ \text{COND/0} \})
2. \(R \Rightarrow Q \{ \text{OLDC/0} \})
Figure 8: Proof tree for program S
3. ((CF or UA or LA) and CF) implies T{COND/1}
4. ((CF or UA or LA) and not CF and UA) implies T{COND/2}
5. ((CF or UA or LA) and not CF and not UA and LA) implies T{COND/3}
6. ((CF or UA or LA) and not CF and not UA and not LA) implies T
7. (T and (OLDC =/= COND)) implies Q{OLDC/COND}
8. (T and (OLDC == COND)) implies Q

The proof of these verification conditions reduces to term re-writing in OBJ3 relying on the laws of propositional logic specified in the built-in object PROP and the following declarations:

```plaintext
obj PROG-VERIFICATION is .
  op R : -> Prop .
  op T : -> Prop .
  op Q : -> Prop .
  op T{COND/0} : -> Prop .
  op T{COND/1} : -> Prop .
  op T{COND/2} : -> Prop .
  op T{COND/3} : -> Prop .
  op R{COND/0} : -> Prop .
  op R{COND/1} : -> Prop .
  op Q{OLDC/COND} : -> Prop .

eq T = ((COND == 0) and (not 'CF) and (not 'UA)
  and (not 'LA)) or
  ((COND == 1) and 'CF) or
  ((COND == 2) and (not 'CF) and 'UA) or
  ((COND == 3) and (not 'CF) and (not 'UA) and 'LA).

eq R = ('COND == 0) and T .

eq Q = T and ('COND == 'OLDC) .

eq T{COND/0} = ((0 == 0) and (not 'CF) and (not 'UA)
  and (not 'LA)) or
  ((0 == 1) and 'CF) or
  ((0 == 2) and (not 'CF) and 'UA) or
  ((0 == 3) and (not 'CF) and (not 'UA)
  and 'LA).

eq T{COND/1} = ((1 == 0) and (not 'CF) and (not 'UA)
  and (not 'LA)) or
  ((1 == 1) and 'CF) or
  ((1 == 2) and (not 'CF) and 'UA) or
  ((1 == 3) and (not 'CF) and (not 'UA)
  and 'LA).

eq T{COND/2} = ((2 == 0) and (not 'CF) and (not 'UA)
  and (not 'LA)) or
  ((2 == 1) and 'CF) or
  ((2 == 2) and (not 'CF) and 'UA) or
  ((2 == 3) and (not 'CF) and (not 'UA)
  and 'LA).
```

37
eq \( T\{\text{COND}/3 \} = ((3 == 0) \text{ and (not 'CF) and (not 'UA) and (not 'LA)) or}
\)
\((3 == 1) \text{ and 'CF) or}
\((3 == 2) \text{ and (not 'CF) and 'UA) or}
\((3 == 3) \text{ and (not 'CF) and (not 'UA) and 'LA)} \).  

eq \( R\{\text{COND}/0 \} = (0 == 0) \text{ and } T\{\text{COND}/0 \} \).  

eq \( R\{\text{COND}/1 \} = (1 == 0) \text{ and } T\{\text{COND}/1 \} \).  

eq \( Q\{\text{OLDC}/0 \} = \text{T and ('COND == 0)} \).  

eq \( Q\{\text{COND}/1 \} = T\{\text{COND}/1 \} \text{ and (1 == 'OLDC)} \).  

end

The first verification condition is proved by:

=============

*** VC 1

obl> reduce (not (CF or UA or LA)) implies R{COND/0}.

reduce in PROG-VERIFICATION : not (CF or UA or LA)

implies R{COND/0}

rewrites: 445

result Bool: true

=============

To understand the proof we remember that

\( R\{\text{COND}/0 \} = (0 == 0) \text{ and } T\{\text{COND}/0 \} \)

with

\( T\{\text{COND}/0 \} = ((0 == 0) \text{and not CF and not UA and not LA) or}
\((0 == 1) \text{ and CF) or}
\((0 == 2) \text{ and not CF and UA) or}
\((0 == 3) \text{ and not CF and not UA and LA) \)

Further, the proof of the above verification condition’s truth in PROG-VERIFICATION exploits standard equalities such as

============================

p and true = p
p or false = p
not (p or q) = not p and not q
(p or q) and p = p

============================

and others. Hence,

\( R\{\text{COND}/0 \} \)

re-writes to

not CF and not UA and not LA

because all other components in the disjunction become false due to the fact that the equality expressions 0 == 1, 0 == 2, and 0 == 3 are false and only 0 == 0 is true.

