South African Computer Journal
Number 2
May 1990

Suid-Afrikaanse Rekenaar-tydskrif
Nommer 2
Mei 1990

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the South African Computer Society and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Suid-Afrikaanse Rekenaarvereniging en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083

Assistant Editor: Information Systems
Professor Peter Lay
Department of Accounting
University of Cape Town
Rondebosch 7700

Editorial Board

Professor Gerhard Barth
Director: German AI Research Institute
Postfach 2080
D-6750 Kaiserslautern
West Germany

Professor Judy Bishop
Department of Electronics and Computer Science
University of Southampton
Southampton SO 5NH
United Kingdom

Professor Donald Cowan
Department of Computing and Communications
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

Professor Jürg Gutknecht
Institut für Computersysteme
ETH
CH-8092 Zürich
Switzerland

Professor Pieter Kritzinger
Department of Computer Science
University of Cape Town
Rondebosch 7700

Professor F H Lochovsky
Computer Systems Research Institute
University of Toronto
Sanford Fleming Building
10 King's College Road
Toronto, Ontario M5S 1A4
Canada

Professor Stephen R Schach
Computer Science Department
Vanderbilt University
Box 70, Station B
Nashville, Tennessee 37235
USA

Professor Basie von Solms
Departement Rekenaarwetenskap
Rand Afrikaanse Universiteit
P.O. Box 524
Auckland Park 0010

Subscriptions

	Annual	Single copy
Southern Africa:	R32-00	R8-00
Elsewhere:	$32-00	$8-00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Editorial

Information Systems Research: A Teleological Approach?

The request to write this editorial came at a very opportune time, coinciding as it did with an intense examination of the development of the field of information systems and an analysis of the progress of IS research. I have therefore used this opportunity to focus my thoughts and outline some of my conclusions. By doing so I don't pretend to answer any questions, merely perhaps to stimulate thought amongst those SACJ readers involved in IS research.

The last fifteen years has seen a tremendous growth in the study of information systems. During this period a number of journals devoted to IS research appeared such as *MIS Quarterly, The Journal of MIS, Information and Management* and *Data Base.* There are now many research-based activities: the International Conference on Information Systems; the annual IS doctoral dissertation colloquium; and various awards for IS research contributions. Hundreds of universities world-wide have formed information systems departments with (reasonably) standard curricula.

Yet with all this, what has *really* been achieved from a research viewpoint? Are we any closer to understanding the true nature of information systems? Is there a general unified theory of information systems? Is there even an accepted, unique body of IS knowledge? The answer to all of these must surely be no.

We have, I believe, achieved precious little. Yes, we do understand something of IS development approaches. We understand a little more now than we used to about how users interact with systems. But to get back to the first question, do we really understand what information systems *are* and how they work? No. Which begs the question: Why not?

There are, again I believe, a number of reasons, but the foremost must be that the majority of people in the IS research community either reside in the business schools of the USA or are drawn from other disciplines. These people, it would appear, are researching for research's sake; to publish in order to secure tenure or develop a research track record, not to further the body of knowledge of the subject. There seems an almost frantic zeal to generate and test hypotheses, trying to adopt and pursue what is seen to be a "scientific approach". But there is very little focus - there can't be, or the answers to my questions earlier would be yes rather than no!

Let me hasten to add that there is nothing unique about these IS researchers. "Publish or perish" is still very much alive and well! But also they are really not all that different from other social scientists. As Nagel [3] observed:

> "... in no area of social enquiry has a body of general laws been established, comparable with outstanding theories in the natural sciences in scope of explanatory power or in capacity to yield precise and reliable predictions ..."

Why should this be the case? Is it because the great intellects gravitate to the natural sciences and the social sciences pick up the second best who are incapable of generating these general laws? I hope not! The answer may well be that we have become locked into a particular research approach which is inappropriate to developing a body of social science, and more particularly, IS knowledge. Maybe we should be learning from our own source discipline (systems theory) and be developing a real research approach which complements our field of study.

To explore this further let me go back to the roots of information systems. What is an information system? Do we really have an accepted definition? Probably the most widely referenced is that provided by Davis and Olson [2]:

> "an integrated, user-machine system for providing information to support operations, management and decision-making functions in an organization. The system utilizes computer hardware and software; manual procedures; models for analysis, planning, control and decision making; and a database".

Note how this emphasizes the man-machine interrelationship and underscores computers as a core component when they are not even necessarily a part of the information system. The worst aspect is that it does little to describe what a system is, and this may well be one of the causes of our research dilemma. Again, if we draw on systems theory then a more appropriate definition might well be: "a hierarchical set of procedures utilizing information to monitor and control organizational performance". Note that this definition fits with general systems theory that *all systems* have four basic foundations: cybernetics, hierarchy, control and information [1].

