Formal Aspects of Computing (1997) 9: 448–468

Formal Aspects of Computing

Theories for Mechanical Proofs of Imperative Programs

Wim H. Hesselink

Department of Mathematics and Computing Science, Rijksuniversiteit Groningen, Groningen, The Netherlands

Keywords: Theorem proving; While theorem; Shared variables; Peterson's algorithm; Fairness

Abstract. For convenient application of a first-order theorem prover to verification of imperative programs, it is important to encapsulate the operational semantics in generic theories. The possibility to do so is illustrated by two theories for the Boyer–Moore theorem prover Nqthm.

The first theory is an Nqthm version of the classical while-theorem. Here the main interest is to show how one can use Nqthm's facilities to constrain and to functionally instantiate for the development and application of a generic theory. The theory is illustrated by a linear search program.

The second theory is a finitary approach to progress for shared-memory concurrent programs. It is illustrated by Peterson's algorithm for mutual exclusion of two processes. The proof of progress for Peterson's algorithm is new. The assertion of bounded fairness is slightly stronger than the conventional notion of weak fairness. This new concept may have other applications.

1. Introduction

The purpose of this paper is to present two theories that can be used in combination with a general purpose theorem prover to certify correctness of imperative programs. The theories are developed for the theorem prover Nqthm of Boyer and Moore, cf. [BoM88]. The imperative programming language used is as simple as possible. It only contains assignment, conditional choice, sequential composition,

Correspondence and offprint requests to: Wim H. Hesselink, Dept. of Mathematics and Computing Science, Rijksuniversiteit Groningen, Postbox 800, 9700 AV Groningen, The Netherlands. http://www.cs.rg.nl/~wim, email: wim@cs.rug.nl

and the repetition (**while**). We use an operational semantics and treat repetitions by means of invariants and variant functions. The first theory is the classical theorem for the total correctness of a repetition. Here the main point is that this classical approach can be conveniently formulated and applied in a first-order theorem prover.

In the second part of the paper, we present a theory for concurrent processes with shared variables. In that case, we use program counters and **goto**'s instead of repetitions, because the prover must argue about arbitrary interleavings of elementary commands.

The language has no nondeterministic constructs. Its expression level contains all Nqthm expressions and is therefore very powerful. We have omitted variable declarations and type restrictions, since the verifications associated can be delegated to compilers (in the case of concurrency, the shared variables are declared to distinguish them from the private ones).

The sequential case is illustrated by a proof of total correctness for a linear search algorithm. For the case of concurrency we prove that Peterson's algorithm establishes mutual exclusion for two processes and that each process gets access to the critical section under bounded fairness, which is a strengthening of weak fairness.

1.1. Motivation

A central goal of computer science is to enlarge the trustworthiness of computer programs. It is well known that testing is important but not sufficient. Formal verification when the design phase is concluded is usually impossible. We therefore prefer formal verification during design.

In all cases however, there is the problem of the trustworthiness of the tester or the verifier. A partial solution to that problem is to use formal verification by means of a mechanical theorem prover of acknowledged quality. For then, anyone who trusts the prover and can read the assertions that the prover verifies, can be convinced of the validity of these assertions without going through the whole proof (cf. [Moo96]).

An important source of programming errors is modification. Programs must be modified, but every modification threatens the correctness. This holds a forteriori for verification during design. In these cases, mechanical correctness proofs are very helpful, for it is often relatively easy to adapt the correctness proof to the modified program. The prover can do the tedious but critical verification that most of the old arguments still work, whereas the human modifier can provide insights at points where new arguments are needed.

The choice of a theorem prover is difficult but important. The prover must be trustworthy and well documented, it must have sufficient intelligence to prove routine lemmas, its language must be rich enough to conveniently express the assertions that have to be proved. This is the minimum: one may also want adequate libraries, a good user interface, good user support, etc.

We are working with the Boyer–Moore theorem prover Nqthm, cf. [BoM88, BoM92], which satisfies at least our minimum requirements. It also has extensive libraries, good user support, and a large user group. An important point is that its expression language is relatively poor, we come back to this below. A positive point is that it is freely available (public domain) and that it runs on all kinds of platforms. It has a rather primitive `emacs` user interface.

Our goal is verified design of programs (algorithms). So we assume that the program is executed on a reliable system. Our goal is thus complementary to the goal of a reliable and verified system, as represented by [Moo96].

The classical methods to prove program correctness rely on Hoare triples or Dijkstra's weakest preconditions, cf. [Dij76, Gri81, Kal90]. In both cases one has to *calculate* with predicates. Therefore predicates must not be treated as syntactic objects, but as boolean functions. Hoare triples are boolean functions on pairs of predicates. Weakest preconditions are functions from predicates to predicates (predicate transformers). Therefore these methods need functions of functions, i.e. higher order functions.

There are good theorem provers that can work with higher order functions, e.g., HOL [GoM93], PVS [OSR93], etc. A logic with higher order functions requires a sophisticated type system to avoid paradoxes. In fact, it must not allow the definition of a boolean function `p` with the property that `(p x) = (not (x x))`, since then the value of `(p p)` would be its own negation. In a theorem prover, higher order facilities make the number of possible choices in a proof bigger. This should make it harder to guide the prover intelligently or to provide it with adequate heuristics, but at this point we do not speak from own experience with such provers.

The prover Nqthm is at the other extreme: it does not have higher order functions, and it is untyped in the sense that every function with arity r can be applied to every sequence of r values. Indeed Nqthm's great theorem proving power is based on this simplicity. On the other hand, Nqthm has facilities that mimic higher order logic: it provides the possibility to functionally instantiate the results of an axiomatic theory and it has a function `eval$` that can interpret syntactic terms. We shall use these two facilities in a crucial way.

1.2. Ways to Use a Prover to Reason about Programs

As indicated above, we have chosen to use an existing reliable and powerful theorem prover as a starting point. There are many ways of adapting an existing tool to construct a proof checker for a given language, see [SkS93]. One approach is to use the existing tool as a subroutine to a newly constructed proof checker. This approach is used in the TLA [Lam94] proof checker where the Larch Prover (LP) [GaG88] is used as a back-end theorem prover [EGL92]. It is also possible to encapsulate the prover by providing a special purpose interface, i.e. a parser/unparser which translates the programming logic into the logic of the prover and vice versa. This is the way the duration calculus proof assistant of [SkS93] is built on top of the Prototype Verification System (PVS) of [OSR93]. It is also possible to express the semantics of the programming language directly in the language of the prover, as is done for instance by Hooman [Hoo94], who also uses the theorem prover PVS. Other projects have used HOL in the same way, e.g., [Gor89, BaW90].

