SPECIAL ISSUE

7th SA COMPUTER RESEARCH SYMPOSIUM
When the first SA Computer Symposium was held at the CSIR in the early eighties, it was unique. There was no other forum at the time for the presentation of research in computer science. In the intervening decade, conferences, symposia and workshops have sprung up in response to demand, and now there are several successful ventures, some into their third or fourth iteration. Each of these addresses a specific topic - for example, hypermedia, expert systems, parallel processing or formal aspects of computing - and attracts a specialised audience, well versed in the subject and eager to learn more. For the main part, the proceedings are informal, and certainly not archival.

SACRS, though, is still unique, in that it deliberately covers a broad spectrum of research in computing, and in addition, seeks to provide a lasting record of the proceedings. To achieve the second aim, we negotiated with the SA Institute of Computer Scientists for the proceedings to form a special issue of the SA Computer Journal, and the copy you have in front of you is the result. The collaboration between the symposium committee and the journal’s editorial board placed high standards on the refereeing and final presentation of the papers, to the symposium’s benefit, while we were still able to maintain a fresh, audience-oriented approach to the selection of papers.

This is SACJ's first such special issue, and the largest issue (at 145 pages) to date. We hope that it is only the beginning of future such collaborations.

In all 29 papers were received, all were refereed twice, and 19 were chosen for presentation by the programme committee. All the papers were thoroughly revised by the authors on the basis of the referee’s comments, and the committee’s suggestions aimed at making the material more accessible to a broadly-based audience. Papers had to be new, and not to have been presented elsewhere, a requirement that is still unusual within the SA conference round.

A third goal of SACRS has been to invite keynote speakers, usually from overseas. This year, we are fortunate to present Dr Vinton Cerf, the father of the Internet and a world-renown expert on computer networks. Although his paper is not available for this special issue, it will appear later in SACJ. Through the good offices of Professor Chris Brink of UCT, we also have three other speakers from Germany, Canada and the US adding interest to the event, and two of their papers appear in this issue.

The programme committee originally devised a theme for the symposium - “Computing in the New South Africa”. We received several queries as to the meaning of this theme, but unfortunately few papers that addressed it directly. One prospective author went as far as to enquire whether computer research would survive in the new South Africa. Another felt that his work was definitely not in the theme, as it was genuine, old world, basic, theoretical science! Nevertheless, there are two papers that consider one of South Africa’s key issues, that of language. Others look at the success we have achieved in applying technology to mining, and the future of low-cost operating systems. In all, the mix of papers represents a balance between the theoretical and the practical, the past and the future, all firmly based in the computing of the present.

Organising the symposium has involved the hard work of several people, and I would like to thank in particular

• Derrick Kourie, my co-organiser, and the editor of SACJ for his invaluable advice and hard work throughout the planning and implementation stages;
• Riel Smit, the production editor, for attaining such a high standard in such a short time for so many papers;
• Gerrit Prinsloo and the staff at the CSSA for their efficient and quite delightfully unfussy organisation;
• Persetel for their very generous sponsorship of R25000, and Tim Schumann for taking a genuine interest in our events;
• the Foundation for Research Development for sponsoring Vint Cerf’s visit;
• and finally the Department of Computer Science of the University of Pretoria for providing the ideal working conditions for undertaking ventures of this kind, and especially Roelf van den Heever for his unfailing encouragement and support.

Judy M. Bishop
Organising Chairman, SACRS 1992
Guest Editor, SACJ Special Issue
Referees

The journal draws on a wide range of referees. The following were involved in the refereeing of the papers selected for this special issue. Their role in certifying the papers and their contribution to enhancing the quality of papers is sincerely appreciated.

John Barrow
Ronnie Becker
Danie Behr
Sonia Berman
Liesbeth Botha
Theo Bothma
Chris Brink
Peter Clayton
Ian Cloete
Antony Cooper
Elise Ehlers
Quintin Gee
Andy Gravell
Wendy Hall
David Harel
Scott Hazelhurst
Derrick Kourie
Willem Labuschagne
Doug Laing
Dave Levy
Graham McLeod
Hans Messerschmidt
Deon Oosthuizen
Ben Oosthuysen
Neil Pendock
D Petkov
Martin Rennhackkamp
Cees Roon
Jan Roos
John Shochot
Morris Sloman
Riel Smit
Pat Terry
Walter van den Heever
Lynn van der Vegte
Herman Venter
Herna Viktor