An additional aspect not apparently recognised by IS researchers is that the information system, just like any other system, biological or otherwise, suffers from the problem first identified by our own Jan Christiaan Smuts [4]: that of holism. Simply put, this says that the whole is greater than the sum of the parts. This means that information systems, unlike science, cannot be reduced to simple isolated fields of enquiry and then analyzed or tested using hypotheses and laboratory experiments from which elaborate generalizations may be inferred. They have levels of complexity with new factors emerging at each level. The problem with most of the current research is that it starts out with a reductionist approach and then focuses on the highest (or lowest) level. Thus the majority of the topics have as their target the interaction between user and computer or the management or application of technology. There is very little research that is taking place at fundamental level, that of developing a general theory of information sytems. This is the teleological approach, searching for the natural laws and developing the theory based on deduction and logical development. Until we can advance *that* area of knowledge and, from a basis of these fundamental laws, develop a hierarchy of hypotheses that can *then* be tested, we will have little focus to our IS research. It will remain a fragmented, uncohesive smattering of the work of individuals who are merely grasping at tenure. There are few people who would today argue against the inclusion of information systems as a field of study at a university or as a fruitful research area. But until such time as we focus on the foundation theory, it will remain unstructured and immature.

References

[1] P Checkland, [1984], *Systems Thinking, Systems Practice*, John Wiley and Sons, New York.

[2] Davis and Olson, [1985], *Management Information Systems: Conceptual Foundations, Structure and Development*, 2nd Ed: MacGraw-Hill, New York.

[3] E Nagel, [1962], *The Structure of Science*, Routledge and Hegan Paul, London.

[4] J C Smuts, [1929], *Holism and Evolution*, MacMillan, London.

Prof Peter Lay
Assistant Editor: Information Systems

This SACJ issue is sponsored by
Department of Computer Science
University of Cape Town

An Estelle Compiler for a Protocol Development Environment

P S Kritzinger and J van Dijk
Data Network Architectures Laboratory, University of Cape Town, Rondebosch, 7700

Abstract

The development of communication protocol standards has been accompanied by the standardization of formal specification languages to describe these standards in a clear and unambigious way. Simultaneously, techniques for the performance analysis of protocols and their validation have been proposed. This paper describes experience with the development of a compiler for one such specification language, Estelle. The compiler forms part of a comprehensive protocol development environment. Apart from a brief description of the functionality provided by this environment, aspects of Estelle which influenced the design of the compiler are described as well as salient aspects of the compiler itself.
Keywords: *Estelle, Compilers, Computer networks, Specification Languages.*
Computing Review Categories: *B.4.4, C.2.0, C.2.2, C.4, D.2.5, D.3.4*

Received May 1989, Accepted March 1990.

1 Introduction

There is currently a great deal of activity in developing international standards for the protocols and services of computer communication network software. Central to these standardization efforts is the concept of a Formal Description Technique (or FDT) required to produce clear and unambiguous descriptions of those standards. One such FDT is the language "Estelle" (Extended Finite State Language) [5] which uses an extended finite state machine to describe the behaviour of the protocol.

Alongside the development of formal description techniques, tools and techniques have developed for the validation [15], performance analysis [7] and testing [12] of software specified in Estelle.

Most of these techniques make use of the specification itself or a trace, generated by a simulated execution (or meta-implementation) of the specified communication software as well as auxiliary information, such as the set of states, generated by the FDT compiler.

It makes sense to integrate these various techniques and tools, with the compiler, into a single software environment to assist the developer of computer communication network software. Such a system called the Protocol Engineering Workbench (PEW) is described in this paper. A similar undertaking is the Estelle part of the SEDOS project [16] under the ESPRIT programme in Europe. The purpose of the paper is mainly to describe experience gained with the language and the development of the Estelle compiler.

2 The Estelle language

Before discussing our experience with the implementation of the compiler for Estelle it is worth highlighting aspects of the language which influenced the design of the compiler.

Estelle is based on an extended finite state machine model and a large subset of the Pascal programming language. The framework of an Estelle specification is a set of cooperating entities, each described as a *module*, interacting with each other by exchanging information through *channels*.

A module definition consists of two parts, namely a module header definition and one or more associated module body definitions as described in the following example.

```
module A systemprocess (n: integer);
  ip p: T(S) individual queue;
     p1: U(S) common queue;
     p2: W(K) common queue;
  export X,Y integer; Z:boolean
end;
```

The module header definition defines the module's interface with other modules in the specification. This interface includes interaction points, exported variables and module parameters. The interaction points define permissible interactions over a channel and present a named point of a bind with another interaction point. The exported variables define variables in an instance of a module which may be accessed and modified by the parent process. The parameters (when

specified) of a module allow values to be passed to instances of the module.