We have chosen not to hide or encapsulate the prover. There are two reasons. Firstly, the user must be able to guide the prover. In our view, this requires the user to have knowledge and experience with the prover and its language. We expect that an encapsulation could work nicely for simple problems, but become a hindrance when the proof is harder.

A second argument against hiding is that we expect the proof system to yield a certificate of validity. If the prover is accepted as sound, an input file that leads to a complete proof may be regarded as a certificate, at least if the proof obligations

are adequate and complete, and the axioms introduced are valid. When the prover is encapsulated in a new tool, however, the soundness of the combination is questionable again.

We have therefore chosen to work directly with the prover. The question then arises how to represent programs and specifications of programs. As mentioned above, Hooman [Hoo94] uses a semantic encoding. For example, on p. 30, he encodes

```
r:= r − y ; z:= z+1
```

as the function

```
seq(assign(r, LAMBDA s: val(s)(r) − val(s)(y)),
    assign(z, LAMBDA s: val(s)(z) + 1) )
```

In [BaW90], almost the same representation is used, but function `val` is not needed.

Of course, such an encoding can be generated mechanically by some preprocessor, but then the soundness of the preprocessor is a new proof obligation. We therefore prefer to use a syntactic encoding of the program. Below we propose an imperative programming language in which the above fragment is written as

```
'( (put r (difference r y))
   (put z (plus z 1)) )
```

This is much closer to an actual programming language. In particular, the state variable `s`, the functions `val` and `seq`, and the lambda abstraction have been removed. In a simple program, the semantic overhead may be acceptable, but in larger programs it can soon become unmanageable. In [Hes95], we had a shared-variable distributed algorithm of 32 elementary commands. In [Hes96] we treat a message-passing tree algorithm (GHS) with 11 different messages for which the syntactic representation fills two pages. In such cases the correspondence between the syntactic representation and the semantic object becomes a new source of programming errors.

Next to syntax comes semantics. If one wants to prove theorems about programs, the semantics of the programming notation must be well defined. There are many ways to define semantics of programming languages: operational, denotational, axiomatic, etc. For work with a first-order theorem prover, the easiest way is to use operational semantics, i.e. to define the meaning of a program by means of an interpreter which is a function in the logic of the prover, cf. [BoM88], Section 3.6, and [Moo96].

1.3. Working with Nqthm

In this subsection we briefly sketch how to work with Nqthm. We refer to the Handbook [BoM88] for more information.

The interaction with Nqthm is a dialogue in which the user submits definitions and lemmas to the prover and the prover ideally answers by accepting the definitions and proving the lemmas. In this way a data base of known facts (rewrite rules) is built. In practice the user often submits a lemma the prover cannot prove (often because it is not valid). In that case, the user must try and diagnose the failure by inspecting Nqthm's output. It may be helpful to submit an auxiliary lemma first, or to instruct Nqthm that some definition must (or must not) be unfolded.

Nqthm is deterministic and it backtracks only when it starts a proof by induction. Therefore the search path of Nqthm is very important. This path can be influenced, implicitly or explicitly, by many decisions of the user. For example, the

order of the hypotheses of a lemma may influence the way the lemma can be used as a rewrite rule later.

The result of proof design with Nqthm is a so-called event file. Such a file is an input file for the prover with a theory consisting of definitions, axioms, and theorems (lemmas). It is an ordinary text file that can be inspected and modified. If well designed, submission of this file to Nqthm leads to a mechanical proof of the theorems in the theory, without any user interaction. In this way the event file is a certificate of the correctness of its theory. Event files either start from scratch or start from a library file created by other event files.

In this paper we describe four event files. The files `while` and `weakfairness` start from scratch, whereas `linsearch` builds on `while`, and `peterson` builds on `weakfairness`. These event files are available from our Web site [Hes@]. They are rather small, together less than 900 lines.

1.4. Generic Theories and Functional Instantiation

In a higher order theorem prover, one can universally quantify over functions and also use embedded universal quantification. So, one may have a theorem saying that, for every function g, always A implies always B, i.e.

$$(\forall g \in R \to S :: (\forall x :: A) \Rightarrow (\forall y :: B))$$

Here A and B are formulae that may mention function g. Such a theorem can then be used by instantiating g with a concrete function $g1 \in R \to S$. In a first order logic as Nqthm, this is not directly possible. Nqthm provides, however, the possibility to declare an undefined function symbol g, to postulate the axiom A which is then automatically universally quantified over x, to prove B as a theorem, and finally to use the theorem by functional instantiation of $g1$ for g. Of course, the last step requires verification that A holds when g is replaced by $g1$.

The unrestricted introduction of axiom A has the danger that A may be inconsistent. Then the functional instantiation may even succeed, since false is provable from A. The authors of Nqthm therefore strongly discourage the use of unrestricted axioms. Instead they offer a harmless way to postulate axioms by means of the `constrain` command. This serves as an axiom, but the user is forced to submit a witness for the undefined function symbols, for which the axiom can be proved. The witness is often completely trivial and uninteresting.

We use the `constrain` facility to develop two generic theories. The first theory is the classical theorem for total correctness of a repetition. The main interest of this case is the applicability of a generic theory.

The second theory is a treatment of concurrency with shared variables and interleaving semantics. Here one has to formalize the assumption that all processes make progress. For this purpose we develop a finitary theory of fairness. Such a finitary theory is convenient but not strictly necessary for the treatment of fairness with Nqthm. See Russinoff [Rus92] for an alternative.

1.5. Nqthm Notation

Many proofs published by Boyer, Moore, and their associates (e.g., [Moo94]) rephrase definitions and lemmas into more traditional notation to make them more broadly accessible. Since we want to explain how Nqthm can be used to verify proofs, we cannot always do so. Actually, it is our experience, e.g., while reading

[Moo94] with a group of non-users of Nqthm, that rendering into more traditional notation frequently raises questions that can only be answered by discussion of the Nqthm notation. Finally, this paper relies on the interpretation of quotations of terms for which a more traditional notation could be utterly confusing.

2. Representation and Modification of the State

In this section we describe the construction of the interpreter, but first, briefly, some aspects of Nqthm's language and logic. Nqthm's language is a dialect of pure LISP. In particular, all operators are prefix functions. The application of a function `fn` on two arguments `x` and `y` is denoted `(fn x y)`.

Nqthm's truth values are `(true)` and `(false)`, which can be abbreviated to `t` and `f`. The most fundamental logical operator is "`if`", characterized by the axiom that `(if x y z)` equals `z` if `x` = `f`, and otherwise `y`. In fact, Nqthm is weakly typed and its functions are almost always total: when `x` is not a truth value, the term `(if x y z)` is well defined and also equal to `y`. The only nontotal functions are relatives of `eval$`, see below.

