**4^{de}
th**

**SUID-AFRIKAANSE REKENAARSIMPOSIUM
SOUTH AFRICAN COMPUTER SYMPOSIUM**

**HOLIDAY INN PRETORIA
JULIE 1 – 3 JULY 1987**

Proceedings
of the
4th South African Computer Symposium

Holiday Inn, Pretoria

1 – 3 July 1987

edited by

Pieter Kritsinger

**Computer Science Department
University of Cape Town**

PREFACE

Computer science is an emerging discipline which is having difficulty in being recognised as a worthy member of the sciences. I will paraphrase John Hopcroft, co-winner of the 1986 Turing Award, when, during a recent interview, he said that the primary reason for the lack of recognition, is the age of our researchers. Probably not one of the researchers who presented their work at this symposium is older than 45. I know of no computer scientist in South Africa who is in a position where (s)he can affect funding priorities. As far as I know we have no representation on any of the committees of the Foundation for Research Development and for our Afrikaans speaking fraternity, none who is a member of the *Akademie vir Wetenskap en Kuns*. It will take time and conscious effort to establish our presence. The same is true of course for our universities. Again, with one exception, I know of no dean of a science faculty, vice-principal or principal who is a computer scientist. We consequently spend an enormous amount of time trying to explain the needs of computer science and its difficulties. I believe this symposium is a further step towards accreditation by our peers and superiors from the other sciences.

The total number of papers submitted to the Programme Committee for consideration was 34. Each paper was reviewed by three persons knowledgeable in the field it represents. Of those submitted, 23 were finally selected for inclusion in the symposium. As a result the overall quality of the papers is high and as a computer science community in Africa we can be justly proud of the final programme.

This is the fourth in the series of South African computer symposia. This year the symposium is sponsored by the Computer Society of South Africa (CSSA), the South African Institute for Computer Scientists and the local IFIP Committee. The executive director of the CSSA and his staff deserve warm thanks for handling the organisation as well as they have, while the Organising Committee provided Derrick and I with very valuable advice.

Finally I would like to express my sincere appreciation to the authors, to the members of the Programme Committee and particularly the reviewers. Without the kind cooperation of everyone, this symposium would not have taken place.

Pieter Kritzinger
July 1987.

SYMPOSIUM CHAIRMAN: PS Kritzinger, University of Cape Town.

SYMPOSIUM CO-CHAIRMAN: D Kourie, University of Pretoria.

MEMBERS OF THE PROGRAMME COMMITTEE

Judy Bishop, Witwatersrand University

Chris Bornman, UNISA

Hannes de Beer, Potchefstroom University

Gideon de Kock, Port Elizabeth University

Jaap Kies, Western Cape University

Derrick Kourie, Pretoria University

Pieter Kritzinger, Cape Town University

Tony Krzesinski, Stellenbosch University

Michael Laidlaw, Durban Westville University

Peter Lay, Cape Town University

Ken MacGregor, Cape Town University

Theo McDonald, Orange Free State University

Jan Oosthuizen, University of the North

Dennis Riordan, Rhodes University

Alan Sartori-Angus, Natal University

John Shochot, Witwatersrand University

Theuns Smith, Rand Afrikaans University

Trevor Turton, ISM (Pty) Ltd

Gerrit Wiechers, Infoplan.

LIST OF REVIEWERS

BERMAN Sonia

BISHOP Judy

BORNMAN Chris

CAREY Chris

CHERENACK Paul

DE BEER Hannes

DE VILLIERS Pieter

GORRINGE Pen

KIES Jaap

KOURIE Derrick

KRITZINGER Pieter

KRZESINSKI Tony

LAIDLAW Michael

LAY Peter

MacGREGOR Ken

MATTISON Keith

McDONALD Theo

RENNHACKKAMP Martin

RIORDAN Denis

SATORI-ANGUS Alan

SCHOCHOT John

SMITH Theuns

TURTON Trevor

VAN DEN HEEVER Roelf

VAN ROOYEN Hester

VON SOLMS Basie

VOS Koos

TABLE OF CONTENTS

Keynote Address

"An Extensible System and Programming Tool for Workstation Computers."	1
Niklaus Wirth, ETH, Zurich	

Invited Lectures

"The Relationship of Natural and Artificial Intelligence."	not included in Proceedings.
G Lasker, University of Windsor, Ontario.	

"Software Engineering: What Can We Expect in the Future?"	not included in Proceedings.
D Teichrow, University of Michigan, U.S.A.	

Computer Languages I

"SPS-Algol: Semantic Constructs for a Persistent Programming Language."	13
S Berman, University of Cape Town.	

"Petri Net Topologies for a Specification Language."	25
R Watson, University of the Witwatersrand.	

"Towards a Programming Environment Standard in LISP." ...	45
R Mori, University of Cape Town	

"ADA for Multiprocessors: Some Problems and Solutions." ..	63
J Bishop, University of the Witwatersrand.	

Computer Graphics

"Polygon Shading on Vector Type Devices."	75
C F Scheepers, CSIR.	

"Hidden Surface Elimination in Raster Graphics Using Visigrams."	97
P Gorringer, CSIR.	

Database Systems I

"On Syntax and Semantics Related to Incomplete Information Databases."	109
M E Orlowska, UNISA.	

"Modelling Distributed Database Concurrency Control Overheads."	131
M H Rennhackkamp, University of Stellenbosch.	

Operating Systems

"The Development of a Fault Tolerant System for a Real-time Environment."	149
M Morris, CSIR.	

"A New General-purpose Operating System."	161
B H Venter, CSIR.	

Computer Languages II

"The Representation of Chemical Structures by Random Context Structure Grammars." 175
E M Ehlers and B von Solms, RAU.