Together with a module header, a module body is used to define a module whose external visibility is defined by the header definition and behaviour defined by the module body definition. A body definition consists of three parts, namely a declaration-part, an initialization-part and a transition-declaration-part describing the transitions of the extended finite state machine (EFSM).

The declaration-part of a module body consists of constant, type, channel, (child) module header, (child) module body, internal interaction point, module variable, variable, state, state set, and procedure and function definitions. Constant, type, variable and procedure and function definitions are the same as for Pascal. The following is an example of a module body definition consistent with the module header definition given above.

```
body B for A;
  module A1 process;
  ip p1: T1(R1) individual queue;
     p2: T1(R2) individual queue;
     p': T(S) individual queue;
  end;
  body B1 for A1; "body definition" end;
  body B2 for A1; "body definition" end;
end;
```

Channel definitions define interactions which are associated with a role in the channel. Thus, an interaction point with a certain role can receive interactions associated with the opposite role in the channel and send interactions associated with its own role (e.g. sender, receiver). The internal interaction point definitions are similar to the interaction point definitions in the module header definition but in this case represent the possible interactions allowed within the module instance or with child instances. The module header and module body definition parts defined within a module body allow the definition of submodules of the module. Module variables are used to identify child module instances to be created within the initialisation part of the module. A state definition enumerates the possible states in which the underlying FSM of the module may be. State sets are used to collectively identify groups of states.

The initialization-part of a module body definition is used to initialize or create the child module and to initialise other objects defined in the body such as states, variables, etc. The following is a typical initialisation part of the module body B for the module A defined above.

```
initialize
 begin
 init X with B1;
 init Y with B2;
 init Z with B1;
 connect X.p1 to Y.p2;
 connect Y.p1 to Z.p2;
 attach p to X.p';
end;
```

The transition-declaration-part is central to the underlying FSM model of the language and defines the possible transitions. The syntax of this part can be complex and will only be described very briefly.

Various clauses can be given to define the conditions under which a transition can be fired. These clauses are **from**, the state or states in which the FSM's must be, **to**, the state the FSM will be after the transition, **when**, which defines an interaction which must be at the front of the queue associated with an interaction point, **provided**, a boolean condition, **priority**, which assigns a priority to the transitions amongst the firable transitions, **delay**, which defines a delay to a firable spontaneous transition, i.e. a transition without a **when**-clause. There may also be an **any** clause, which allows a short-hand notation for transitions.

These clauses can be written in any order, with the constraint that a transition may not contain both a **delay**-clause and a **when**-clause. Clauses can be nested so that a clause can be associated with a number of transition blocks and their clauses. For example, the following illustrates a **to**-clause associated with two transition blocks:

```
trans
  to State0
    from State0 when Ip.I begin...end;
    from State1 when Ip.I begin...end;
```

which is a shorthand notation for the following two transitions.

```
trans
  from State0 to State0 when Ip.I begin...end;
trans
  from State1 to State0 when Ip.I begin...end;
```

The reader is referred to the standards document [5] for a detail description of the language.

A number of Estelle processors have been described in the literature. Before discussing our compiler, which we shall refer to as the Estelle-E compiler, it is interesting to note certain aspects of other Estelle compilers.

3 Other Estelle compilers

The NBS Estelle compiler described by Linn et al. [9] generates C code from an Estelle specification. At the time of writing, the Pascal features not implemented are sets and nested procedures. Variant records and functions returning non-simple types may apparently be used with caution. In addition, true Estelle features which were not implemented include the any-clause and arrays of interaction points. The exist-expression and unspecified types are allowed in restricted form. In general, semantic checking performed during compilation and by the run time environment is limited. LEX and YACC were used in the development of the compiler.

The compiler reported by de Souza and Ferneda [4] was implemented in Prolog and produces Pascal code. The compiler is associated with a software environment for validation and performance analysis. An early version of Estelle is supported. The compiler is called "Estelle/83".

Ansart *et al.* [1] reported a compiler developed by making use of a compiler generation tool called SYNTAX, developed by INRIA in France. The compiler produces a representation of the specification which can be used for simulation, verification and automated implementation. The 1985 DP 9074 Estelle version is supported.

Véda [6], a protocol prototyping tool, includes an Estelle processor which produces pseudo-Pascal as target code. The compiler does not implement dynamic configuration and therefore avoids the associated problems such as parent-child relationships, activities, etc. Véda supports a subset of Estelle called FDT-E, which was based on CCITT's recommendation X.250 and is extended with a simplified description of static architectures.