2.1. Association Lists and the Evaluator

The operational semantics of an imperative programming language requires the concept of state. The state of a computer program determines the values of all program variables. So it can be represented by an association list, a list of pairs where each pair consists of a variable and the associated value. The function that retrieves the first pair with given key (first element) is defined by

```
(assoc key x) =
    (if (nlistp x) f
        (if (equal key (caar x)) (car x)
            (assoc key (cdr x)) ) )
```

The first line says that `(assoc key x)` = `f` if `x` is empty when regarded as a list. If `x` is nonempty, the first pair is `(car x)` and the first key is `(car (car x))`, abbreviated `(caar x)`. If `key` differs from `(caar x)`, function `assoc` is called recursively on the tail of list `x`, denoted by `(cdr x)`. If we only need the value associated to a given key, we may use function `lookup` given by

```
(lookup key x) = (cdr (assoc key x))
```

Our event file `while` begins with the definition of a function to modify the state. This function is called `putassoc`, and is given by

```
(putassoc var w x) =
  (if (nlistp x) (cons (cons var w) nil)
      (if (equal var (caar x))
          (cons (cons var w) (cdr x))
          (cons (car x) (putassoc var w (cdr x))) ) )
```

If `x` represents the empty list, the first line specifies that the result is an association list with the pair `(cons var w)` as its only element. Otherwise, the first key-value pair with key equal to `var` gets the new value `w`. After these definitions Nqthm easily proves the crucial identity

```
(assoc key (putassoc var w x)) =
(if (equal key var) (cons var w) (assoc key x) )
```

Below we need the Nqthm function `list` which takes an arbitrary number of arguments and returns the corresponding list:

```
(list) = nil ;
(list a .. z) = (cons a .. (cons z nil)..) .
```

Nqthm has a function `eval$` to interpret quotations of terms where the interpretation of atoms is given by an association list. For example,

```
(eval$ t '(plus 5 vii) x) = 12
```

if `(lookup 'vii x) = 7`. Function `eval$` is partial: `(eval$ t '(gg 5) x)` is only defined if the user has defined a function `gg`. These examples may be sufficient for our purposes. We don't want to explain all intricacies of quotations of terms and of the function `eval$` (for our purposes, the first argument of `eval$` can always be `t`). Yet we have to say more about quotation of terms.

In computer science we often need to argue about big nested structures like expressions or programs. For these purposes the Nqthm logic has an abbreviation convention that uses the symbol `quote`, abbreviated `'`. If `X` is a symbol or an S-expression, then `(quote X)` or shorter `'X` is usually the quotation of a term. This goes as follows. The quotation of a symbol like `plus` is the literal atom `'plus`. Now, recursively, if `'X, 'Y, ..., 'Z` are abbreviations of terms, then `'(X Y ... Z)` is the abbreviation of `(list 'X 'Y ... 'Z)`. For example,

```
(car   '(plus 5 vii)) = 'plus
(caddr '(plus 5 vii)) = 'vii
```

We refer to [BoM88] for more and more precise information.

2.2. The Interpreter and Linear Search

We now construct an interpreter for while–programs. We use a well known linear search program as an example. So, assume given an unknown array $a[0\,..\,n)$ and a value y. The program must determine the least index m with $a[m] = y$. If value y does not occur in the array, then $m = n$ must hold in the postcondition. We use the program

$k := 0;\quad m := n;$
while $k \neq m$ **do**
  if $a[k] = y$ **then** $m := k$ **else** $k := k + 1$ **fi**
od

The postcondition required is

$$(m = n \quad \vee \quad a[m] = y) \quad \wedge \quad (\forall\, i : 0 \leqslant i < m : a[i] \neq y)$$

We construct an interpreter `exe` for a language in which the above program can be written:

```
(linsearch) =
    '( ( (put k 0) (put m n) )
       (while (not (equal k m))
              (if (equal (lookup k a) y)
                  (put m k)
                  (put k (add1 k)) ) ) ) )
```

The first two `put` expressions are the initializing assignments to `k` and `m`. They are taken together to form the initialization. The expression `(lookup k a)` stands for $a[k]$; here we assume that the value of `'a` is an association list.

We now define a command interpreter `exe`. It is a function that modifies the global state according to the command it executes. We model the global state as an association list `x` which binds values to the program variables, which for `linsearch` are `'k`, `'m`, `'n`, `'y`, and `'a`. The expressions in the program are evaluated by `eval$` with respect to the global state. The assignments (`put`) are performed by means of the function

```
(modify var exp x) = (putassoc var (eval$ t exp x) x)
```

Since a while–program need not terminate, we give the interpreter `exe` an argument `rd` to bound the recursion depth. We let `exe` yield `f` in case of nontermination or too large recursion depth. Since `x = f` stands for nontermination, this state must always yield the final state `f`. In this way we arrive at the definition

```
(exe rd cmd x) =
    (cond ((or (zerop rd) (not x)) f)
          ((nlistp cmd) x)
          ((nlistp (car cmd))
           (case (car cmd)
                 (put (modify (cadr cmd) (caddr cmd) x))
                 (if (if (eval$ t (cadr cmd) x)
                         (exe rd (caddr cmd) x)
                         (exe rd (cadddr cmd) x) ))
                 (while (if (eval$ t (cadr cmd) x)
                            (exe (sub1 rd) cmd
                                 (exe rd (caddr cmd) x) )
                            x ))
                 (otherwise f) ) )
          (t (exe rd (cdr cmd) (exe rd (car cmd) x))) )
```

The functions `cond` and `case` serve to make case distinctions. Function `cond` expects its arguments to be pairs. It yields the `cdr` of the first pair for which the `car` has a value $\neq$ `f`. The `car` of the last pair must be equal to `t`. The function `case` evaluates its first argument and treats the remaining arguments as an association list. The `case` expression in `exe` distinguishes the constructors `'put`, `'if`, and `'while`. The last case of the `cond` expression is the sequential composition of a list of commands. Notice that first the head (`car`) of the list is executed and then the tail (`cdr`).

Termination of function `exe` follows from the fact that in each recursive call the sum of `rd` with the size of `cmd` is smaller. For easy recursive definitions Nqthm finds such a termination measure automatically, but in this case it needs a hint.

In Nqthm (unlike LISP), the `car` of a nonlist is welldefined and equal to 0. It follows that in our interpreted language, the `if` statement with guard *false* without **else** branch is equivalent to *skip*.