UNISA
University of Cape Town
University of Pretoria
University of Cape Town
University of Pretoria
University of Pretoria
University of Cape Town
Rhodes University
Stellenbosch University
CSIR
Rand Afrikaanse University
DALSIG
University of Southampton
University of Southampton
The Weizmann Institute of Science
University of the Witwatersrand
University of Pretoria
UNISA
ISM
University of Natal
University of Cape Town
University of the Orange Free State
University of Pretoria
University of Pretoria
University of the Witwatersrand
University of Natal
Stellenbosch University
University of Pretoria
University of Pretoria
University of the Witwatersrand
Imperial College, London
University of Cape Town
Rhodes University
University of Pretoria
University of Pretoria
University of Fort Hare
Stellenbosch University
Mechanizing Execution Sequence Semantics in HOL

Gavan Tredoux
Laboratory for Formal Aspects and Complexity in Computer Science, Department of Mathematics, University of Cape Town.

Abstract

The mechanization in Higher Order Logic of a general-purpose operational semantics for programming languages is described. The mechanization allows the sound derivation of Dijkstra-style axiomatic semantics. A small programming language fragment is presented as an illustration

Keywords: HOL, Mechanized Proof, Program semantics, Weakest preconditions
Computing Review Categories: D.2.4, D.3.1, F.3.1, F.3.2, F.4.1

Received March 1992, Accepted April 1992

1 Introduction

This paper describes a mechanization, in the interactive theorem prover HOL, of a general-purpose semantics for programming languages, based on execution sequences. This provides an environment in which a wide class of formalisms for reasoning about programs can be mechanized, including Dijkstra's algebra of weakest preconditions, dynamic logic and Hoare-style specifications. The aim is to show that execution sequences provide a mathematically 'clean' semantics for programming languages that is straightforward to mechanize in HOL, and properly models both bounded and unbounded non-determinism. This is in contrast to other semantics for non-determinism, for instance powerdomain constructions, whose mathematical sophistication makes mechanization in HOL more difficult.

2 An Overview of HOL

HOL is a proof assistant and verifier, descended from the LCF family of theorem provers [11, 17], embedding a higher-order logic in a functional programming language, ML [9]. The system guarantees adherence to the logic by isolating its axioms and primitive rules of inference in an abstract data type, thm. The strongly typed nature of ML ensures that theorems (objects of type thm) can only be obtained from the axioms and primitive rules of inference. The embedding in ML also allows an arbitrary degree of mechanization, in the form of derived rules of inference and proof strategies, while still guaranteeing soundness. This is in contrast to non-inferential approaches to theorem proving, which often deliver 'theorems' after complicated rewriting and resolution procedures without guaranteeing the soundness of those procedures. To date, HOL has been used mostly for the specification and verification of hardware [6], but the expressive power of higher-order logic captures, in principle, anything that is mathematically tractable [5, 10].

The HOL logic is based on Church's Simple Theory of Types [7, 1], and is essentially Predicate Calculus enriched with typed lambda calculus. Variables may range over functions, and the arguments of functions may themselves be functions. Each term of the logic must be well-typed, having a type consistent with the types of its sub-terms. HOL has a number of built in types, for example boolean values bool, natural numbers num, and strings string. There are also type constructors, which construct new types from existing ones (for example function spaces between types, lists of elements of a type) and facilities for adding new types to the logic. Denote a typed term by t:s where t is a term and s is its type. For example, denote a variable x of type bool by x:bool. Most of the constructs of the predicate calculus are implemented straightforwardly in HOL, but some unusual features of the logic worth noting are:

- Predicates are functions with range bool. For example \( \lambda n: \text{num}. n < 5 \) is a predicate on the natural numbers, mapping any natural number to T (for true) or F (for false), depending on whether it is less than 5 or not. A set of values of some type may be represented by its membership predicate on that type.
- Hilbert's \( \varepsilon \)-operator plays an important role. If P is a predicate over values of type \( \sigma \), then \( \varepsilon x: \sigma. P x \) is an unspecified but fixed variable x of type \( \sigma \), such that if anything satisfies the predicate P, then x satisfies P - if nothing satisfies P then \( \varepsilon x: \sigma. P x \) is an unspecified but fixed value of type \( \sigma \). More formally, it is an axiom of the logic that if \( \vdash \exists x. P x \), then \( \vdash P(\varepsilon x. P x) \) [12, 1].
- There is notation for conditional statements. 'If c then d else e' is denoted by 'c \Rightarrow d|e'.
- The type discipline is polymorphic, so that terms with variable types are allowed [14].