Another Estelle to C compiler was described by Vuong et al. [14]. LEX and YACC were used to generate the compiler. Features which were not implemented include global variables and nested subroutines. The with-statement is also not supported. Problems encountered as a result of the rather interesting exceptions to scope rules cause further restrictions to be imposed. For example, interaction-argument-identifiers may not be used for local variable identifiers in a module body. Identifiers used in the C routines which are added to create the specification code, may not be used in the specification itself.

Other compilers that the were pointed out to the authors by the referees are those by BULL SA in France [18] and KDD in Japan [11].

4 The Estelle-E compiler

There are clearly many issues in the design of a compiler and in any one case a particular method used may be superior to another. This section describes the issues and experience gained in developing our particular version of an Estelle compiler which we call Estelle-E.

Overview

The Estelle-E compiler is a three pass compiler. It uses recursive-descent as parsing method and on-the-fly code generation to produce a pseudo-code called E-code, based on p-code from the Pascal-compiler [3] and e-code from the Edison compiler [2]. The compiler was implemented using Brinch Hansen's Pascal- compiler [3] as a basis.

Organization

The organization of the Estelle Compiler is as follows:

Pass1: This pass contains the lexical analysis phase of the compiler, scanning the source text and producing an intermediate code file.

Pass2: Three distinct phases can be found in Pass2, namely syntactic analysis (parsing), semantic analysis, and intermediate code generation. Pass2 scans the intermediate code produced by Pass1, and in turn, produces an intermediate code file.

Pass3: Addresses are resolved by Pass3, which also performs simple code optimization. Pass3 reads the intermediate file produced in Pass2 and produces the final pseudo-code file.

The main problem associated with a multipass compiler, namely its poor performance due to enthusiastic disk access, was not considered to be of overriding importance. The advantages gained in having clearly defined output from each phase to examine during development and testing were more important.

Another important design decision concerns the choice of parsing method.

Parsing Method

Initially, the Estelle-E compiler was based on a tabular LL(1) parser, but after encountering problems, the conventional recursive-descent technique was finally used. Using a tabular LL(1) parser initially seemed to solve the problems involved with the implementation of a language under development. We believed that, by using a tabular method and being able to generate a parser, would enable swift changes to be made to

accommodate the new proposals for Estelle. Experience showed that the recursive-descent based compiler was much easier to modify. Problems we encountered with the tabular LL(1) compiler were concerned with semantic analysis and code generation. While the literature provides more than enough information concerning LL(1) parsing, little can be found concerning the back-end of a tabular LL(1)-based compiler. We therefore decided to use recursive-descent, which is well-documented and for which many example compilers exist. Perhaps equally important, the decision to use recursive-descent was also taken in the interests of simplicity.

Choice of target language

It was originally intended that Estelle would be translated to a concurrent programming language or programming language with a multi-tasking facility such as Ada, Edison, C or Modula-2. In fact this is the solution followed by the most of the developers of the compilers mentioned in the previous section.

By contrast, it was decided that the Estelle-E compiler would generate low-level code, which would facilitate the development of the run time environment and the debugger, the meta-implementation provided by the latter being essential for the workbench mentioned before. Specifically, pseudo-code is generated for a theoretical machine known as the Estelle Machine.

While the generation of pseudo-code, as opposed to native code, causes a degradation in the execution speed, the simplicity gained is significant. Major problems inherent in code generation, such as instruction selection and register allocation are avoided, while other problems, such as memory management, are simplified. In addition, by rewriting the Estelle Machine, the system can be ported from one computer architecture to another. The BULL compiler [18] uses the same approach.

By placing another layer between the operating system and executing programs, the interpreter allows a protocol developer or analyst to simulate the environment of the protocol. A method used with a high-level language as the target language is to have a library of routines to link, producing a specific instance of the meta-implementation with certain characteristics. For example, an instance of a protocol is created with queues using a first-come-first-serve queuing discipline. To produce a meta-implementation using a different queuing discipline, a re-compilation or re-link is needed. With the Estelle machine approach, run time changes can be made to, for example, queuing disciplines and error characteristics of the environment.

An important consideration is, naturally, providing for concurrent execution of specifications. One approach (used by some of the other implementors) is to use the concurrent features of the language/operating system to provide concurrency, e.g. each instance of a module is implemented as a process in Unix. The use of E-code was a deliberate attempt to keep control of pseudo-concurrent execution of module instances. In this way it is to an extent possible to examine concurrent execution of the specification.

5 Experience with compiling Estelle

In this section a few of the problems which arose in writing a compiler for the language are highlighted.