The other verification conditions are verified similarly.
---

### VC 2

Obj> reduce R implies Q\{OLDC/0\}.

result Bool: true

### VC 3

Obj> reduce ((CF or UA or LA) and CF)

implies T\{COND/1\}.

result Bool: true

### VC 4

Obj> reduce ((CF or UA or LA) and (not CF) and UA)

implies T\{COND/2\}.

result Bool: true

### VC 5

Obj> reduce ((CF or UA or LA) and (not CF) and

(not UA) and LA) implies T\{COND/3\}.

rewrites: 363

result Bool: true

### VC 6

Obj> reduce ((CF or UA or LA) and (not CF) and (not UA)

and (not LA)) implies T.

result Bool: true

### VC 7

Obj> reduce (T and (OLDC /= COND)) implies Q\{OLDC/COND\}.

result Bool: true

### VC 8

Obj> reduce (T and (COND == OLD)) implies Q.

result Bool: true

---

With the proof of all eight verification conditions the root of the proof tree in Figure 8 is verified and, hence, program $S$ is proved correct with respect to its formalised requirements.

## 8.8 Improvement of Program check

The previous section illustrated the correctness proof for an ST program, viz., that of a program derived from problem specifications for the function block check. Instead of verifying this program, we could have derived the simpler program shown in Figure 9 from post-condition $Q$ defined at the beginning of Section 8.7.

The post-condition of $S_1$ is now

$$(Q_1 \text{ or } (Q_2 \text{ or } (Q_3 \text{ or } Q_4)))$$

which, due to the associativity of Boolean disjunction, may be written as

$$Q_1 \text{ or } Q_2 \text{ or } Q_3 \text{ or } Q_4.$$
Figure 9: Simpler program derived from post-condition Q

If we follow $S_1$ with $S_4: \text{OLDC} := \text{COND}$, we have the additional conjunction \text{OLDC} = \text{COND}, because \text{OLDC} does not occur in $Q_1$ ... $Q_4$.

This example shows that in special cases correctness proofs can more easily be obtained with the help of predicate calculus. The latter, however, is a solely manual method and, therefore, turns less manageable with increasing program size. This is the reason for presenting a tool based method in the last section, although it appears less efficient in the case of procedure check. Moreover, tool supported methods are considered to be more reliable.

9 Verification of a Timer Program using HOL

In this section we investigate a third way of verifying function block programs written in Structured Text in that we use the HOL system\footnote{Actually we use HOL90, which is a re-implementation of HOL in Standard ML.} \cite{holl90} as a mechanised proof assistant. HOL supports interactive theorem proving based on a typed variant of Church’s higher order predicate logic\footnote{Higher order logic extends first order logic by including variables whose values are functions and allowing higher order functions which use other functions as arguments and values.} \cite{holl90}. That is, proofs are constructed interactively with assistance by the HOL system, but they are guided by the user who defines the structure and content of a proof. HOL’s specification logic is interfaced with the Standard ML (SML) functional programming language in which proof procedures and strategies such as reduction to normal forms or case analysis can be encoded. As an example we use a delay timer introduced as follows:

A non-re-triggerable monostable element with selectable delay, i.e., when the element is in the non-excited state and it detects a rising edge at its input, then it switches its output to the logical true state for a duration specified by the delay, $t$.