For reasoning about program `linsearch`, we give names to its components

```
(init0)    = '((put k 0) (put m n))
(guard0)   = '(not (equal k m))
(body0)    = '(if (equal (lookup k a) y) (put m k)
                  (put k (add1 k)) )
(linloop)  = (list 'while (guard0) (body0))
```

Now one can prove that `(linsearch)` equals `(list (init0) (linloop))`.

In order to argue about the values of the variables `'n`, `'m`, `'y`, `'k`, `'a`, we define corresponding state functions `nn`, etc., by

```
(nn x)    = (lookup 'n x)
(aa i x)  = (lookup i (lookup 'a x))
```

and `mm`, `yy`, `kk` in the same way as `nn`. Notice that `nn`, `yy`, `aa` depend on the state `x`, although the program does not change them.

Nqthm can now directly prove that the initialization works as expected:

```
(prove-lemma exe-init0 (rewrite)
 (implies (and x (not (zerop rd)))
       (and (exe rd (init0) x)
         (equal (mm (exe rd (init0) x)) (nn x))
         (equal (nn (exe rd (init0) x)) (nn x))
         (equal (yy (exe rd (init0) x)) (yy x))
         (equal (aa i (exe rd (init0) x)) (aa i x))
         (equal (kk (exe rd (init0) x)) 0) ) ) )
```

Here we give the command to prove as it is submitted to Nqthm. Every lemma needs to have a name, here `exe-init0`. The argument `(rewrite)` means that the lemma if proven can be used later as a rewrite lemma. Notice that we exclude the virtual state `x = f` initially and, hence, as a resulting state.

2.3. A General Theory for the Repetition

The repetition requires more work than the initialization. Since repetitions are always proved in the same way, we develop a general theory for repetitions of the form **while** *B* **do** *S*. For this purpose we axiomatically introduce five Nqthm functions. We represent the guard *B*, the body *S*, the invariant, and the variant function by functions `guard`, `lbody`, `invariant`, and `avf`, respectively. We also need a function `upbexe` as an upper bound of the recursion depth for the execution of the body.

The axiom is submitted in such a way that no inconsistency in the database can be generated. For this purpose, the axiom is accompanied by a list of instantiations for the functions, such that the axiom is satisfied.

```
(constrain while-axiom (rewrite)
  (and (implies (and (invariant x)
                     (eval$ t (guard) x) )
                (and (invariant (exe (upbexe) (lbody) x))
                     (lessp (avf (exe (upbexe) (lbody) x))
                            (avf x) ) ) )
    (not (zerop (upbexe)))
    (not (invariant f)) )
  ((invariant (lambda (x) f)) ; an instantiation
    (avf      (lambda (x) 1))
    (upbexe   (lambda () 1))
    (guard    (lambda () (false)))
    (lbody    (lambda () nil)) ) )
```

The axiom consists of three conjuncts: the first one says that, if the guard holds, execution of `lbody` preserves the invariant and decreases the variant function. The

requirement that upbexe differs from 0 is mainly for convenience. The third requirement says that the invariant does not hold in the virtual state f. Therefore, the first requirement implies termination of the body within upbexe unfoldings.

It is straightforward to verify that the axiom holds for the trivial instantiation provided. The axiom is used to argue about the repetition given by

```
(loop) = (list 'while (guard) (lbody))
```

In fact, the axiom is used to prove that, if the invariant holds initially and the number rd is sufficiently large, the loop terminates in a state where the invariant and the negation of the guard hold. This is expressed in

```
(prove-lemma invariant-theorem (rewrite)
    (implies (and (not (lessp rd (plus (avf x) (upbexe))))
                  (invariant x) )
             (and (not (eval$ t (guard) (exe rd (loop) x))
                  (invariant (exe rd (loop) x)) ) ) )
```

2.4. The Application of the General Theory

The invariant theorem can be applied in concrete cases by giving instantiations for the five functions introduced in while-axiom. Of course, the validity of the instantiated axiom must then be verified. This is exemplified in the concrete case of linsearch. The postcondition required is formalized in

```
(post0 i x) =
    (and x
         (or (equal (mm x) (nn x))
             (equal (aa (mm x) x) (yy x)) )
         (implies (and (numberp i) (lessp i (mm x)))
                  (not (equal (aa i x) (yy x))) ) )
```

The first conjunct of post0 expresses that the final state must differ from f, i.e., that the program must terminate. The second conjunct says that $m = n$ or $a[m] = y$. The third conjunct says that $a[i] \neq y$ holds for arbitrary i with $0 \leqslant i < m$.

We define the invariant jq* with arguments i and x as the conjunction of invariants representing

(Jq0) $m = n \quad \vee \quad a[m] = y$,
(Jq1) $0 \leqslant i < k \quad \Rightarrow \quad a[i] \neq y$,
(Jq2) $k \in \mathbb{N} \quad \wedge \quad m \in \mathbb{N}$,
(Jq3) $m \geqslant k$,
(Jq4) $x \neq f$.

For example (Jq1) is represented by

```
(jq1 i x) =
    (implies (and (numberp i) (lessp i (kk x)))
             (not (equal (aa i x) (yy x))) )
```

The invariant (Jq4) is unusual: it expresses that the body of the loop always terminates.

Remark. We number the invariants consecutively, so that during the design it is easy to see whether all current invariants have been treated. They get names with second

letter q, so that even in large files they can easily be located and renamed when necessary.

Having defined the invariants, we also construct a variant function $\mathtt{vf} = m-k$, and then prove that the `while-axiom` holds in our application. The first conjunct is expressed in

```
(prove-lemma while-lemma (rewrite)
    (implies (and (jq* i x)
                  (eval$ t (guard0) x) )
             (and (jq* i (exe 1 (body0) x))
                  (lessp (vf (exe 1 (body0) x))
                         (vf x) ) ) ) )
```

The other two conjuncts of `while-axiom` are easier. Now the axiomatic theory can be applied in the Nqthm command

```
(functionally-instantiate linsearch-loop (rewrite)
  (implies (and (not (lessp rd (plus (vf x) 1)))
                (jq* i x) )
           (and (not (eval$ t (guard0) (exe rd (linloop) x)))
                (jq* i (exe rd (linloop) x)) ) )
  invariant-theorem
  ((invariant (lambda (x) (jq* i x))) ; the instantiation
   (upbexe    (lambda () 1))
   (guard guard0) (avf vf) (lbody body0) (loop linloop) ) )
```

This is a functional instantiation of the invariant theorem determined by the association list with `invariant` as its first key. For simplicity we have omitted the hints to the prover. In this case the body of the loop is a straightline command. Therefore we can take `upbexe` equal to 1, as shown in `while-lemma`. Nested loops can also be treated, but then termination of the body needs bigger recursion depth.