The When-clause of the Transition Part

In the draft of the international standard [5] (on which the compiler described here was based) constraints are defined for certain constructs of Estelle. For the **when**-clause the following constraint is given:

> "The occurrence of an interaction-identifier in a **when**-clause shall constitute a defining-point of each interaction-argument-identifier in the interaction-argument-list
> associated with the interaction-identifier for the scope-region associated with the **when**-clause."

Unfortunately, together with the grammar rules and constraints defining transitions, the above constraint implies that interaction-argument-identifiers are visible for a only portion of a scope-region. Because the scope is extended for only a part of the scope-region, the **when**-clause represents an exception to the general scope scheme. Instead of simply adding the interaction-argument-identifiers to the symbol table, it is necessary to remove the entries once the region associated with the **when**-clause is closed. To compound the agony, the syntax does not clearly define an end to a **when**-clause, and this information is difficult to ascertain for a transition-group.

The following example illustrates one way in which a scope region of a **when**-clause can be closed.

1. **trans**
2. **from** $State_0$
3. **to** $State_1$
4. **when** $Ip.I_1(Ia_{11}, Ia_{12}, Ia_{13})$;
5. **provided** $Ia_{11} = Constant$

6. **begin** S_1 **end**;
7. **when** $Ip.I_2(Ia_{21}, Ia_{22}, Ia_{23})$;
8. **provided** $Ia_{21} = Constant$
9. **begin** S_2 **end**;

The scope-region of the interaction-arguments $Ia_{11} \cdots Ia_{13}$ of the interaction I_1 is from line 4 to the beginning of line 7, where the interaction I_2's interaction-arguments $Ia_{21} \cdots Ia_{23}$ become visible. In this case the region associated with the **when**-clause is terminated by being superseded by another **when**-clause. The scope-region of the **when**-clause can also be closed by the symbol "**trans**", which defines the start of another transition-group, the symbol "**end**" which indicates the end of the module body definition, or by a clause which occurred before the **when**-clause in the clause-group.

The problems with the scope region were solved in Estelle-E by first restricting a transition declaration so that only one transition block is permissible per transition declaration. In effect, all that the restriction does is to disallow the use of a short-hand notation for transitions. In addition, the scope region of the transition block is associated with the transition declaration, i.e. the scope region starts as soon as the transition declaration starts, and not when the transition block starts. Interaction-argument-identifiers in a **when**-clause can then be uniquely defined for the scope region in which they are visible. This restriction brings the scope rule for interaction-arguments in line with the scope scheme for Estelle, i.e. an identifier is declared in some declaration-statement and is removed at the end of the block. In the same way the interaction-arguments are defined by the **when**-clause and removed at the end of the transition-block.

Other clauses of the transition part

The **when**-clause was discussed in the previous section as an example where the scope scheme of the compiler had to be circumvented. In general the clauses of a transition declaration together demonstrate an interesting use of the stop set in syntac error recovery. The specification of Estelle defines the following five clauses of a transition-declaration:

> "In a clause-group only one clause of each type, i.e. **provided**-clause, **from**-clause, etc, may occur. In addition, in a clause-group where a **when**-clause occurs, no **delay**-clause may occur, and vice versa. The clauses in a clause-group may occur in any order. The restriction placed on the clause-groups discussed in the previous section, namely that only one clause-group is allowed per transition block means that some of the more complicated rules in the specification are ignored."

The problem to be solved is thus to allow a number of clauses in any order, but only one of each type. In the compiler the problem is solved by starting with all the possible symbols in the clauses, and then removing them from the stop set as soon as the relevant clause has occurred in the clause group. In this way syntactic errors can be generated when clauses are repeated, and the usual syntax error recovery and reporting can be performed. It interesting to note that some of the conditions in the syntactic analysis change to accommodate this scheme. For example, a test

> "**if** *Symbol* **in** *ClauseGroupSymbols* **then** ..."

changes to

> "**if** *Symbol* **in** (*Stop* * *ClauseGroupSymbols*) **then** ..."

i.e. not only must the symbol be a first symbol of a clause, but also in the stop set.

An additional problem caused by a combination of the arbitrary order of clauses and the extension of the scope region by the **when**-clause is that order of evaluation of the clauses is important. The following example from ISO 9074 [5] illustrates the dependence problem:

1. **provided** $q > 1$;
   when $ip.m(q)$; and

2. **when** $ip.m(q)$;
   provided $q > 1$;

In the second case the **provided**-clause tests the parameter "q" obtained in the **when**-clause. In the first case a variable, "q", which happens to have the same name as the parameter in the **when**-clause is tested. Because the **when**-clause opens the scope region, it is important to distinguish between the two cases of evaluation order.