Proof in HOL is by natural deduction; theorems are obtained by rules of inference from axioms and previously proved theorems, in the 'forward' direction. HOL also supports simulation of 'backward' goal-directed proof, through the use of Milner's tactics [15], which automatically translate goal-directed 'proofs' into forward proofs [16].

The HOL logic can be extended by postulating new
axioms, which risks the introduction of inconsistency, or by definition, which is conservative in that it cannot introduce inconsistency [13]. There are three mechanisms for making new definitions:

- **Type definition.** A new type can be introduced as a non-empty named subset of an existing type, by providing an isomorphism from the new type to the subset, and proving that the subset is non-empty. New type operators may also be defined, and there are facilities for defining a restricted class of recursive types [13].

- **Constant definition.** A new constant can be introduced as a non-recursive abbreviation for a term in the logic.

- **Constant specification.** If it can be proved that \( \vdash \exists x.Px \), then a constant \( c \) can be introduced satisfying \( \vdash Pc \). This allows partial specification, where \( P \) fixes only certain properties of the constant and not others; for example, a total function could be introduced by specifying its behaviour on a subset of its domain and leaving it unspecified elsewhere.

The extensive use of definitions entails that proofs rely heavily on rewriting routines; these are implemented as derived rules of inference in ML procedures, guaranteeing soundness. ML is itself a higher-order functional language (functions may take functions as arguments) so that rewriting routines may be soundly tailored by the user to specific situations, in arbitrarily complex ways. The soundness of user extensions to inference mechanisms poses a difficult problem for most theorem-provers not in the LCF fold [3].

Extensions of the logic are organized hierarchically in theories; named collections of types, type operators, constants, definitions, axioms and theorems. A library of commonly used theories is built-in, for example theories concerning sets, the natural numbers and finite-length lists. The expressive power of the HOL logic, due to its higher-order variables and type polymorphism, allows the formulation and proof of very general definitions and lemmas, which can be reused in many different contexts. The extent to which this simplifies mechanization will be revealed later in this paper.

3 Execution Sequence Semantics

Only an overview of execution sequence semantics can be given here; for a more detailed treatment see [4], where a similar semantics is used to model Dijkstra's algebra of weakest preconditions [8]. Execution sequences (exseqs) are sequences of machine states, where the states are left undefined, of one of the following kinds:

- **Finite and properly terminated.** Denoted by \( < s_1, s_2, s_3, \ldots, s_n > \), where the \( s_i \) are states.
- **Finite and improperly terminated.** Denoted by \( < s_1, s_2, s_3, \ldots, s_n > \).
- **Infinite, non-terminating.** Denoted by \( < s_1, s_2, s_3, \ldots, s_n, \ldots > \).

Each exseq thus represents a computation path, starting at the first state in the sequence and proceeding through the remaining states. Aborted computation is distinguished from non-termination as improper termination. Denote an arbitrary exseq by \( < s_1, s_2, s_3, \ldots, s_n, \ldots > \), which may be of any of the above kinds. The following operations on exseqs are essential for defining the semantics of programs (note that the operations are only defined for the kinds of exseqs indicated):

- **first** \( < s_1, s_2, s_3, \ldots, s_n > = s_1 \).
- **last** \( < s_1, s_2, s_3, \ldots, s_n > = s_n \).
- **length** \( < s_1, s_2, s_3, \ldots, s_n > = n \).
- **length** \( < s_1, s_2, s_3, \ldots, s_n > = n \).
- **<** \( < s_1, s_2, s_3, \ldots, s_n > = < s_1, s_2, s_3, \ldots, s_n, f_n, \ldots > > = < s_1, s_2, s_3, \ldots, s_n, f_n, \ldots > > = t_n \).

Note that the resulting exseq has the same subtype as the second exseq, that the intermediate state is not repeated, and that the join is undefined where \( s_n \neq t_1 \).