"A Generalised Expression Structure." 189
W van Biljon, CSIR.

Computer Networks and Protocols I

"An Approximate Solution Method for Multiclass Queueing Networks with State Dependent Routing and Window Row Control." 203
A E Krzesinski, University of Stellenbosch.

"A Protocol Validation System." 227
J Punt, University of Cape Town.

Computer Networks and Protocols II

"Protocol Performance Using Image Protocols." 251
P S Kritzing, University of Cape Town.

Artificial Intelligence

"A Data Structure for Exchanging Geographic Information." 267
A Cooper, CSIR.

"The Design and Use of a Prolog Trace Generator for CSP." 279
D G Kourie, University of Pretoria.

Database Systems II

"An Approach to Direct End-user Usage of Multiple Databases." 297
M J Phillips, CSIR.

"A Semantic Data Model Approach to Logical Data Independence." 329

S Berman, University of Cape Town.

Information Systems

"The ELSIM Language: an FSM-based Language for the ELSIM SEE." 343
L du Plessis and C Bornman, UNISA.

"Three Packaging Rules for Information System Design." . 363
J Mende, University of the Witwatersrand.

Computer Languages III

"Experience with a Pattern-matching Code Generator." ... 371
M A Mulders, D A Sewry and W R van Biljon, CSIR.

"Set-oriented Functional Style of Programming." 385
C Mueller, University of the Witwatersrand.

Tutorial

The use of Modula-2 in Software Engineering." 399
N Wirth, ETH, Zurich.

DAY 1

07h30	Registration and Coffee.
08h45	Welcoming address, President of the South African Institute of Computer Scientists, Dr. G. Wiechers.
09h00	Invited Lecture. Professor D. Teichrow, University of Michigan. <i>Software Engineering, ... What Can We Expect in the Future.</i>
10h00	COFFEE Computer Languages I. Chairman: G. Wiechers.
10h15	S. Berman, University of Cape Town. <i>SPS-Algol: Semantic Constructs for a Persistent Programming Language.</i>
10h50	A. Watson, University of the Witwatersrand. <i>Petri Net Topologies for a Specification Language.</i>
11h25	R. Mori, University of Cape Town. <i>Towards a Programming Environment Standard in USP.</i>
11h50	J. Bishop, University of the Witwatersrand. <i>ADA for Multiprocessors: Some Problems and Solutions.</i>
12h30	LUNCH

	Computer Graphics. Chairman: D. Kourie	Operating Systems. Chairman: K. MacGregor.
14h00	C. F. Scheepers, CSIR. <i>Polygon Shading on Vector Type Devices.</i>	M. Morris, UNISA. <i>The Development of a Fault Tolerant System for a Real-time Environment.</i>
14h35	P. Gorrige, CSIR. <i>Hidden Surface Elimination in Raster Graphics Using Visigrams.</i>	B. H. Venter, CSIR. <i>A New General-purpose Operating System.</i>
15h15	COFFEE	COFFEE
	Database Systems I. Chairman: B. von Solms.	Computer Languages II. Chairman: J. Bishop.
15h30	M.E. Orlowska, UNISA. <i>On Syntax and Semantics Related to Incomplete Information Databases.</i>	E.M. Ehlers and B. von Solms, Randse Afrikaanse Universiteit. <i>The Representation of Chemical Structures by Random Context Structure Grammars.</i>
16h05	M.H. Rennhackkamp, Stellenbosch University. <i>Modelling Distributed Database Concurrency Control Overheads</i>	W. van Biljon, CSIR. <i>A Generalised Expression Structure.</i>

18h00

Cocktail Party in Cullinan Room A.

DAY 2

- 08h30 Keynote Address by Profesor Niklaus Wirth, Swiss Federal Institute for Technology, Zurich.
An Extensible System and a Programming Tool for Workstation Computers.
- Computer Networks and Protocols I.** Chairman: P.S. Kritzinger.
- 09h30 A.E. Krzesinski, University of Stellenbosch.
An Approximate Solution Method for Multiclass Queueing Networks with State Dependent Routing and Window Flow Control.
- 10h05 J. Punt, University of Cape Town.
A Protocol Validation System.
- 10h30 COFFEE
- Computer Networks and Protocols II.** Chairman: R. van der Heever.
- 11h00 P.S. Kritzinger, University of Cape Town.
Protocol Performance using Image Protocols.
- 11h35 Invited Lecture by Professor G. Lasker, University of Windsor, Ontario.
The Relationship of Natural and Artificial Intelligence.
- 12h30 LUNCH

<p>Artificial Intelligence.
Chairman: G. Lasker.</p> <p>14h00 A. Cooper, CSIR
<i>A Data Structure for Exchanging Geographic Information.</i></p> <p>14h35 A. I. Newcombe, University of Cape Town and R. Rada, National Library of Medicine, Maryland.
<i>Strategies for Automatic Indexing and Thesaurus Building.</i></p> <p>15h15 COFFEE</p> <p>Database Systems II.
Chairman: C. Bornman.</p> <p>15h30 M.J. Philips, CSIR.
<i>An Approach to Direct End-user Usage of Mutiple Databases.</i></p> <p>16h05 S. Berman, University of Cape Town.

<i>A Semantic Data Model Approach to Logical Data Independence.</i></p> | <p>Information Systems.
Chairman: D. Teichrow.</p> <p>L du Plessis and C. Bornman, UNISA.
<i>The ELSIM Language: an FSM-based Language for the ELSIM SEE.</i></p> <p>J. Mende, University of the Witwatersrand.