Fortunately, no ambiguity caused by the use of the same name arises, as the scheme used for the **when**-clauses allows a distinction between the two objects, even if the order of evaluation is changed. The parameter "q" and the variable "q" are at different levels of the symbol table and are therefore distinguishable.

E-code generated

The use of on-the-fly code generation was in most respects an adequate solution. For the Pascal features of the language the usual p-code type instructions were generated. For most of the

Estelle constructs single E-code instructions sufficed. However, in the case of the transition-declaration-part it does result in rather messy code. Because clauses must be evaluated, it is necessary to include code for these expressions. Our solution is to produce a series of transition-declaration-statements with references to the sections of E-code representing the clauses. A more sophisticated method of code generation allowing code motion might have produced a better solution.

In general the code produced for a transition-declaration is as follows:

L_1: (* Code for the **when**- or **delay**-clause. *)
L_2: (* Code for the **provided**-clause. *)
L_3: (* Code for the **priority**-clause. *)
(* Code for the procedure and function *)
(* declarations. *)
L_4: (* Code for the statement-part. *)
(* Code to remove interaction in **when**- *)
(* clause from queue in the case of an *)
(* input transition. *)

Note that L_1, L_2 and L_3 can be in any order.

The following instruction is in general generated for a transition declaration:

Transition(*From*, *To*, *Input*, L_1, L_2, L_3, L_4)

where "*From*" is a state set constant indicating from which state(s) the transition may be made, "*To*" is the state to which the current state must change, (−1 indicates the same state), while "*Input*" is a boolean indicating whether the transition is an input transition or a spontaneous transition, "L_1" contains the address of the **when**-clause or **delay**-clause for input and spontaneous transitions respectively, and "L_2" and "L_3" contain the addresses of the **provided**-clause and **priority**-clause respectively. "L_4" contains the address of the transition block.

The code produced for the example transitions in the section concerning **when**-clauses would be as follows:

L_5: **Variable**(Ip_l, Ip_d)
Interaction(I_1)
Return
L_6: (* Code for $Ia_{11} = Constant$ *)
Return
L_7: (* Code for S_1 *)
(* Code to remove I_1 from the queue. *)
EndTrans
L_8: **Variable**(Ip_l, Ip_d)
Interaction(I_2)
Return
L_9: (* Code for $Ia_{21} = Constant$ *)
Return
L_{10}:(* Code for S_2 *)
(* Code to remove I_2 from the queue. *)
EndTrans
L_{11}:**Transition**([0], 1, 1, L_5, L_6, −, L_7)
Transition([0], 1, 1, L_8, L_9, −, L_{10})
EndTransitions

Where Ip_l and Ip_d are the block-level and relative displacement of the interaction point Ip respectively. The E-code instruction **Variable**(Ip_l, Ip_d) places the address of the interaction point onto the stack. The scheduler uses the transition statements to set up a table of transitions. When testing to see which transitions are firable, the scheduler need only find one clause which is not satisfied to find a transition unsuitable. It is therefore advantageous to have the clauses separate to speed up the transition evaluation function.

Other problems with Estelle

From the viewpoint of the implementor of an Estelle compiler, the form of **forone**-, **exist-one**- and **all**-statements is difficult, as these statements can contain either a domain-list or a module-domain, necessitating at least three symbol look-ahead in some cases to distinguish between the two forms. As the domain-list form of the statements can be expressed in conventional constructs their inclusion contributes nothing to the language and makes compilation messy. For example, the **all**-statement

"**all** x : *Seq* **do**"

where "*Seq*" is the integer subrange "0..1", can be written as

"**for** x := 0 **to** 1 **do**"

with a previously declared variable "x" of type "*Seq*".

In addition, in the case of assignment-statements, semantic information is necessary to distinguish between syntactic constructs. For example, if V_1 is a character, and V_2 and V_3 are integers, the expression "$V_1 := V_2 + V_3$" is syntactically correct but semantically incorrect. However, in the case where V_1, V_2 and V_3 are module variables, the assignment is syntactically incorrect as an expression is not permissible on the right-hand side of a module variable assignment. Only semantic information can be used for correctly checking syntax, forcing the compiler to collect semantic information even when the "**any** constant" or "..." constructs make it impossible to perform complete semantic analysis of the specification.