Programs are identified with sets of exseqs, with each exseq a possible computation path. This models non-determinism by allowing a program to contain exseqs starting from the same state; the program may follow any of its possible computation paths starting from such a state, non-deterministically. Programs may be said to be deterministic if they have no more than one exseq starting in any state. As programs are sets, they may be combined using the usual set-theoretic operations, for example union and intersection, and are completely specified by their elements. Let \( S \) be the set of all machine states, and then define the following fragment of a programming language:

- **skip** \( = \{ < s, s > | s \in S \} \), a program which does nothing and then terminates.
- **P ;; Q** \( = \{ e \in P \land deQ \text{ and the join is defined} \} \cup \{ e \in deP \text{ and does not terminate properly} \} \), sequential composition of two programs.
- **P || Q** \( = P \cup Q \), non-deterministic choice between programs, modelled by set-theoretic union.

In the model chosen here, every program computation does something so that skip is treated as an identity mapping between states, rather than a program which does nothing, for instance one which contains only singleton exseqs. This is a next-state relation: intermediate states are significant. Models in which intermediate states are not significant use an input-output relation, for example Dijkstra's algebra of weakest preconditions. Hence skip is not an identity with respect to sequential composition, as the first example below shows:

- **skip ;; skip** \( = \{ < s, s > | s \in S \} \)
- **(skip ;; skip)** \( = \{ < s, s + > | s \in S \} \cup \{ < s, s > | s \in S \} \).

This is because skip is treated as a program which does not alter a state, but nevertheless performs a computation. Of course, viewed in terms of the input-output relation, skip alters neither input nor output, and hence 'does nothing.'

Note that the use of union to model non-deterministic choice diverges from [4], who use Hilbert's ε-operator, with some unfortunate consequences. Defining for programs \( P \) and \( Q \) \( P || Q = \varepsilon x : (x = P) \lor (x = Q) \), entails that:

- If \( P \) and \( Q \) are deterministic then \( P || Q \) is deterministic, so that no new non-determinism is introduced.

82 SACJ/SART, No 7, 1992
Which program $\varepsilon$ chooses is fixed and unspecified, but it is either $P$ or $Q$, and so it must be deterministic.

- $(P \parallel Q) :: (P \parallel Q)$ is either the program $P :: P$ or the program $Q :: Q$, since what $\varepsilon$ chooses is fixed. This excludes the possibilities $P :: Q$ and $Q :: P$, which is most undesirable.

The use of union to model non-deterministic choice can include sequences and some of their (initial) subsequences in the same program, as the example $(\text{skip} || (\text{skip} :: \text{skip}))$ demonstrates; contrast this with [4], where $\varepsilon$ was used specifically to exclude subsequences. This extends non-determinism to include termination, for it is not determined whether in a state that is the last of a subsequence, the program will terminate, i.e. it is on the computation path represented by the subsequence, or proceed to compute, i.e. it is on the computation path represented by the supersequence. This notion of non-determinism was raised but not pursued in [4].

### 4 Representing Execution Sequences

Execution sequences are represented in the HOL logic, forming an extension of that logic; they are defined in terms of constructs of the logic already present, using the type definition and constant definition facilities outlined above. It is worth emphasizing that this is a conservative extension of the logic - soundness is guaranteed - and that an axiomatic definition of execution sequences, while it would appear to be more natural and would entail less work, would carry the danger of introducing inconsistency.

To represent execution sequences, a more concrete notion of 'machine state' is needed; here states are simply thought of as mappings from variables (identified by character strings) to their values (which are just natural numbers). This may be defined in HOL as the following type abbreviation:

```haskell
let state = define_type 'state' 
let val (w :: state) = define_type 'w :: state' 
let INCP = new_recursive_definition false exseq 'INCP' 
let INF! = new_recursive_definition false exseq 'INF!' 
let INCP! = new_recursive_definition false exseq 'INCP!' 
let val = new_recursive_definition false w :: state 'val' 
```

The type of execution sequences is defined in HOL as a disjoint union of sub-types of finite and infinite sequences, where an infinite sequence of states is represented by a mapping from numbers to states $f :: \text{num} \rightarrow \text{state}$ indexing the elements, and a finite sequence by a tuple $(f, n)$, where $f :: \text{num} \rightarrow \text{state}$ indexes the elements, and $n$ represents the length. To ensure that the finite sequences are non-empty, and that each exseq is uniquely represented by a tuple $(f, n)$, we require that $0 < n$, and that $f(i)$ be an arbitrary but fixed constant where $n \leq i$, so that values of $f$ which do not represent elements represent nothing. Thus the type of finite sequences is defined as that subset of the cartesian product $(f :: \text{num} \rightarrow \text{state}) \otimes (\text{num})$ satisfying the above conditions on $f$ and $n$.\footnote{This is a simplification: a type of finite sequences of elements of any type $\tau$, $(\tau ::\text{finseq})$, is defined, and a finite sequence of states is instantiated as $(\text{state} :: \text{finseq})$ — but this is not to our purpose here.}