<i>Three Packaging Rules for Information System Design.</i></p> <p>COFFEE</p> <p>Computer Languages III.
Chairman: N. Wirth.</p> <p>W. van Biljon, CSIR.
<i>Experience with a Pattern-matching Code Generator.</i></p> <p>C. Mueller, University of the Witwatersrand.
<i>Set-oriented Functional Style of Programming.</i></p> |
|--|--|

- 16h45 Open Forum with professors G. Lasker, D. Teichrow and N. Wirth.
Moderator: Dr. D. Jacobson.
- 19h30 Symposium Banquet in Cullinan Room.
Guest speaker, Dr. D. Jacobson, Group Executive: Technology, Allied Technologies Limited.

DAY 3

08h00 Registration (Tutorial only).

08h30 Tutorial.

The Tutorial will be given by professor Niklaus Wirth, Division of Computer Science, Swiss Federal Institute of Technology, Zurich.

The use of Modula-2 in Software Engineering.

Topics to be covered include:

What is Software Engineering?

Data types and structures.

Modularization and information hiding.

Definition and implementation parts.

Separate compilation with type checking.

Facilities to express concurrency.

Pompous programming style.

What could be excluded?

12h15 **Close of Symposium.**

12h30 LUNCH

The Design and Use of a Prolog Trace Generator for CSP

D. G. KOURIE

Department of Computer Science, University of Pretoria, Hatfield, Pretoria 0083, South Africa

SUMMARY

The role of traces in the context of formal description techniques is discussed, as well as issues arising in the generation of these traces by means of software. A trace generator (written in Prolog) for systems described in terms of CSP is outlined, with emphasis on the translation from CSP to Prolog, the central implementation issues in a Prolog context, and techniques for generating all traces up to termination or recursion. Some experiences with the use of the generator are mentioned, and potential extensions for generating traces of LOTOS descriptions are mentioned.

INTRODUCTION

The need for formally describing a complex system (sometimes referred to as defining the system) before implementation is widely acknowledged, not only in order to communicate to the implementor what is to be done, but also to better understand the complexities of the system before implementation. To this end, several formal description techniques (FDTs) have recently been developed, including Estelle,¹ LOTOS² and CSP.³ The latter technique is due to Hoare, and together with LOTOS, stems from the work of Milner.⁴

This paper describes the design and implementation of a trace generator for an important subset of CSP. The trace generator is written in Waterloo Core Prolog (version 1.7) and runs on a Hitachi PS 7/83 in a 2 megabyte VM/CMS partition. It is designed to provide a list of traces of a given system, up until either termination or recursion is indicated. Special attention is paid to minimizing the effort required to translate a CSP definition into a Prolog format for input to the generator.

In the next section the notion of a trace is briefly and informally discussed in order to provide the necessary background. Approaches of other researchers to the generation of traces and the implementation of FDTs are considered, and a motivation for the present research is given.

The following section then describes the way in which CSP definitions may be translated into input to the system. This hinges on the use of Prolog operators to parallel the implemented CSP operators. The syntax and semantics of each such operator are briefly and informally given. A section then follows which outlines the central recursive approach to trace generation in Prolog. Some peripheral design issues are discussed in the succeeding section.

Finally, experiences with the trace generator to date are mentioned, possible extensions are surveyed and some conclusions suggested by the study are drawn.

Traces as a means of characterizing a process

In both CSP and LOTOS a process is defined in terms of its behaviour *vis-à-vis* its environment. This behaviour is expressed by referring to atomic synchronous events by means of which the process interacts with its environment. From this viewpoint a process may be described at two levels of formality and rigour.

In the first place, the FDT language itself may be used to formally and rigorously describe what process transformation(s) may potentially occur when events in a process environment are offered to the process. Such a description is based upon the use of operators which, in general, conjoin events to each other and/or to named subprocesses which represent the subsequent (transformed) behaviour of the process. The syntax and semantics of the implemented CSP operators are discussed later.

However, a second, but less precise, way of characterizing a process—which is derivable from the first description—is to provide the set of so-called traces of the process. By this is meant a set whose elements consist of event sequences (possibly of infinite length), with the following properties. When presented with a trace (i.e. an event sequence), a deterministic process will always interact with (or 'accept') each successive event in turn, and be transformed to a new subprocess which will accept the next event. More concisely, a deterministic process may be said to accept each of its traces.

If the possibility that a non-deterministic process will accept a sequence of events may be ruled out *a priori*, then the sequence does not constitute a valid trace of the process. Otherwise, the event sequence is a trace which the process may potentially accept, but may also non-deterministically refuse to accept. In this sense, the traces of a non-deterministic process only partially describe non-deterministic process behaviour. Furthermore, there is not a one-to-one correspondence between a given set of traces and a process.

Traces, as a method of characterizing processes, are thus imprecise. Nevertheless, when trying to get a feel of the process's behaviour, it is intuitively appealing to examine some or all of its traces, since they are suggestive of the process behaviour in a less abstract way than the formal process description. Furthermore, in verifying the process definition itself it is sometimes necessary to make some statement about its traces (usually postulated as a first order predicate calculus formula) and then to prove that the process definition satisfies this statement. Hence, obtaining information about the traces of a formally described process is both useful and potentially necessary for verification. For this reason several efforts have been made to write software which relates traces to process definitions.

Software to relate traces to process definitions

One way of doing this is to have a system which 'implements' an FDT definition, i.e. uses the definition together with an event sequence as input, and answers 'yes' or 'no' according to whether the sequence has been accepted. The significance of a 'no' reply for a non-deterministic process definition depends on how the system treats non-determinism.