It is easy to verify that the conclusion of `linsearch-loop` implies the post-condition `post0`. Combining these results with lemma `exe-init0`, we finally obtain correctness of `linsearch`:

```
(prove-lemma linsearch-correct (rewrite)
     (implies (and x ; the initial state is valid
                   (lessp (nn x) rd)
                   (numberp (nn x)) )
              (post0 i (exe rd (linsearch) x)) ) )
```

Unfortunately, if this assertion is the specification of `linsearch`, the following program is an easier implementation

$m := 0;\quad y := a[0]$, that is

```
'( (put m 0) (put y (lookup 0 a)) ).
```

To avoid such unsatisfactory implementations, we also specify that the program variables `n`, `a`, and `y` must not be modified. For this purpose, we introduce a function `writtenvars` that, given a command, yields the list of variables that are threatened to be modified by the command. This function satisfies

```
(writtenvars (linsearch)) = '(m k)
```

It follows that `n`, `a`, and `y` are constants for `linsearch`.

3. Multiprogramming

We now show how one can treat sequential processes that communicate by means of shared variables. So we have a number of processes, each of which executes a sequential program, which is divided into atomic statements. The processes execute the atomic statements interleaved in an arbitrary way. The global state consists of the values of the shared variables, together with the values of the private variables of the processes. A *transition* of the system is a step from one state to another in which one process executes an atomic statement. An *execution* of the algorithm is a sequence of transitions that starts in some initial state. A state is called *reachable* if it occurs in an execution.

An *invariant* is defined to be a predicate that holds in all reachable states. Following [Tel94], we write $\{P\} \rightarrow \{Q\}$ to denote that every atomic step of the algorithm that starts in a state where P holds, terminates in a state where Q holds. We define a predicate P to be a *strong invariant* if it holds initially and satisfies $\{P\} \rightarrow \{P\}$. It is easy to see that every predicate implied by a strong invariant is itself invariant. Note that Tel ([Tel94] p. 51) calls invariants what we call strong invariants.

We now go into the Nqthm modelling of such systems. As above we want to use function `eval$` for the evaluation of expressions, but now we want that, when process q evaluates an expression, it uses the values of its own private variables. The shared variables must be visible to all processes. In order to force the right distinction between shared variables and private variables, we use a declaration, say `d`, of the list of shared variables and we filter the shared state, say `z`, by means of function `cleanalist` given by

```
(cleanalist d z) =
    (if (nlistp d) nil
        (cons (cons (car d) (cdr (assoc (car d) z)))
              (cleanalist (cdr d) z) ) )
```

We then structure the global state `x` as a pair of association lists, one for the private states and one for the shared variables. We thus define the shared state and the private states by

```
(shared    d x) = (cleanalist d (cdr x))
(privstate q x) = (cdr (assoc q (car x)))
```

We define the function `ev` that yields the value of expression `exp` for process `q` in global state `x`, according to declaration `d`, by

```
(ev d q exp x) =
    (eval$ t exp
           (append (shared d x)
                   (cons (cons 'self q)
                         (privstate q x) ) ) ) )
```

Thus, as noticed by a referee, if a private variable clashes with a shared variable, the shared variable takes precedence.

All processes execute the same program. The above definition binds the constant `self` to the executing process. We construct a function `putgen` with arguments `d`, `q`, `var`, `exp`, `x`, which modifies the global state `x` by binding the value

of `exp` according to `d`, `q`, `x` to the variable `var`, which is treated as a global variable if it belongs to `d`, and as a private variable otherwise.

Then we construct a function `exec` analogous to the sequential version `exe` in 2.2, but now without a **while** statement, and with a **case** statement. Just as in the sequential case, the `if` statement with guard *false* without **else** branch is equivalent to *skip*.

We give each process a private variable *pc*, its program pointer. We write *pc.q* to denote *pc* of process *q*. Prior to execution of a command by process *q*, the value of *pc.q* is incremented by default with 1. For this purpose, we subject the program to function `add1pc`, defined by

```
(add1pc-prim prog) =
  (if (nlistp prog) nil
    (cons (cons (caar prog)
                (cons '(put pc (add1 pc)) (cdar prog)) )
      (add1pc-prim (cdr prog)) ) )

(add1pc prog) = (list* 'case 'pc (add1pc-prim prog))
```

For example, if `prog` is the program

```
'( (0 (put y (add1 y)))
   (1 (put pc 0)) )
```

then `(add1pc prog)` equals

```
'(case pc
       (0 (put pc (add1 pc)) (put y (add1 y)))
       (1 (put pc (add1 pc)) (put pc 0)) ).
```

3.1. Peterson's Algorithm

Peterson's algorithm [Pet81] is the simplest algorithm for mutual exclusion for two processes. The algorithm establishes that two processes (say 0 and 1) are never at the same time in the state *crit*, while it enables each of them to reach *crit* from time to time.

For this purpose it uses two shared variables *active*: **array**[0 . . 1] **of** *boolean* and *you*: {0, 1}. A process may enter state *crit* when it equals *you* or when the other process is not *active*. A process that "wishes" to access state *crit* assigns to *you* the name of the other process, as if it were politely saying "after you".

In the state with *pc.q* = 0, process *q* has the nondeterminate choice between staying asleep or trying to access *crit*. It may remain asleep forever, but there is never a guarantee that it remains asleep forever. This part of the program belongs to the modelling, not to the protocol. The program is as follows.

0 **goto** 0 [] *active*[*self*] := *true*;
1 *you* := 1 − *self*;
2 **if** *you* = *self* **then goto** 4;
3 **if** *active*[1 − *self*] **then goto** 2;
4 *active*[*self*] := *false*; **goto** 0.

State function *crit* is defined by

$$crit.q \equiv pc.q = 4$$

Mutual exclusion is therefore expressed by the invariant

(Jq0) $\neg(pc.0 = 4 \ \wedge \ pc.1 = 4)$

Remarks. In this program, the processes leave the critical section "immediately", that is, in the next step they take. Yet, since the model is untimed, they may remain in the critical section for an arbitrarily long period of time. In this period they may perform critical actions, but these actions need not be modeled in the program.

The tests in lines 2 and 3 of the program are often combined in one test;

wait until $\neg active[1-self] \vee you = self$

(cf. [ApO91] p. 285). This has the disadvantage that two shared variables must be evaluated in one atomic statement. With respect to safety it is obvious that the two versions are equivalent; but this is less obvious for the progress requirement. See also [ChM88] pp. 362–367.

In order to prove the invariance of (Jq0), we postulate the invariants

(Jq1) $active[q] = (pc.q \in \{1,2,3,4\})$,
(Jq2) $\neg(pc.you \in \{2,3,4\} \ \wedge \ pc.(1-you) = 4)$.