Minor aspects

The standard data type "**real**" is defined for Estelle. The use of this data type in communication

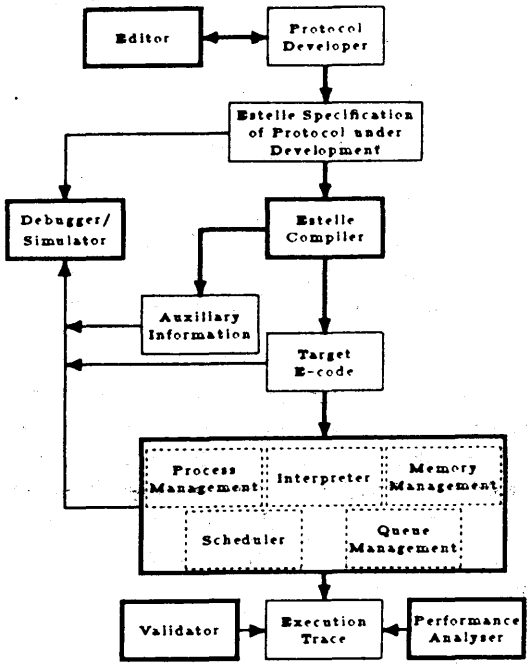

Figure 1: Functional components of the Protocol Engineering Workbench.

software is so unusual that it may as well have been omitted.

The inclusion of subranges in a strongly typed language can be argued against on the basis that types can be introduced that are either disjoint, contained in other subranges, or even overlapping [2]. Run-time testing is therefore necessary to decide whether two types are compatible.

Similar arguments are possible for sets which have members defined by subranges as compatibility can not be tested statically, and only with great difficulty during run-time execution.

6 Estelle and the PEW

The PEW is a software environment which consists of the various functional components illustrated in Figure 1. The reader is referred to that figure for the following description. The compiler translates the Estelle description of the software to be developed to E-code and furthermore provides access to auxiliary information such as the symbol table of the compiler.

The E-code serves as input to the run time environment. The run time environment consists of the various modules illustrated which together allow pseudo-concurrent execution of the E-code.

An important component of the workbench is the debugger which, in conjunction with the auxiliary information provided by the compiler, the specification itself and the run time environment, is used to examine the execution of the protocol machine. The debugger can display and execute specification code (the meta-implementation), control execution flow, and examine and change values in memory and message queues as well as process queues.

The debugger is also used to simulate errors in the run time environment (such as losing a message on a queue) and to generate an execution trace recording the progress through the code. This information is then used in the analysis component of the workbench consisting of a validator and a performance analyzer. The entire system is designed to run on an 80286 processor with an 80287 coprocessor (the latter for the performance analyzer [8]).

7 Conclusion

The orginal design of our compiler for the formal protocol specification language Estelle used a tabular LL(1) parser and was to use a high-level language such as Ada or C as the target language.

The LL(1) parser was used in the design in order to rapidly accommodate changes in the language which was still being developed. It turned out that a recursive-descent technique was equally amenable to changes and was preferred because of its relative simplicity.

Rather than a high-level target language, a special pseudo-code for a theoretical Estelle machine was used in order to facilitate the design of the software development environment around the compiler.

In common with a number of other Estelle compiler writers, restrictions had to be placed on certain Estelle contructs to facilitate compilation. Transition declarations were restricted, as were **all**- and **forone**-statements and **exist**-expressions. Moreover, real numbers and **with**-statements were excluded because they do not contribute significantly to the language, while **any**-clauses were excluded as a result of the restrictions placed on transitions.

References

[1] J P Ansart *et al.*, [1986], Software Tools for Estelle, *Protocol Specification, Testing and Verification VI.*, B Sarikaya and G V Bochmann (Eds.), Elsevier North-Holland, Amsterdam, 55-61.

[2] P Brinch Hansen, [1982], *Programming a Personal Computer*, Prentice-Hall, Englewood Cliffs, New Jersey.

[3] P Brinch Hansen, [1985], *Brinch Hansen on Pascal Compilers*, Prentice-Hall, Englewood Cliffs, New Jersey.

[4] W L de Sousa and E Ferneda, [1988], A Compiler for a Formal Description Technique, *Computer Communication Systems.*, A G Cerveira (Ed), Elsevier North-Holland, Amsterdam, 275-283.

[5] ISO/IEC JTC 1/SC 21N 3611 [1989], Final Text of ISO/DP 9074, Information Processing Systems – Open Systems Interconnection – Estelle – A Formal Description Technique Based on an Extended State Transition Model.

[6] C Jard *et al.*, [1988], Development of Véda, a Prototyping Tool for Distributed Algorithms, *IEEE Trans. on Software Engineering*, **14**(3), 339-352.

[7] P S Kritzinger, [1986], A Performance Model of the ISO Communication Architecture, *IEEE Trans. on Communications*, **34**(6), 554-563.

[8] P S Kritzinger, [1988], A Modelling Technique and Tool for Communication Protocol Performance Evaluation, Tech. Rept. CS-88-01-00, Dept. Computer Science, Univ. of Cape Town, 24p.