1. In some cases it may mean that the sequence always leads to deadlock on the particular implementation. In such an implementation, deterministic choices are always made at points where the definition specifies the possibility of a non-deterministic choice. (Note that this would nevertheless be a valid implementation of the non-deterministically defined process.)
2. Alternatively, a 'no' reply might indicate that this particular implementation has made a non-deterministic choice which led to the deadlock, but that the same system may, if rerun, make an alternative choice. Non-deterministic choices are therefore implemented using some random number function.
3. Finally, if the implementation keeps track of all possible trace paths, a 'no' reply will mean that the sequence will always lead to deadlock for any implementation.

Hoare³ gives an outline of how Lisp may be used to implement a CSP process definition, and discusses the first and the last approach for treating non-determinism. Ural and Probert⁵ use Prolog as a vehicle for implementing Estelle process definitions and base their treatment of non-determinism on the third alternative. Gilbert⁶ has implemented LOTOS in Parlog using the second alternative for handling non-determinism. This system, however, constructively generates a trace of the implementation, i.e. produces an event sequence as output, rather than responding to a given input sequence.

A further issue in systems of the above nature is that of translating the FDT definition to a form acceptable for the target implementation system. In the aforementioned works, Hoare³ and Ural and Probert⁵ rely on hand translation, whereas the work of Gilbert⁶ is orientated towards automated translation.

Orientation of this paper

The approach taken here is to generate 'all' traces of a process, dealing with non-determinism according to the third approach described above. The notion of 'all' traces should, however, be qualified as follows:

1. It is assumed that the environment of the process for which traces are being sought will always offer an appropriate event for interaction—i.e. that processes will only deadlock if the so-called 'stop' process is explicitly used in the process definition or if it arises by virtue of the use of the concurrency operator (as discussed below). When recursion arises, this is detected and indicated by a special notation.
2. The resulting traces may, in principle, be divided into two lists, say s and t , with s prefixed to t . Two limiting divisions would render s the empty list on the one hand, and t the empty list on the other. In terms of the definition of a trace² each possible s obtainable from such a division is also a trace of the process, including the division for which s is the empty list. Such a trace signifies that after the last event in the trace is offered by the environment, it then either does not offer another event to the process, or possibly offers an event with which the

process cannot interact. Although it would be trivial to provide all traces of this nature, it is somewhat redundant since these subtraces can be readily deduced from the traces generated.

The decision to focus on CSP rather than LOTOS, at least initially, was primarily motivated by a desire for a system which would be relatively easy to use, both as a tool for teaching FDTs, and as a practical tool for system design. It was felt that since CSP is syntactically both more economical and somewhat less formalized, and since it is more comprehensively described in the literature (owing to the eloquence of its author) than is LOTOS (which is the product of a committee), hand translations from CSP to Prolog would be made relatively easily. In fact, it turns out that the Prolog representation may be designed to be so close to CSP that the question of translation hardly arises.

However, in the longer term it is hoped to modify and extend the work in order to provide a LOTOS trace generator. This is part of a project to provide an environment for LOTOS, and to this end a LOTOS syntax checker has already been developed.⁷ Because of this longer term orientation towards LOTOS, and because of the suggestiveness of some of the terminology used in LOTOS documents, this terminology will sometimes be used below.

TRANSLATING CSP TO PROLOG

General format for process abstractions

The trace generator is in a file which is consulted upon entering the Prolog interpreter. It expects input regarding the system of processes to be traced in the form described below. Normally this input will be contained in a file which the Prolog interpreter consults, and views as additional facts to be added to the Prolog database.

An input file therefore contains one or more (say N) Prolog facts (representing so-called CSP process abstractions) of the form:

$$X := Y.$$

where $:=$ is defined in the trace generator as an infix (Prolog) operator with arguments X and Y .

X is a structure with 0 or more arguments and is the name of a process whose traces are to be generated from its definition as contained in Y , the so-called behaviour expression part of the process abstraction $X := Y$. Some of the arguments of X may be (Prolog) variables which occur in Y . Process X may be referenced from the behaviour expression part of any (including its own) process abstraction. In such a reference (called a process instantiation) 0 or more of the variable arguments may be instantiated to an integer or atomic value.

The behaviour expression Y is the definition of the process X and has the form:

$$Y_1 \text{ Op1 } Y_2 \text{ Op2 } \dots \text{ Op}_{n-1} \text{ } Y_n. (n > 0)$$

Here, (with some exceptions discussed below) each Y_i is either a process instantiation or it is a subexpression (say Z) whose form is similar to Y , or it is considered to be an event (i.e. provided that it is a valid Prolog term).

Operator syntax

Each *Opi* represents one of a set of CSP operators which have been defined as Prolog infix operators in the trace generator. The following is a list of the currently implemented operators (note that owing to constraints imposed by both the keyboard used and the Prolog interpreter used, the notation differs slightly from the original CSP):

```
--> ,
; , | , <> , |-,
|| , ||| , ^>
```

Each line above contains operators declared with the same precedence, the first line operator having the lowest precedence (i.e. binding its operands the tightest) and the last line the highest precedence (i.e. binding its operands the weakest). Furthermore, all operators are declared as right-associative.

The implication of the above is that, for example, $a \text{-->} b \text{-->} c$ is interpreted as $a \text{-->} (b \text{-->} c)$, (by virtue of the right-associativity), that $a ; b <> c$ is interpreted as $a ; (b <> c)$, (again by right-associativity, since the operators have equal precedence), and that $a ; b || c$ is interpreted as $(a ; b) || c$ (since the precedence of $;$ is lower than that of $||$). However, parentheses may be used to enforce associativity as required.

Operator semantics

The expression $a \text{-->} \text{proc}$ is read as 'event a then proc ', and describes a process which, when presented by its environment with event a will be described by the process instantiation proc (or by the event proc if proc is not a process instantiation). Note that the trace generator always treats whatever appears to the left of the --> operator as an event.