For the proof of (Jq2) we need the obvious invariant

(Jq3) $you \in \{0,1\}$.

We define (Jq*) to be the conjunction of (Jq0), (Jq1), (Jq2), (Jq3). The initial state satisfies (Jq*). Every atomic action preserves (Jq*), that is $\{(Jq^*)\} \rightarrow \{(Jq^*)\}$. Therefore, (Jq*) is a strong invariant, and the predicates (Jq0), (Jq1), (Jq2), (Jq3) are invariants, as required.

For the mechanical verification, we model the nondeterminate choice at $pc = 0$ by means of a private variable `oracle`:$\mathbb{N}$ together with an unknown boolean function `wait`. To guarantee that it has a new value each time it is inspected, `oracle` is incremented after each inspection. We model the array a as an association list. We thus get the program

```
(peterson) =
    '( (0 ((if (wait oracle)
               (put pc 0)
               (put active (putassoc self (true) active))
           (put oracle (add1 oracle)) ))
       (1 (put you (other self)))
       (2 (if (equal you self) (put pc 4)))
       (3 (if (lookup (other self) active) (put pc 2)))
       (4 (put active (putassoc self (false) active))
          (put pc 0) ) )
```

Here `(other p)` stands for the process not equal to `p`.

The semantics of this program are determined by the declaration of the shared variables `dcl-p` and the function `step`.

```
(dcl-p)     = '(you active)
(step p x)  = (exec (dcl-p) p (add1pc (peterson)) x)
```

The remainder of the mechanical proof of safety of the algorithm goes as follows. We define state functions to express the values of the variables (both private and shared). We formulate lemmas to express how these values change under the atomic steps of the algorithm (these lemmas are useful later on, but also form an important

check on the correctness of the modelling used). We then define the invariants. For example, invariant (Jq1) is expressed by

```
(jq1 q x) =
    (equal (lookup q (active x))
           (member (pc q x) '(1 2 3 4)) )
```

and its invariance is proved in

```
(prove-lemma jq1-kept-valid (rewrite)
    (implies (jq1 q x)
             (jq1 q (step p x)) ) )
```

Similarly, (Jq2) is expressed by

```
(jq2 x) =
    (not (and (equal (pc (other (you x)) x) 4)
              (member (pc (you x) x) '(2 3 4)) )) )
```

Its proof of invariance is more involved

```
(prove-lemma jq2-kept-valid (rewrite)
    (implies (and (jq2 x)
                  (jq3 x)
                  (member p '(0 1))
                  (jq1 (other p) x) )
             (jq2 (step p x)) ) )
```

Notice that (Jq1) is a strong invariant, but (Jq2) is not.

We then form the conjunction `jq*` to represent (`Jq*`), and we prove that it is a strong invariant. It follows, for example, that `jq1` is an invariant. For details we refer to the event file `peterson`.

The invariance of (Jq0) expresses mutual exclusion, the essential safety property. Mutual exclusion can be trivially established, however, by precluding one of the processes to enter *crit*. Peterson's algorithm is better than that. It also satisfies the progress requirement that, if one of the processes, say q, indicates the "wish" to enter *crit* and if both processes do sufficiently many steps, then process q indeed will enter *crit*. So we also have to formalize and prove this progress requirement.

A process q indicates its wish to enter *crit* by making $pc.q = 1$. So, there is a pair of predicates, P and Q, on the state, and we want to prove that, if an execution starts in a state where P holds and all processes perform sufficiently many actions, the execution contains a state where Q holds.

In order to prove this progress requirement we first formalize it in the more general context of bounded fairness for a finite number of processes. The traditional notion of weak fairness says that every execution sequence that starts in a state where P holds and in which every process performs infinitely many actions, contains a state where Q holds. In Nqthm it is not convenient to argue about infinite sequences. We therefore prefer a finitary and slightly stronger notion of bounded fairness, which only needs finite sequences.

3.2. Progress under Bounded Fairness

We write $q: x \to y$ to indicate that, if process q performs an action in the (global) state x, the resulting (global) state can be y. We write $q: \{P\} \to \{Q\}$ to indicate that,

if process q performs an action in a state where predicate P holds, the action necessarily terminates in a state where Q holds. As announced above, we write $\{P\} \to \{Q\}$ to indicate that $q: \{P\} \to \{Q\}$ holds for all processes q.

A *schedule* is a sequence of process names. An *execution sequence* is a sequence of (global) states. An execution sequence $y = (y_0, \ldots, y_n)$ *satisfies* a schedule $s = (s_0, \ldots, s_{n-1})$, iff we have $s_i: y_i \to y_{i+1}$ for all $i < n$.

We say that a state x *leads to* predicate Q under schedule s iff every execution sequence y with $y_0 = x$ that satisfies schedule s, contains some state y_i where Q holds (it need not be the last one).

Let L be a set of process names. A schedule s is called 1-*fair for* L iff it contains all elements of L. The schedule is called k-*fair for* L iff it is a catenation of k schedules that are each 1-fair for L. Here, it is not sufficient to require that the schedule contains all elements of L at least k times.

We say that predicate P *leads to* Q *under bounded fairness for* L iff, for every state x where P holds, there is a number k such that x leads to Q under every k-fair schedule for L. If we omit L, we mean that L is the set of all processes.

Remark. Note that we assume that every process is always enabled, although its action may be equivalent to *skip*. Under this assumption it is not hard to prove that, if P leads to Q under bounded fairness, it also does so under weak fairness.

In fact, let t be an infinite fair execution sequence starting in a state where P holds. Then t has a prefix in which every process acts at least once. This prefix is 1-fair. The corresponding suffix is still fair. By induction, it follows that t has a k-fair prefix (for every k). Therefore t contains a state where Q holds, thus proving weak fairness.

Bounded fairness is stronger than weak fairness. For example, consider two processes $q0$ and $q1$ with shared integer variables y and z, given by

$q0$: $*[y := y + z]$
$q1$: $*[z := -1]$

Here, the star * means infinite repetition. Precondition $y = z = 1$ leads to $y = 0$ under weak fairness, since $q1$ will be executed while $y \geqslant 0$ holds and then $q0$ can decrement y until $y = 0$. There is no number k such that $y = z = 1$ leads to $y = 0$ under every k-fair schedule (since $q0$ may act more than k times before $q1$). Therefore, $y = z = 1$ does not lead to $y = 0$ under bounded fairness.

The above definition allows the following rule to prove *leadsto*–relations. It is inspired by the rules for UNITY, cf. [ChM88].

Theorem. Let P and Q be predicates and let L be a set of processes. Let *avf* and *hot* be state functions, *avf* with integer values, and *hot* with process values.