The facilities in HOL for defining new types allow 'free type definitions' using a BNF-style syntax, so that existing types may be easily combined in disjoint unions. These are used to combine finite and infinite sequences to form exseqs, as the following HOL type definition shows:

```haskell
let exseq = define_type 'exseq' 
  'exseq = FIN (state)finseq 
  | INF (num -> state) 
  | INC (state)finseq';
```

Thus, an exseq is either a finite sequence of states labelled FIN, an infinite sequence labelled INF, or a finite sequence labelled INC. These labellings correspond to properly terminated exseqs, non-terminating exseqs, and improperly terminated exseqs, respectively. To facilitate reasoning about the lengths of sequences, the type of natural numbers $\text{num}$ is extended to include $w$, forming the new type, $w :: \text{nat}$, and a partial order on $w :: \text{nat}$, denoted by '<!', is defined in the obvious way:

```haskell
let w :: nat = define_type 'w :: nat' 
  'w :: nat = OMEGA | NAT num';
```

Since exseqs have been divided into different subtypes, there must be a way of explicitly determining the subtype of any exseq; these tests will play an important role in the examples that follow. We will also need a way of getting the 'n' out of 'NAT n', if we want to add and subtract lengths of finite sequences. The following functions perform these operations, and are defined in HOL by cases, using the structure of the types exseq and $w :: \text{nat}$. Here $\text{TERM}$, $\text{INFI}$ and $\text{INCP}$ are boolean functions which return $\text{T}$ if their arguments are finite and properly terminated, finite but not properly terminated, or infinite (non-terminating), respectively. The function $\text{val}$ will strip the NAT label off an element of $w :: \text{nat}$ to return a natural number, and is undefined for $\text{OMEGA}$.

```haskell
let TERM = new_recursive_definition false exseq 'TERM' 
  'TERM (FIN f :: exseq) = T' 
  'TERM (INF f :: exseq) = F' 
  'TERM (INC f :: exseq) = F*';
let INFI = new_recursive_definition false exseq 'INFI' 
  'INFI (FIN f :: exseq) = F' 
  'INFI (INF f :: exseq) = T' 
  'INFI (INC f :: exseq) = T*';
let INCP! = new_recursive_definition false exseq 'INCP!' 
  'INCP! (FIN f :: exseq) = F' 
  'INCP! (INF f :: exseq) = F' 
  'INCP! (INC f :: exseq) = T*';
let val = new_recursive_definition false w :: nat 'val' 
  'val (NAT n) = n*';
```

Exseqs satisfy some fundamental properties which provide an abstract characterization, or specification, of them, and which must be derived from any representation to establish that they have been correctly represented. For instance, any exseq is completely determined by its elements, length and subtype (FIN, INF or INC); this is captured by the following HOL theorem, derived directly from the definitions involved:

($\forall$ x y. x sametype exseq y $\Rightarrow$

(length x = length y) $\land$

($\forall$ n. (NAT n) $<$ '! (length x) $\Rightarrow$ (element n x = element n y)))

SACJ/SART, No 7, 1992 83
Thus, exseqs of the same subtype with the same length and the same elements are equal, and therefore the representation of exseqs is well-defined. A mechanism for explicitly constructing exseqs containing particular states containing just the states of the constructions, where tics, only singletons and pairs need to be explicitly constructed. The following theorems establish the correctness of the constructions, where $sng(s)$ returns a singleton exseq containing just the state $s$, and $pair(s, t)$ returns an exseq containing just the states $s$ and $t$:

1. $\forall s:state. \; length(sng\; s) = NAT\; 1$
2. $\forall s:state. \; first(sng\; s) = s$
3. $\forall s:state. \; last(sng\; s) = s$
4. $\forall t:state. \; length(pair(s, t)) = NAT\; 2$
5. $\forall t:state. \; first(pair(s, t)) = s$
6. $\forall t:state. \; last(pair(s, t)) = t$

The following theorems establish the correctness of the representation of the join function, by describing the length of the join and the elements it contains.