If it is desired that a procedure described by the behaviour expression proc1 should follow on a procedure described by the behaviour expression proc2 , this is expressed by using the successor operator $;$ as follows: $\text{proc1} ; \text{proc2}$. If a trace of proc1 ends with the special behaviour expression called *stop*, then that trace is also the trace of $\text{proc1} ; \text{proc2}$, since *stop* is used to describe a process which never actually engages in any events which its environment may present. Otherwise a trace of $\text{proc1} ; \text{proc2}$ is a trace of proc1 prefixed to a trace of proc2 .

The behaviour expression $a \text{-->} \text{proc1} | b \text{-->} \text{proc2}$ is read as 'a then proc1 choice b then proc2 '. This describes a process which will interact with either event a or event b if these events are presented for interaction by the process's environment. Subsequent behaviour of the process is then described by proc1 or proc2 , respectively, depending on which event was first presented. It is syntactically required by CSP that a and b be different, but the trace generator will not check this.

If $\text{proc_a} := a \text{-->} \text{proc1}$ and $\text{proc_b} := b \text{-->} \text{proc2}$, then the following expression is used to denote general choice:

```
proc_a <> proc_b.
```

The semantics of this behaviour expression is identical to $a \text{-->} \text{proc1} | b \text{-->} \text{proc2}$, provided that a and b are different. However, if they are the same, the process that is

described by $\text{proc_a} <> \text{proc_b}$ will accept event a and subsequently non-deterministically (from the environment's point of view) decide to behave according to either proc_1 or proc_2 .

At a more generalized non-deterministic level, if proc_a and proc_b are defined as above, then the behaviour expression

$$\text{proc_a} \mid \mid \text{proc_b}$$

describes a process which, *ab initio*, non-deterministically chooses which path it will follow, even if a and b are different: i.e. it will either interact with an event a presented by the environment and subsequently behave as proc_1 , or with an event b and subsequently behave as proc_2 . However, from the environment's perspective, it is only possible to deduce, *a posteriori*, which choice the process has made.

The concurrency operator, $\mid \mid$, is used to denote how two processes may synchronize (i.e. mutually interact) with each other. Events with which the global process described by the behaviour expression $\text{proc_a} \mid \mid \text{proc_b}$ can interact are divided into two categories: those events with which both proc_a and proc_b can interact (call these synchronizing events), and those events which these processes do not share (called non-synchronizing events). If, at some point, proc_a and proc_b are willing to interact with mutually disjoint subsets of synchronizing events only, then a deadlock occurs and $\text{proc_a} \mid \mid \text{proc_b}$ degenerates into the process called stop. Otherwise, if the sets are not disjoint but are still limited to synchronizing events, they will simultaneously interact with a common synchronizing event (i.e. they will synchronize). However, if one or more of these sets also includes non-synchronizing events, the relevant process may first interact with such an event and later either synchronize with its partner process or deadlock.

The interleaving operator, on the other hand, simply ignores all synchronizing possibilities and the process described by the behaviour expression $\text{proc_a} \mid \mid \mid \text{proc_b}$ will accept any interleaving derived from a proc_a trace and a proc_b trace, respectively. If, at any point, both proc_a and proc_b will accept a specific event offered by the environment, then in the behaviour expression $\text{proc_a} \mid \mid \mid \text{proc_b}$ only one of the component expressions (chosen non-deterministically from the environment's point of view) will interact with the event.

Finally, the interrupt operator, $\sim >$, is used to show that the process described by the behaviour expression $\text{proc_a} \sim > \text{proc_b}$ will behave precisely as defined by proc_a , until the environment offers an event which proc_b can accept. The subsequent behaviour of $\text{proc_a} \sim > \text{proc_b}$ then corresponds to the subsequent behaviour of proc_b . This interruption may occur right at the start, but may not occur once proc_a either terminates successfully or deadlocks.

Exceptions to operator syntax

The exceptions to the above syntax are as follows: in the process abstraction $X := Y$, components of Y may also take one of the following two forms:

- (a) $Y_i \setminus L$ where Y_i is of the form described above, and where L is a Prolog list of the form $[a_1, a_2, \dots, a_n]$. The semantics of such an expression is that all events in the list L which form part of the trace of Y_i are hidden from the environment of Y_i and should therefore not appear explicitly as events in a trace of X .

- (b) $Y_i:s$ where Y_i is a Prolog variable and s is a Prolog atom. In this case the trace generator expects a Prolog fact in the input of the form $\text{sort}(s, \text{List})$ where List is a Prolog list representing sort values which the variable Y_i may assume. The syntactically correct usage of this form is as follows:

$$Y_i: s \rightarrow f(\dots, Y_i, \dots).$$

where f is any functor with Y_i as one of its arguments. This notation is intended to abbreviate the CSP expression: $a_1 \rightarrow f(\dots, a_1, \dots) \mid a_2 \rightarrow f(\dots, a_2, \dots) \mid \dots \mid a_n \rightarrow f(\dots, a_n, \dots)$ where $a_1, \dots, a_n$ are the elements of the list appearing in the Prolog fact $\text{sort}(s, \text{List})$.

Further translation considerations

Since all of the above representations are treated as Prolog facts embedded within the Prolog database, and since the database regards all text enclosed by `/*` and `*/` as comments, it is possible to add comments in this form to the input as required.

Note that in the implementation the Prolog operators, `':'` and `'/'` must be defined with lower precedence than those mentioned above, and that the operator `':='` must have a higher precedence.