Assume that

(i) P implies $avf > 0$,
(ii) $\{avf > 0\} \to \{avf > 0 \vee Q\}$,
(iii) $avf > 0$ implies $hot \in L$,
(iv) for every pair of processes p, q, and every positive integer V, we have

$$p: \{avf = V \wedge hot = q\} \to \{avf < V \vee (avf = V \wedge hot = q \wedge p \neq q)\}.$$

Then predicate P leads to Q under bounded fairness for L.

Remarks. As above, *avf* stands for abstract variant function. State function *hot* refers to the process that is responsible for decrementing *avf*.

It is clear that, if predicate P implies P' and P' leads to Q, then P leads to Q. Without loss of generality, we may therefore replace predicate P by $avf > 0$. In that

way, the theorem gets simpler. We prefer to give the above version, however, since it more clearly separates the specifying predicates P and Q from the auxiliary state functions *avf* and *hot*.

Condition (iv) is equivalent to the conjunction of three conditions. For every pair of processes p, q, and every positive integer V:

(a) If *avf* is positive, it descends: $\{avf = V\} \to \{avf \leqslant V\}$.

(b) If *avf* is positive and process *hot* acts, *avf* decreases:

$$p: \{avf = V \wedge hot = p\} \to \{avf < V\}.$$

(c) If *avf* is positive, then *hot* is constant unless *avf* decreases:

$$\{avf = V \wedge hot = q\} \to \{avf < V \vee hot = q\}.$$

Proof of the theorem. Let x be a state where P holds. Put $k = avf.x$. By (i), we have $k > 0$. Let s be a k-fair schedule for L. It suffices to prove that x leads to Q under schedule s.

It follows from (iii) and (iv) that, if z is a state with $avf.z = V > 0$, then z leads to $avf < V$ for every 1-fair schedule for L. In fact, $hot \in L$ and *hot* remains constant while *avf* does not decrease. If process *hot* itself acts, *avf* decreases.

Since schedule s is a catenation of k 1-fair schedules, every execution y with $y_0 = x$ that satisfies schedule s contains a state y_i with $avf.y_i \leqslant 0$. By condition (ii), the first state with this property satisfies predicate Q. □

The event file `weakfairness` contains the mechanical proof of this theorem. This proof starts with the axiom that represents the assumption (iii) and (iv) of the theorem:

```
(constrain avf-axiom (rewrite)
 (and (numberp (avf x))
   (implies (not (equal (avf x) 0))
         (and (member (hot x) (plist))
           (lessp (avf (next (hot x) x))
               (avf x) )
           (not (lessp (avf x) (avf (next p x))))
           (implies (not (lessp (avf (next p x))
                                (avf x) ))
                 (equal (hot (next p x))
                     (hot x) ) ) ) ) )
 ((avf (lambda (x) 0))
  (hot (lambda (x) 0))
  (plist (lambda () '(0)))
  (next (lambda (p x) x)) ) )
```

Here `plist` represents L and `next` is an abstraction of the transition relation. We may assume that `next` is deterministic, for all nondeterminacy can be hidden in hidden variables of the state `x`.

In order to follow a schedule until $avf = 0$ is reached, we define function `next*` by

```
(next* s x) =
    (cond ((nlistp s) x)
          ((zerop (avf x)) x)
          ( t (next* (cdr s) (next (car s) x)) ) )
```

We construct a function `subset*` such that `(subset* k L s)` expresses that schedule `s` is `k`-fair for `L`. We then prove that $avf > 0$ leads to $avf = 0$:

```
(lemma next*terminates (rewrite)
    (implies (and (not (lessp n (avf x)))
                  (subset* n (plist) s) )
             (zerop (avf (next* s x))) ) )
```

We relate this result to the goal Q by first postulating an axiom that represents (ii):

```
(constrain goal-axiom (rewrite)
    (implies (and (not (equal (avf x) 0))
                  (equal (avf (next p x)) 0) )
             (goal (next p x)) )
    ((goal (lambda (x) t))) )
```

and then proving

```
(lemma next*terminates-at-goal (rewrite)
    (implies (and (not (lessp n (avf x)))
                  (not (equal (avf x) 0))
                  (subset* n (plist) s) )
             (goal (next* s x)) ) )
```

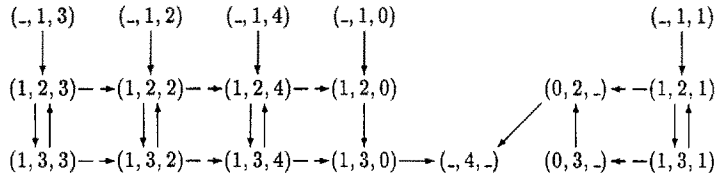
The relation with the initial condition via (i) is so obvious that we need not treat it in the abstract theory.

Remarks. In the theorem we do not introduce an invariant, since we only need to argue about states with $avf > 0$. In applications we will use invariants, but we then define avf in such a way that $avf > 0$ implies the invariants.

3.3. Progress for Peterson's Algorithm

We claim that $pc.q = 1$ leads to *crit.q* under bounded fairness. In order to prove this we construct state functions avf and *hot*, so as to apply the theorem of the previous section.

We take $q = 0$ and analyse the possible states and their transitions with regard to the question how process 0 proceeds to *crit*. Since the number of states is small, it would be easy to use a model checker for this purpose. Actually, we do it by hand. Because of invariant (Jq1), the reachable states are characterized by the value of *you* and the two values of *pc*. So the state is characterized by the triple (*you*, *pc*.0, *pc*.1). We thus get the diagram below.



The aim of process 0 is to reach state $(\_, 4, \_)$, where *crit*.0 holds. Here, "_" indicates a *don't care* component. The solid arrows in the diagram represent transitions that process 0 can do. The broken arrows represent transitions by process 1, that are needed for process 0 to reach *crit*. It is clear from the diagram that process q cannot further approximate *crit* when it satisfies

$$blocked.q: pc.q \in \{2, 3\} \wedge you \neq q \wedge pc.you \neq 0$$

We therefore define

$$hot.q = \textbf{if } blocked.q \textbf{ then } 1-q \textbf{ else } q \textbf{ fi}$$