An ST program implementing this functionality and now to be verified is given by:
FUNCTION_BLOCK timer(d,delay,q);
   BOOLEAN d,q;
   DURATION delay;
   BEGIN
      BOOLEAN previous INIT(FALSE), triggered INIT(FALSE);
      TIME deadline;
      IF NOT triggered THEN
         IF NOT previous AND d THEN
            triggered:=TRUE; deadline:=clock()+delay
         FI
      FI;
      IF triggered THEN triggered:=clock() LT deadline FI;
      previous:=d;
      q:=triggered
   END;

The technique we apply to the timer program is different from the Hoare logic and predicate transformer approach in two respects: Hoare rules and weakest pre-conditions provide special purpose logics that reflect the semantics of programming language concepts in their structure but differ, in general, from the logics used to model the formal semantics of the programming language at hand (In many cases the actual compiler represents the only formal semantics available.). In this experiment, however, syntax and semantics of simple ST are fully defined in HOL so that formal reasoning about ST programs is based directly on their semantics. This obviates the need for explicit construction of pre- and post-conditions and the verification of their consistency with the design specification. A second difference, as we shall see in the sequel, is that proofs are more direct and transparent as opposed to theorem proving in HOL.

9.1 Abstract Syntax of Simple ST

The abstract syntax of our subset of ST is re-specified below using a slightly different variant of BNF. In the syntax description, the variable V ranges over program variables, E₁, E₂ range over natural number expressions, B ranges over Boolean expressions, and variables S, S₁, S₂ range over statements.

\[
S ::= \text{skip} \\
| V ::= E \\
| S₁; S₂ \\
| \text{IF } B \text{ THEN } S₁ \text{ ELSE } S₂ \\
| \text{read } V
\]

Statement \text{skip} denotes the no-operation statement, which has no effect on the actual program state. It is introduced to simplify the semantic definition. It can be used, for example, to model conditional statements with empty else clauses by the expression “\text{IF } B \text{ THEN } S \text{ ELSE skip } \text{FI}”. Note that we omitted the while loop, which is not used in program \text{timer}, to simplify the semantic definition and correctness proof of this program in HOL.

To avoid confusion with the interpretation of certain ASCII characters in HOL and Simple ST, respectively, ST expressions are enclosed in “‘--’” and “‘--’” within HOL terms, such as --‘v := E‘--. Furthermore, statement sequences are separated by a double semicolon, and program variables are written in upper case letters while logic variables are written in lower case. Names of program variables are represented by strings, and states are modelled as functions mapping variables onto natural numbers. The abbreviation for a HOL type on the level of SML is written as follows:
val state = ty_antiq("=":string->num");
where ty_antiq denotes the predefined type antiquotation operator.

Natural number expressions and Boolean expressions are just modelled by total functions nexp
and bexp mapping states onto numbers and Booleans, respectively:
val nexp = ty_antiq("=":state -> num");
val bexp = ty_antiq("=":state -> bool");
where the unquote operator "=" yields the value of the term to which it is applied.

We can now use the recursive types package of HOL to define the syntax of Simple ST statements
and programs:

val stmt =
define_type{name="stmt",
type_spec = 'stmt = skip
| := of string # nexp
| ;; of (stmt#stmt)
| if of `bexp#stmt#stmt
| read of string',
fixities = [Prefix,Infix 400, Infix 350, Prefix, Prefix]});

The recursive types package derives a standard syntactic theory including a structural induction
theorem:
val induct = save_thm ("induct",prove_induction_thm stmt);
and an exhaustive case analysis theorem:
val cases = save_thm ("cases", prove_cases_thm induct);
for statements. Further generic proofs can be derived to show that the constructors of the
abstract syntax are one-to-one.

9.2 Semantics of Simple ST in HOL

The semantics of Simple ST are modelled by function

EVAL : stmt-- > state-- > state-- > state

which maps statements, a particular environment, and state into a new state. The environment
state env models program input and, thus, serves to give meaning to read statements. EVAL is
defined by primitive recursion over the stmt type such that

EVAL S env s = t

holds exactly when statement S is being executed in the environment env and the initial state
s terminates in the final state t.

Before presenting further HOL definitions, we have to mention that the usual logic symbols
\neg, \vee, \land, \forall, \exists, and \lambda are represented in a somewhat unfamiliar fashion by ", \|, /, !, ?, and \, respectively, because HOL solely uses ASCII characters.