Finally, if several process abstractions have names consisting of an identical functor but differing arguments, those process abstractions with variable arguments should be entered after those with non-variable arguments in order to prevent endless recursion in the Prolog searches. For example, if the following process abstractions are entered in the given order, then the latter will procedurally be interpreted as referring to values of X which are not equal to 1.

```
f(1) := a --> b.
f(X) := p --> f(X-1).
```

By reversing the order, no restriction is procedurally implied on the value of X , and the translation rules given above make no provision for any explicit declarative restriction. Such an ordering will therefore imply a trace consisting of an infinite number of ps for, say, $f(2)$. In practice, the Prolog interpreter will exhaust available memory and fail.

THE CORE APPROACH

The central idea in the trace generation program is to use a Prolog rule of the form

```
analyse (Expression, Trace) :-
    analyse_process (Expression, Trace);
    analyse_hiding (Expression, Trace);
    analyse_interleaving (Expression, Trace);
    analyse_concurrent (Expression, Trace);
    analyse_interrupt (Expression, Trace);
    analyse_gen_choice (Expression, Trace);
```

```

analyse_or (Expression, Trace);
analyse_choice (Expression, Trace);
analyse_set (Expression, Trace);
analyse_successor (Expression, Trace);
analyse_then (Expression, Trace).
analyse_event (Expression, Trace).

```

where each of the alternative goals in the body of this rule attempts to produce a Prolog list in the second argument (Trace) representing a trace for the instantiated value of the first argument (Expression). Only one of the alternative goals will succeed, and success depends upon the instantiated form of Expression when the analyse rule is invoked. (A third argument—not shown above—is also used in analyse. It is used to deal with recursive definitions and will be discussed below.) It is a procedural requirement that the analyse_process rule is attempted first, and that the analyse_event rule is attempted last, after all else has failed.

A simplified version of analyse_process may be defined as

```

analyse_process(X, Traces) :-
    X := Y,
    analyse (Y, Traces).

```

i.e. a process abstraction is sought in the database with a name that matches X, and if found, Y is instantiated to its behaviour expression part. An attempt is then made to analyse the behaviour expression, Y. Note, however, that this simplified version of the rule does not cater for the possibility that X may be a structure with instantiated arguments which may require some evaluation in Y. For example, if the behaviour expressions

```

f(N) := a --> g(N+1).
g(2) := b --> .....

```

appeared in the database, and analyse_process(f(1), Traces) was invoked, then proof of the goal f(1) := Y would result in Y being instantiated to a --> g(1+1), and a trace of [a.g(1+1)] would result instead of [a.b,.....]. Extensions to analyse_process, involving *inter alia* an alternative rule, are implemented in the trace generator. This allows for the evaluation of valid Prolog arithmetic expressions in the arguments of structures by invoking a rule called find_args.

The analyse_hiding rule is as follows:

```

analyse_hiding (X\C, Traces) :-
    analyse(X, Xtraces),
    remove(C, Xtraces, Traces).

```

where remove is a utility procedure which removes all elements of the list C from the list Xtraces to produce the required list, Traces. Note that if the hiding operator, \, was not the main functor of the first argument of the invoking analyse rule, then analyse_hiding could not succeed.

The rule for `analyse_interleaving` is simply

```
analyse_interleaving( X ||| Y, Traces) :-
    analyse(X, Xtraces),
    analyse(Y, Ytraces),
    interleave(Xtraces, Ytraces, Traces).
```

where an `interleave` rule is used to interleave the first two of its list arguments, producing a third interleaved list.

Rules for analysing expressions of the form $X \rightarrow Y$ and $X \parallel Y$ differ from the above only in that `interrupt` and `concurrent` list utility procedures are invoked, respectively, instead of the `interleave` procedure. This produces the required trace for expressions containing the `interrupt` operator, but involves a simplification of the theory in the case of the `concurrency` operator.

This is due to the fact that the present approach has avoided explicit representation of the so-called alphabet of a process. Of the operators implemented, such representation is only really relevant for the `concurrency` operator, since what has previously been termed the set of synchronization events for the behaviour expression $X \parallel Y$ is more precisely defined as the intersection of the so-called alphabets of X and Y . (The alphabet of a process X is the set of all events which the environment may potentially offer X .) In the absence of explicit knowledge of these alphabets, the intersection of `Xtraces` and `Ytraces` is used as the set of synchronization events in the rule:

```
analyse_concurrent ( X || Y, Traces) :-
    analyse (X, Xtraces),
    analyse (Y, Ytraces),
    concurrent (Xtraces, Ytraces, Traces).
```

Since a given invocation of `analyse(X, Xtraces)` does not necessarily produce an instantiation of `Xtraces` containing all elements of the alphabet of X (and similarly for `Ytraces`), it is clear that the intersection of `Xtraces` and `Ytraces` may turn out to be a proper subset of the set of synchronization events. This potential problem can be easily overcome if the alphabet of each process is given explicitly. This would, however, increase the complexity of the input requirements for the trace generator, and has therefore been avoided.

It should also be noted that the `concurrent` rule must cater for traces which end up in deadlock by virtue of the processes wishing to interact with different synchronizing events. Such deadlock is also indicated when one process is ready to synchronize, while the other has already stopped. The `concurrent` rule caters for these eventualities, subject to the aforementioned simplification.

The following rule for the procedure `analyse_choice` ensures that all traces generated by both X and Y are obtained:

```
analyse_choice (X | Y, Traces) :-
    analyse(X, Traces) ; analyse(Y, Traces).
```

Note that for technical reasons it is syntactically incorrect for either X or Y to be simply a process instantiation, rather than a so-called guarded expression of the form $a \rightarrow P$.²

Furthermore, it is syntactically incorrect to use the choice operator to express non-determinism by an expression such as $a \rightarrow P \mid a \rightarrow Q$. The trace generator is indifferent to such syntax violations, and its input should be screened by a syntax checker.