In order to find a variant function *avf* we proceed as follows. We extend the diagram with all actions by process 1. We then assign values *vf* to the states, starting with $vf = 0$ at the aim $(\_, 4, \_)$, in such a way that condition (iv) of the theorem is satisfied. For example, $vf = 1$ at $(0, 2, \_)$, $vf = 2$ at $(0, 3, \_)$, etc. When all states have a value for *vf*, we invent some recipe that expresses *vf* as a function. In this way, we arrive at the definition

$$\begin{aligned} vf.q = \ & \textbf{if } pc.q = 1 \textbf{ then } 9 \\ & \textbf{elsif } pc.q \in \{0,4\} \textbf{ then } 0 \\ & \textbf{elsif } you = q \textbf{ then } pc.q - 1 \\ & \textbf{else case } pc.(other.q) \textbf{ of} \\ & \quad 0{:}7 - pc.q; \\ & \quad 1{:}3; 2{:}7; 3{:}8; 4{:}6 \\ & \textbf{case fi.} \end{aligned}$$

Now we let Nqthm verify that, if $vf.q > 0$ and (Jq*) holds, no process can increment *vf.q* and each action of *hot.q* decrements *vf.q*. Moreover, *hot.q* does not change unless *vf.q* decreases, and *crit.q* is established when $vf.q = 0$ is reached. It follows that the theorem can be used with for *avf* the function *vfc.q* given by

$$vfc.q = \textbf{if } (\text{Jq*}) \textbf{ then } vf.q \textbf{ else } 0 \textbf{ fi}$$

This shows that indeed $pc.q = 1$ leads to *crit.q* under bounded fairness.

The final fairness result of the mechanical proof is

```
(lemma wish-leads-to-crit (rewrite)
    (implies (and (equal (pc q x) 1)
                  (member q (bit))
                  (jq* x)
                  (subset* 9 (bit) s) )
             (crit q (step* q s x)) ) )
```

Here `(bit)` is the list of processes `'(0 1)` and `(step* q s x)` is the state reached from state `x`, when the algorithm executes according to schedule `s`, and not longer than needed for process `q` to reach `crit`. So this proves that, if some process $q \in \{0,1\}$ has $pc.q = 1$ and the invariant holds, every execution under a 9-fair schedule leads to *crit.q*.

We do not know other formal or mechanical proofs of progress for Peterson's algorithm. It is likely that Russinoff's approach [Rus92] can be used to give a proof based on the classical concept of weak fairness. Yet it may be easier to give an Nqthm proof that bounded fairness implies weak fairness.

4. Concluding Remarks

Although Nqthm is a first-order theorem prover, its facility to instantiate results of an axiomatic theory allows the development of generic theories for the treatment of imperative programs. We use Nqthm's facility to interpret quotations of terms to express algorithms in a concise and formal way. In this way Nqthm's lack of higher order functions is somewhat compensated.

In both examples treated we formulated a final result with explicit bounds. Nqthm does not provide existential quantification to abstract from such bounds. See also the discussion in [Moo96] Section 6.3.

For Peterson's mutual exclusion algorithm, we have proved progress under bounded fairness, a notion slightly stronger than weak fairness.

From our experience [Hes95, Hes97], we can say that the methods proposed are applicable to bigger programs, in particular to bigger distributed programs. We fear that the lack of higher-order abstractions may become inconvenient for specifications in large modularized software.

Acknowledgement

Remarks and questions of the referees have led to some important improvements.

References

[ApO91]* Apt, K. R. and Olderog, E.-R.: Verification of Sequential and Concurrent Programs. Springer, 1991.

[BaW90] Back, R. J. R. and von Wright, J.: Refinement concepts formalized in higher order logic. *Formal Aspects of Computing* **2** (1990) 247–272.

[BoM88] Boyer, R. S. and Moore, J. S.: A Computational Logic Handbook. Academic Press, Boston etc., 1988.

[BoM92] Boyer, R. S. and Moore, J. S.: A Computational Logic Handbook, Authorized Excepts from a Proposed Second Edition, to be obtained by ftp from Computational Logic Inc. Information available at `nqthm-request@cli.com`.

[ChM88] Chandy, K. M. and Misra, J.: Parallel Program Design, A Foundation. Addison–Wesley, 1988.

[Dij76] Dijkstra, E. W.: A discipline of programming. Prentice–Hall 1976.

[EGL92] Engberg, U., Grønning, P. and Lamport, L.: Mechanical verification of concurrent systems with TLA. In *Computer Aided Verification*. Springer Verlag 1992, LNCS 663.

[GaG88] Garland, S. J. and Guttag, J. V.: LP: the Larch Prover. In E. Lusk and R. Overbeek, (eds.): 9th Conference on automated deduction (CADE). Springer 1988, LNCS 310, pp. 748–749.

[Gor89] Gordon, M. J. C.: Mechanizing programming logics in higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, (eds.): *Current trends in hardware verification and theorem proving*. Springer 1989, pp. 387–439.

[GoM93] Gordon, M. J. C. and Melham, T. F. (eds.): Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press, Cambridge, UK, 1993.

[Gri81] Gries, D.: The science of programming. Springer 1981.

[Hes95] Hesselink, W. H.: Wait-free linearization with a mechanical proof. *Distributed Computing* **9** (1995) 21–36.

[Hes96] Hesselink, W. H.: The verified incremental design of a distributed spanning tree algorithm – extended abstract. See [Hes@].

[Hes98] Hesselink, W. H.: The design of a linearization of a concurrent data object. To appear in Proceedings Procomet '98. See [Hes@].

[Hes@] Hesselink, W. H.: Web site: `http://www.cs.rug.nl/~wim`.

[Hoo94] Hooman, J.: Correctness of real time systems by construction. In: H. Langmaack, W.-P. de Roever and J. Vytopil, (eds.): *Formal Techniques in real-time and fault-tolerant Systems*. Springer 1994. LNCS 863. pp. 19–40.

[Kal90] Kaldewaij, A.: Programming: the Derivation of Algorithms. Prentice Hall International, 1990.

[Lam94] Lamport, L.: The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* **16** (1994) 872–923.

[Moo94] Moore, J. S.: A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. *Formal Aspects of Computing* **6** (1994) 60–91.

[Moo96] Moore, J. S.: Piton: A mechanically verified assembly–level language. Kluwer, 1996.

[OSR93] Owre, S., Shankar, N. and Rushby, J. M.: User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release). CSL, SRI International, Menlo Park, CA, February 1993 (three volumes).

[Pet81] Peterson, G. L.: Myths about the mutual exclusion problem. *IPL* **12** (1981) 115–116.

[Rus92] Russinoff, D. M.: A verification system for concurrent programs. *Formal Aspects of Computing* **4** (1992) 597–611.

[SkS93] Skakkebæk, J. U. and Shankar, A.: A duration calculus proof checker: Using PVS as a semantic framework. SRI, CSL Tech. Report, December 1993.

[Tel94] Tel, G.: Distributed Algorithms. Cambridge University Press, 1994.

Received June 1997
Accepted in revised form October 1997 by E. C. R. Hehner