Now, function EVAL can be defined by:

let val EVAL =
(\--'EVAL : stmt -- state -- state -- state'\--)

42
in
new_recursive_definition{
  name= "EVAL", fixity= Prefix, rec_axiom=stmt,
  def = --'
  ("EVAL skip" =
    \env s. s) /
(\EVAL (V := E) =
  \env s. (\forall v. (v = v) => E s | s v)) /
(\EVAL (S1 ;; S2) =
  \env s. (\EVAL S2 env (\EVAL S1 env s)) /
(\EVAL (if B S1 S2) =
  \env s. (B s => \EVAL S1 env s |
     \EVAL S2 env s)) /
(\EVAL (read v) =
  \env s. (\forall v. (v = v) => env v | s v))
'--->)
end;

The semantics of a reactive program (or an automaton built from a deterministic program) are defined by a function mapping some initial state and a list of environments (modelling the input sequence) to a list of result states, which are to contain the states of some variables representing the output:

\texttt{EVALR: stmt} \rightarrow \texttt{state} \rightarrow \texttt{statelist} \rightarrow \texttt{statelist} 

In HOL this is stated by:

\texttt{val automaton = ty antiq(=':('state \rightarrow \texttt{state}
  \rightarrow \texttt{state}'=');}

\texttt{let val EVALR =
  ('EVALR : \texttt{automaton} \rightarrow \texttt{state list} \rightarrow \texttt{state}
  \rightarrow \texttt{state list'}--')
in
new_recursive_definition{
  name= "EVAL", fixity= Prefix, rec_axiom=list_Axiom,
  def = --'
  (\EVALR autom NIL s = [s]) /
(\EVALR autom (CONS env envs) s
  = CONS (autom env (HD (EVALR autom envs s)))
     (EVALR autom envs s))
)'--->)
end;

9.3 Timer Verification

To enable the automated verification of the timer program we need four more components:

1. a representation of the program as a term in HOL,
2. the specification of expected timer properties in HOL,
3. the definition of proof tactics, and

43
4. an abundance of lemmata supporting the proof.

The timer program in HOL is defined by:

```haskell
new_definition("timer_prog", --‘timer_prog =
read "signal";
read "delay";
read "clock";
if ( NOT(bvalue "triggered") AND NOT(bvalue "previous")
AND bvalue "signal")
  (* then *)
  ("deadline" := (value"clock" PLUS value"delay")
  (* else *) skip;
"triggered" := (value"clock" LT value"deadline")
"previous" := value"signal"
‘--);
```

where the functions value and bvalue evaluating arithmetic and Boolean expressions, respectively, in a given environment are defined by:

```haskell
val value = new_definition("value",--
  ‘value v = (\s : \"state. s v\")’--);
val bvalue = new_definition("bvalue",--
  ‘bvalue v = (\s : \~state. s v = 1)’--);
```

Note that Booleans are represented by numbers with 0 denoting value false and 1 denoting value true.

The arithmetic operator PLUS and the Boolean operators AND, LT, and NOT are mapped onto corresponding operators in HOL:

```haskell
new_infix_definition ("PLUS",
  --‘$PLUS a b = \s: \~state. a s + b s’--, 400);
new_infix_definition ("AND",
  --‘$AND a b = \s: \~state. a s /\ b s’--, 500);
new_infix_definition ("LT",
  --‘$LT a b = \s: \~state. (a s < b s => 1 | 0)’--, 450);
new_definition ("NOT",
  --‘NOT a = \s: \~state. ~(a s)’--);
```

The specification below, which the timer has to fulfill, is parameterised by an input environment envs. Intuitively, the term tstate envs i refers to the state resulting after i evaluations of program timer in envs. The timer specification relies on the following definitions:

```haskell
val init_state = --‘\s: string. 0’--;
new_definition("tstate", --
  tstate envs i = EL (LENGTH envs - i)
```
(EVALR timer envs 'init_state')--

new_definition("triggered", --'
  triggered envs i = (tstate envs i "triggered" = 1')--
new_definition("signal", --'
  signal envs i = (tstate envs i "signal" = 1')--
new_definition("delay", --'
  delay envs i = tstate envs i "delay"')--
new_definition("clock", --'
  clock envs i = tstate envs i "clock"')--
new_definition("deadline", --'
  deadline envs i = tstate envs i "deadline"')--
new_definition("previous", --'
  previous envs i = (tstate envs i "previous" = 1')--