The rules for general choice and non-deterministic choice have bodies which are identical to that of the `analyse_choice` rule. Their heads are simply `analyse_gen_choice (X <> Y, Traces)` and `analyse_or (X | Y, Traces)`, respectively. The difference between these two operators lies in their respective refusal sets, rather than in their trace sets, as will be discussed below.

The following two rules are required for the successor operator, the first of which checks to see if the last element of the traces of X is the deadlocked process stop. If so the trace of $X;Y$ is the trace of X . If this is not the case, then a trace of $X;Y$ is a trace of X prefixed to a trace of Y , as expressed in the second rule:

```
analyse_successor(X;Y, Traces) :-
    analyse(X, Traces),
    last_elem (Traces, stop).

analyse_successor(X;Y, Traces) :-
    analyse(X, Xtraces),
    not last_elem(Xtraces, stop),
    analyse(Y, Ytraces),
    concatenate (Xtraces, Ytraces, Traces).
```

The rule which finds traces for expressions of the form $X:B \rightarrow Y$ checks the Prolog database for a fact of the form `Sort(B,List)`. Each invocation of this rule (both initially, and upon backtracking) finds a different element of `List` and instantiates X with this value, as well as any expressions containing X which appear in the behaviour expression Y . These latter expressions are evaluated using a `find_arg` rule, which is similar to that used in `analyse_process`, and which produces an evaluated form of Y called `NewY`. The rule for `analyse_then` is subsequently invoked with first argument $X \rightarrow \text{NewY}$. The trace produced by this invocation is a possible trace for $X:B \rightarrow Y$. An explicit form for this rule is therefore

```
analyse_set(X:B -> Y, Traces) :-
    sort(B,List),
    element(X, List),
    find_arg(Y, NewY),
    analyse(X -> NewY,Traces).
```

Elements of the list, `Trace`, finally become instantiated when the rule for analysing the then operator, `->`, and/or the rule for analysing events are invoked. The relevant rules are essentially given by

```
analyse_then (X -> Y, [X|Trace]) :-
    analyse(Y,Trace,ProcList).
```

and

```
analyse_event (X, [X]).
```

These clauses have been slightly simplified, and in the actual implementation checks are first made to ensure that the variable X has not been instantiated to a structure having one of the CSP operators as main functor. (This is done to prevent illegal traces being generated upon backtracking.) If this is proved, then X is assumed to be an atom, and becomes the first element in the list giving the traces of the behaviour expression: $X \rightarrow Y$ (as specified by the `analyse_then` rule), or X becomes the only element of the list of traces for itself in the `analyse_event` rule.

PERIPHERAL DESIGN ISSUES

Generating all traces

Within the limits already mentioned, the implementation provides for generating all traces of a process instantiation. This is done by using the well-known Prolog backtracking feature of failing the goal which invoked the `analyse` clause described above. (Of course, before failing the trace is written onto screen or file.) This results in an alternative trace being sought or, rather, in an alternative route through the proof procedure.

However, it is sometimes possible that various routes through the proof procedure produce the same trace list. A trivial example illustrates the point. Suppose the following process abstraction is given as input:

$$P := (a \rightarrow b \rightarrow \text{stop}) \parallel (b \rightarrow \text{stop})$$

Clearly the trace list $[a,b,b]$ may be derived in more than one way.

This may or may not be useful information for the user of the trace generator. If not, recurring traces may be suppressed along the following lines, where each new trace list not previously generated is dynamically recorded in the Prolog database:

```
traces(X) :- analyse_process(X, Traces),
             not_found(generated(Traces)),
             print_out(Traces),
             fail.
traces(X).
```

A first rule for `not_found(Fact)` tests whether `Fact` is in the database and fails if it is after freezing any other alternatives to satisfy `not_found(Fact)`. This results in a new invocation of the `analyse_process`. A second `not_found(Fact)` rule of lower priority calls the `assertz(Fact)` built-in predicate and succeeds, leading to a printout of the resulting trace.

Note that some Prolog implementations (but not Waterloo Core Prolog, Version 1.7) have built-in predicates, such as `setof` and `findall`, which accomplish much the same as discussed above. If `setof` is available and if X is instantiated, then a call to

```
setof (Traces, analyse_process(X, Traces), Tracelist)
```

will instantiate Tracelist to an ordered list of non-repeated traces of the process X. Similarly, a call to

```
findall (Traces, analyse_process(X, Traces), Tracelist)
```

will instantiate Tracelist to a list of all traces of the process X. Bratko⁸ gives the code for findall, which can be used if it is not available as a built-in predicate.

Coping with recursion

One of the desirable features of CSP-like specification languages is their recursive ability. However, it is clear that the trace generator described thus far will simply go into an endless search if requested to generate traces of the process abstraction $a := b \rightarrow a$.

In order to prevent this, a third argument is added to the analyse clause, as well as to all clauses which it may directly invoke. This argument is a list of the process names which have been analysed at any given point in the trace generation. It thus starts off as the empty list, (i.e. in the clause for traces(X) described above, analyse_process has [] as its third argument) and each time the rules for analyse_process have been successful, the first argument of the head (i.e. the process name) is prefixed to the process list. However, an additional rule for analyse_process (of higher precedence than the already described rules) is now necessary to test whether the process to be analysed already appears in the process list. If so, the process name is provided with a suffix of '*' to indicate recursion, this name is added as the last element of the trace list, the trace generation process terminates successfully and further considerations of traces along this route are frozen using the cut operator. Hence, for the trivial recursive example given above, the generator produces the trace-list [b,a*].