1. $\forall d:exseq. \; (first\; d = last\; e) \implies TERM\; e \implies\; \forall n:num. \; ((NAT\; n < 1\; (length\; e)) \implies (element\; n\; e \cup join\; d) = element\; n\; e) \land ((NAT\; n < 1\; length(e \cup join\; d)) \implies (element\; n\; e \cup join\; d) = element((1+n-\; (val(length\; e)))\; d)))$

2. $\forall d:exseq. \; (first\; d = last\; e) \implies TERM\; e \implies\; (IFI\; d \Rightarrow (length(e \cup join\; d) = OMEGA) \lor (val(length(e \cup join\; d)) = val(length\; e) + val(length\; d) - 1))$

Thus $e \cup join\; d$, if it is defined (ie. $e$ terminates cleanly and the last of $e$ is the first of $d$) has all the elements of $e$ and all the elements of $d$ except its first, in that order. 2

3. $length(e \cup join\; d) = OMEGA$
   if $d$ is infinite, non-terminating
   $val(length(e \cup join\; d)) = val(length\; e) + val(length\; d) - 1$
   if $d$ is finite

Since exseqs are completely characterized by their elements and length, the above theorems alone establish the correctness of the representations of the functions involved.

## 5 Representing Programs

As a program is a set of execution sequences, and sets may be identified with their membership predicates, each program can be represented as a predicate on the type of execution sequences. This allows partial programs, which might not have any execution sequences starting from some states. As programs are predicates, they may be combined using the usual logical operators, and are easy to manipulate in HOL. Moreover, specification languages like Dijkstra's calculus of weakest preconditions and Hoare-style assertions, can now be expressed very naturally in the logic. As specifications employ the notion of predicates on states, the following type abbreviations are useful:

```
new_type_abbrev('command', ':exseq -> bool');
new_type_abbrev('pred', ':state -> bool');
```

Thus commands are mappings from exseq to bool, while predicates on states are mappings from state to bool. Since we are allowing partial commands, to model partial programs, determining the domain of a command will be useful. Moreover, a test for determinism, or lack of it, is essential. The following function constants are defined accordingly:

1. $\forall c:command. \; \forall s:state. \; dom\; c = \lambda s:state. \; ve:exseq. \; c\; e \wedge (first\; e = s)$
2. $\forall c:command. \; det\; c = \lambda e\; exseq. \; (c\; e) \implies (first\; e = first\; e_1) \implies (e_1 = e_2)$

The HOL definitions for the programming language fragment discussed above now follow naturally from their set-theoretic counterparts, where $';;'\;$ sequences commands, and $'\lor'$ chooses between two commands, non-deterministically:

```
skip = \lambda s:state. \; p = pair(s, s)
seq = \lambda c_1, c_2:command. \; c_1 \lor c_2 = \lambda e:exseq. \; ((c_1\; e) \wedge \neg TERM\; e) V (\exists e_2:exseq.\; (c_1\; e_1) \wedge (TERM\; e_1) \land (c_2\; e_2) \land (first\; e_2 = last\; e_1) \land (e = e_1 \cup join\; e_2))
```

Programming constructs have thus been directly represented in the HOL logic; programs can be translated into terms of the logic, where each program is of the type command. This facilitates proving properties of particular programs, for example correctness with respect to weakest precondition specifications, but has the drawback that there is no way of formulating theorems about a particular language as a whole. Theorems may quantify over particular commands only, or over commands as a whole. For example, there is no way of asserting that every program in the fragment introduced above is total, although this is certainly true. This can be remedied by defining a recursive type that represents the language as a syntactic entity, and then recursively associating constructs of the language with terms in the logic of type command, via a 'Meaning' function in classical denotational style. The HOL code below does this for the programming fragment:
Dijkstra's weakest precondition operator [8] has become central to many program specification and transformation schemes, and embedding this in a theorem proving environment allows proof of the soundness of specifications and the correctness of transformations [2]. The definition in HOL is straightforward:

\[ \text{wp} = \forall_{\text{def}} \forall_{\text{command}}. \forall_{\text{pred}}. \forall_{\text{state}}. \forall_{\text{exseq}}. \]

\[
\begin{align*}
\text{wp} c R & = \lambda s: \text{state}. \forall e: \text{exseq}. \\
(c e) & \implies (\text{TERM} e \land (R (\text{last} e)))
\end{align*}
\]