Then, the timer specification is captured by the following HOL term:

val timer_spec = --'!envs k.
  ("(triggered envs 0) /\ "(signal envs 0) /\
   (((0 < k) /\ (k <= LENGTH envs)) ==>
    (triggered envs k = (?i. (i < k)
     /\ "(signal envs i) /\ "(triggered envs i)
     /\ signal envs (i+1)
     /\ (clock envs k < clock envs (i+1)+delay envs (i+1))))))
'--;

Informally, it says that the timer is triggered at time \( k > 0 \) if and only if a signal occurred some time before \( k \), and it will remain triggered until the clock reaches a time which is equal to or greater than the time at which the signal occurred plus the delay specified at that time. Under the premise that the clock produces monotonously increasing time values, stated by

val monotonous = new_definition("monotonous",
  --'monotonous f = (!a b. (a < b) ==> (f a < f b))'--);

the specification turns out to be a provable goal. As the details of the automated proof are of interest to the expert only, we shall only present here a summary of the main stages of the proof.

- First, we evaluate the term EVAL timer_prog, which denotes the semantics of a one-step execution of the timer, to simplify it to some normal form. This is achieved by repeatedly applying re-write rules such as beta reduction, evaluation of comparisons between variable names, and simplification of case statements to this term.

- The result term is then used to derive a recursive characterisation of the states occurring during an execution of the timer program in some environment envs. In particular, it is shown that the values of all variables are initially zero, and for each variable occurring in the program, an equation is proved which characterises its value at some time instant \( k + 1 \) in dependence on the values of the variables at time \( k \) and the environment (i.e., the actual values of the variables clock, signal, and delay) at time \( k + 1 \). These proofs rely on an abundance of lemmata expressing basic facts about list operations and, in particular, indexing of lists.

- Based on these prerequisites, the proof of the timer specification was performed. Its main goal, namely, the logical equivalence characterising whether the timer is triggered or not, was split into two proofs showing the implications from left to right and right to

45
left, respectively. Both proofs are based on induction over the intermediate states, the
main techniques used being implicational reasoning, re-writing, case introductions, and
arithmetic simplifications.

To give the reader a flavour of a mechanised proof in HOL, we present a record of the very last
step of the proof producing the final theorem. This step relies on two already proved theorems,
timer_proof2 and timer_proof4. First, the goal and its premises (which are enclosed in square
brackets) are stated:

\[
\text{set_goal}([-\text{'monotonous (clock envs)'}][-], \text{timer_spec});
\]

After executing the following — admittedly cryptically looking — proof code:

\[
\text{ASSUME_TAC} \text{ timer_proof1 THEN}
\]
\[
\text{UNDISCH_TAC} \quad (\sim(\text{triggered envs 0}) \land \\
\quad \sim(\text{signal envs 0}) \land \\
\quad (0 < k \land k \leq \text{LENGTH envs}) \Rightarrow \\
\quad \text{triggered envs k =} \\
\quad (?i. \\
\quad \quad i < k \land \\
\quad \quad \sim(\text{signal envs i}) \land \\
\quad \quad \sim(\text{triggered envs i}) \land \\
\quad \quad \text{signal envs (SUC i)} \land \\
\quad \quad \text{clock envs k < clock envs (SUC i) +} \\
\quad \quad \quad \text{delay envs (SUC i))})
\]

HOL responds with the final result:

Goal proved.
[monotonous (clock envs)]
\[- \exists k. \\
\quad \sim(\text{triggered envs 0}) \land \\
\quad \sim(\text{signal envs 0}) \land \\
\quad (0 < k \land k \leq \text{LENGTH envs}) \Rightarrow \\
\quad \text{triggered envs k =} \\
\quad (?i. \\
\quad \quad i < k \land \\
\quad \quad \sim(\text{signal envs i}) \land \\
\quad \quad \sim(\text{triggered envs i}) \land \\
\quad \quad \text{signal envs (SUC i)} \land \\
\quad \quad \text{clock envs k < clock envs (SUC i) +} \\
\quad \quad \quad \text{delay envs (SUC i))})
\]

Top goal proved.