[9] R J Linn et al., [1986], User Guide for the NBS Prototype Compiler for Estelle, Report No. ICST/APM 87-1, U.S. National Bureau of Standards (ICST).

[10] [1987] User Guide for the BULL Prototype Compiler for Estelle, BULL S.A., DRCG/ARS, 78430 Louveciennes, France.

[11] [1988] T Hasagawa *et al.*, [1988] An Automated ADA Program Generator from Protocol Specifications Based on Estelle and ANS.1, *Proceedings of the 9th International Conference on Computer Communication*, Tel Aviv, Israel.

[12] [1986] D Rayner, Towards standardized OSI conformance tests, *Protocol Specification, Testing and Verification V.*, M Diaz (Ed.) Elsevier North-Holland, Amsterdam, 441-460.

[13] J van Dijk, [1988], An Estelle Compiler, M.Sc. Thesis, Dept. Computer Science, Univ. of Cape Town.

[14] S T Vuong *et al.*, [1988], Semiautomatic Implementation of Protocols Using an Estelle-C Compiler, *IEEE Trans. on Software Engineering*, **14**(3), 384-393.

[15] C H West, [1978], General Technique for Communication Protocol Validation, *IBM J. Res. Develop.*, **22**(4), 393-404.

[16] M Diaz *et al.* [1989] (Editor), The Formal Description Technique Estelle, North Holland.

[17] [1989] ISO/IEC JTC 1/SC 21/N 3249: Proposed Draft Addendum to ISO 9074:1989: Estelle Tutorial, December 1989.

[18] S Budkowski and P Dembinski [1987], An Introduction to Estelle, *Computer Networks and ISDN Systems Journal*, **14**(1), 3-23.

Acknowledgements: Apart from the authors, Graham Wheeler and Johann Dreyer are also involved with the development of the PEW and have contributed to the design. The authors would also like to thank one of the referees whose extensive comments improved the quality of the paper considerably.

NOTES FOR CONTRIBUTORS

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as a 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 double-space typing on one side only of A4 paper, and provide wide margins.
- 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 on separate sheets of A4 paper, and should be numbered and titled. Figures should be submitted as original line drawings, and not photocopies.
- Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin. Distinguish clearly between such cases as:
  - upper and lower case letters;
  - the letter O and zero;
  - the letter I and the number one; and
  - the letter K and kappa.
- References should be listed at the end of the text **in alphabetic order** of the (first) author's surname, and should be cited in the text in square brackets. References should thus take the following form:
  [1] E Ashcroft and Z Manna, [1972], The translation of 'GOTO' programs to 'WHILE' programs, *Proceedings of IFIP Congress 71*, North-Holland, Amsterdam, 250-255.
  [2] C Bohm and G Jacopini, [1966], Flow diagrams, Turing machines and languages with only two formation rules, *Comm. ACM*, **9**, 366-371.
  [3] S Ginsburg, [1966], *Mathematical theory of context free languages,* McGraw Hill, New York.

Manuscripts *accepted* for publication should comply with the above guidelines, and may provided in one of the following three formats:

- in a **typed form** (i.e. suitable for scanning);
- as an **ASCII file** on diskette; or
- in **camera-ready** format.

A page specification is available on request from the editor, for authors wishing to provide camera-ready copies.

Charges

A charge per final page, scaled to reflect scanning, typesetting and reproduction costs, will be levied on papers accepted for publication. The costs per final page are as follows:

Typed format:	R80-00
ASCII format:	R60-00
Camera-ready format :	R20-00

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

Proofs

Proofs of accepted papers 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 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.

South African Computer Journal

Number 2, May 1990
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 2, Mei 1990
ISSN 1015-7999

Contents

EDITORIAL 1

GUEST CONTRIBUTION
Opening Address: Vth SA Computer Symposium 3
Prof H C Viljoen

RESEARCH ARTICLES

Homological Transfer: an Information Systems Research Method 6
J Mende

On the Generation of Permutations 12
D Naccache de Paz

Coping with Degeneracy in the Computation of Dirichlet Tessellations 17
J Buys, H J Messerschmidt and J F Botha

An Estelle Compiler for a Protocol Development Environment 23
P S Kritzinger and J van Dijk

Predicting the Performance of Shared Multiprocessor Caches 31
H A Goosen and D R Cheriton

Implementing UNIX on the INMOS transputer 39
P J McCullugh and G de V Smit

Data Structuring via Functions 45
B H Venter

COMMUNICATIONS AND REPORTS

FRD Investment in Advanced Computer Science Training 51
ADA Courses and Workshop 51
Book Reviews 52
A Review of the Use of Computers in Education in South Africa
Part I: Primary and High Schools 53