Thus for any command \( c \), \( \text{wp} c \) is a mapping from \( \text{pred} \) to \( \text{wp} \), i.e. a predicate transformer. For any predicate on states \( R \), \( \text{wp} c R \) is a predicate that is satisfied by those states from which, if \( c \) is defined at those states, \( c \) is bound to terminate in a state satisfying \( R \). The following definitions facilitate manipulation of predicates on states:

\[
\begin{align*}
\text{false} & = \forall_{\text{def}} \text{false} = \lambda s: \text{state}. F \\
\text{true} & = \forall_{\text{def}} \text{true} = \lambda s: \text{state}. T \\
\text{OR} & = \forall_{\text{def}} \forall_{\text{Q}}. Q: \text{pred}. \\
\text{AND} & = \forall_{\text{def}} \forall_{\text{Q}}. Q: \text{pred}. \\
\text{IMPLIES} & = \forall_{\text{def}} \forall_{\text{Q}}. Q: \text{pred}. \\
(P s) & \implies (Q s)
\end{align*}
\]

Dijkstra formulates a number of 'Healthiness Conditions', identities involving the weakest precondition operator which any program ought to satisfy, provided that the weakest precondition operator and the programming semantics have been correctly defined. These can be derived formally in HOL from the definitions given above.

\[
\begin{align*}
\forall_{\text{command}}. \\
\forall_{\text{command}}. \forall_{\text{R}}. R: \text{pred}. \\
\forall_{\text{command}}. \forall_{\text{Q}}. Q: \text{pred}. \\
((\text{wp} c Q ) \lor (\text{wp} c R)) \iff (\text{wp} c (Q \lor R))
\end{align*}
\]

The purpose of this paper has been to show that HOL provides a natural vehicle for mechanizing execution sequence semantics, with a view to mechanizing formalisms for specifying and proving properties of programs; thus the programming fragment above was intended to illustrate the principles involved. This is the subject of ongoing research by the author.

References


SACJ/SART, No 7, 1992
Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines:

- Use wide margins and 1 1/2 or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetical order of the (first) author's surname, and should be cited in the text in square brackets [1, 2, 3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) \LaTeX file(s), either on a diskette, or via e-mail/ftp – a \LaTeX style file is available from the production editor;
2. In camera-ready format – a detailed page specification is available from the production editor;
3. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

Further instructions on how to reduce page charges can be obtained from the production editor.

4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for \LaTeX or camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 3 and 4 above will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

## Contents

Forward .......................................................... i  
Referees ................................................................... ii

Machine Translation from African Languages to English  
DG Kourie, WJ vd Heever and GD Oosthuizen ..................... 1  
Automatically Linking Words and Concepts in an Afrikaans Dictionary  
PZ Theron and I Cloete ................................................... 9  
A Lattice-Theoretic Model for Relational Database Security  
A Melton and S Shenoi ..................................................... 15  
Network Partitions in Distributed Databases  
HL Viktor and MH Rennhackkamp ........................................... 22  
A Model for Object-Oriented Databases  
MM Brand and PT Wood .................................................... 27  
Quantifier Elimination in Second Order Predicate Logic  
D Gabbay and HJ Ohlbach ................................................... 35  
Animating Neural Network Training  
E van der Poel and I Cloete .................................................. 44  
HiLOG - a Higher Order Logic Programming Language  
RA Paterson-Jones and PT Wood ........................................... 53  
ESML - A Validation Language for Concurrent Systems  
PJA de Villiers and WC Visser .............................................. 59  
Semantic Constructs for a Persistent Programming Language  
SB Sparg and S Berman ..................................................... 65  
The Multiserver Station with Dynamic Concurrency Constraints  
CF Kriel and AE Krzesinski .................................................. 75  
Mechanizing Execution Sequence Semantics in HOL  
G Tredoux ................................................................... 81  
Statenets - an Alternative Modelling Mechanism for Performance Analysis  
L Lewis .................................................................... 87  
N Pendock ................................................................ 95  
Galileo: Experimenting with Graphical User Interfaces  
R Aptekar and JM Bishop ..................................................... 99  
Placing Processes in a Transputer-based Linda Programming Environment  
Accessing Subroutine Libraries on a Network  
PH Greenwood and PH Nash .............................................. 117  
A Multi-Tasking Operating System Above MS-DOS  
R Foss, GM Rehmet and RC Watkins .................................... 122  
Using Information Systems Methodology to Design an Instructional System  
BC O'Donovan ................................................................ 126  
Managing Methods Creatively  
G McLeod .................................................................. 131  
A General Building Block for Distributed System Management  
P Putter and JD Roos ......................................................... 141