10 Summary

In this report we have presented the capabilities and advantages of formal specification and
validation techniques for the development of high quality function blocks. Our emphasis was on
the functional correctness of elementary function blocks that will be used as standard building
blocks in numerous process control applications and, therefore, justify the extra effort necessary
to obtain guaranteed quality criteria. It should have become clear that manual proofs of verifica-
tion conditions, even for relatively simple programs as they occur in function block design, are
tedious and error prone. We have argued that algebraic specification techniques, and OBJ in
particularly, provide an efficient and computer supported way of constructing and validating formal specifications of software requirements and designs. The developers of Affirm [21] have even gone a step further than those of OBJ3 and support automatic proof generation for algebraic specifications. EHDM [31] and m-Eves [12] are other examples of tools that support specification and verification of imperative programs with the aid of automatic proof generators. Both systems are used in the development of security sensitive software, an application domain where the use of formal specification and verification methods is often mandatory.

We tried to demonstrate to the sceptic that formal methods can be used successfully even in today’s industrial practice. Compared with semi-formal or informal specifications, formal specifications differ by the fact that ambiguous, vague and contradictory requirements descriptions can be avoided. Even more important, effective semantic tools can be provided. Apart from these objective advantages, some authors say that formal specifications are of advantage in software practice even if they are not used to prove certain properties of the specifications or the programs developed from them. In [30, 29] and by others it is argued that it is the main advantage of formal specifications that they help, during the specification process, to clarify requirements or problems of system design, to discover latent design errors and ambiguities, and to make decisions on functional properties at the right time. This statement agrees with our own experiences.

We also tried to meet effusive euphoria by the statement that the limits of formal specification techniques and formal proofs must be realised to avoid unsuitable applications and dire expectations. It is reliably true that proofs help to detect analysis and design errors. Proofs, however, only relate mathematical models of a given reality, but not objects of reality. Therefore, they are of no absolute quality that would make them superior to other validation methods. Nevertheless, they constitute important development techniques, whose usefulness is manifested in the interaction with other methods outlined in previous sections. To balance these constraints of formal verification techniques, we presented specification testing as a complementary measure to validate the adequacy of a specification.

Apart from being amenable to extensive machine manipulation, we claimed that formal specifications provide deeper insight into the structure and behaviour of a function block than informal descriptions ever admit. Furthermore, they provide formal concepts to detect ambiguities and incompletenesses of designs. Other advantages of formal techniques, which we did not discuss here, are their ability to support other development paradigms, such as formal program transformation or automatic code synthesis, and the generation of suitable test data from specifications [9, 34, 13].

For algebraic specifications an early approach to a formal transformation technique has been developed by Guttag and his colleagues [27]. They proposed a pattern matching compiler to transform an abstract data type specification into a LiSP program. More recent work in this area is reported in [33, 2, 35]. The first paper describes the synthesis of C implementations, while the second approach transforms algebraic data type specifications into Pascal programs. During specification construction the consistency and sufficient completeness of a specification is interactively verified. Pascal is also the target language of the transformation system presented in the third reference. It is part of the EASE specification environment [44]. The main problem of automatic code generation is the description of a concrete model that meets the specification. This model includes both a suitable representation of the specified data and an implementation of the relevant operations. In general, code synthesis techniques can only be applied if additional information, such as the explicit distinction between constructor and non-constructor operations, is provided or certain restrictions are imposed on the form of equations.

Remarkable results in transformational programming have particularly been produced in the context of wide spectrum programming systems such as CIP [10, 11, 6] or Refine [1] and its successor KIDS [47]. Program transformation systems allow the designer to derive efficiently
executable programs directly from their abstract specifications. As the transformation steps are not deterministic in each case, user interaction is required. An advantage of both development paradigms is that, under the condition that the correctness of the transformation or synthesis rules employed in the program development process is formally verified, the obligation to verify that a resulting program satisfies its specifications becomes dispensable.

References