Note, however, that in order to enhance the comprehensibility of the output, the process names used in the trace generation may also be inserted into the trace list. It is convenient to distinguish these elements of the list from events in the list in some way, for example by inserting them as single-element lists. This is provided for as part of the analyse_process rules. In this case, the trace list for the aforementioned example becomes [[a],b,[a*]]. Hence it is possible to print the resulting trace in a more meaningful format, for example

```
-----> entering process a
b
-----> entering process a*
```

USAGE TO DATE AND INTENDED USAGE

To date, the implemented trace generator has been used in the specification phase of a small but non-trivial data communications project involving the addition of processes to an existing XXX PAD. An initial attempt at defining the relevant process interactions in CSP was incorporated into the functional specifications document drawn up for the client.

Early versions of the trace generator became functional at about this time, and the CSP definition was used to test both the generator and the definition itself. It was

found that the ability to inspect the resulting set of traces enforced several revisions of the system definition, since in several cases a particular definition attempt resulted in unforeseen traces which did not accurately correspond to what was actually required of the system. This interaction between defining, tracing and redefining eventually resulted in a provisional definition of the required system, which was used as a firm basis for implementation. However, it is envisaged that the provisional CSP system definition will undergo one last revision once the implementation phase has been completed. This is due to the inevitable minor changes which only become apparent during implementation, as well as to minor changes in the requirements expressed by the client.

The intention is to use the generated traces as a basis for drawing up test scenarios during the testing phase. Since event names in the CSP definition have been chosen to clearly differentiate between expected input and output events, it might even be possible to automate this testing process as follows. The traces can easily be written to a file which would serve as input to software processes accessing the modified PAD. These processes will then use the trace data to decide what can be sent to the PAD, and what response should be expected in each case. Unexpected responses (i.e. not provided for in the trace data) will indicate a potentially incorrect implementation. Whether this conceptual approach is actually feasible remains to be seen.

Other smaller CSP definitions have also been traced by a user wishing to become acquainted with CSP. It was clear that the ability to view traces of processes which had been defined made a valuable contribution to the learning process.

EXTENSION POSSIBILITIES

LOTOS trace generation

The trace generator has been written as a pilot project and forms part of a larger ongoing project to develop tools for verifying LOTOS specifications. At present, a LOTOS syntax checker (coded in PL/1) is available, providing error messages and a listing file for a LOTOS specification. It seems feasible that this software could relatively easily be enhanced to additionally provide a translator which translates syntactically correct LOTOS definitions into Prolog facts and rules.

The LOTOS syntax appears to lend itself admirably to such an approach. For example, the ACT ONE data abstraction part of LOTOS (based on the work of Ehrig *et al.*⁹) provides for complete and formal definitions of 'sorts' and 'operations' on sorts. Hence, for example, if the so-called action prefix expression $a?x:s1;PROC[GATE](x)$ appears in a LOTOS specification, then it is syntactically required by LOTOS that the elements of the sort $s1$, (say q, r, s) be defined in the type declaration part of the LOTOS specification. When the syntax checker scans the type declarations there is, in principle, no reason why it should not be able to write a corresponding Prolog fact into a file, such as $sort(s1, [q,r,s])$. Similarly, the syntax checker could appropriately do conversions from capital to small letters and vice versa, as required by Prolog for the representation of atoms and variables. Hence, the above action prefix expression would be changed to $a?X:s1;proc[gate](X)$. Such an expression can then be treated in the same way that the CSP trace generator treats the CSP expression $X:s1 \rightarrow proc(X)$.

It also seems feasible to use the operations and equations parts of the ACT ONE 'type' definitions in LOTOS to construct Prolog clauses which 'implement' the oper-

ations. Further study is required to work out the details of such a scheme. However, evaluation of a boolean expression can already be done relatively easily with only slight modifications to the existing trace generator, and provided that the format of this boolean expression corresponds to Prolog syntax. For example, the LOTOS guarded expression

$$[z = q] \rightarrow \text{PROC}[\text{GATE}] (z).$$

could be translated to the following Prolog fact:

$$(Z = q) \rightarrow \text{proc}[\text{gate}] (Z).$$

where $\rightarrow$ has been appropriately defined as a Prolog operator. Traces for this expression can be found, using the following rule:

```
analyse_guard(X -> Y, Traces) :-
    X,
    analyse(Y, Traces).
```

This is possible because the expression $(Z = q)$ is a valid Prolog goal. Of course, in an actual LOTOS specification the above guarded expression usually appears in a context where the variable Z has been instantiated to some element of a sort. If this instantiation was, for example q , then the metavariable X in the body of the `analyse_guard` rule would be instantiated to $(q = q)$, which is a valid Prolog goal that would succeed; other instantiations of the variable Z would result in failure of the meta-variable X .

Efficiency enhancements

Being a pilot project, not much attention was paid to questions of space and time efficiency during coding. Hence, even though the trace generator was able to cope with a non-trivial CSP definition, it is likely to have difficulty generating traces for a definition of, say, the OSI transport level. Many well-known strategies could be explored to generate improvements in this regard. These include the judicious use of the 'cut' built-in predicate, as well as the use of so-called difference lists.⁸ Of course, compiled Prolog code (not available under Waterloo Core Prolog) would improve time efficiency, and many Prolog compilers are designed to recover space on right recursion.

Alternative interrogation possibilities

Although the initial effort was put into generating all traces, the basic Prolog code is sufficiently versatile to ask several other questions without major changes, except perhaps at the user interface. For example, the generator may, with minimal effort be adapted to

- (a) establish whether a given list is a valid trace of a given process
- (b) find all trace lists which contain a given trace list as a sublist
- (c) establish whether two given processes generate the same traces
- (d) establish whether a trace generated by one process will be accepted by another if a given subset of messages is hidden from the second process
- (e) etc.

This flexibility is typical of Prolog programs, and is in fact one of the attractive features of the language.

CONCLUSION

Experience to date with the trace generator has led to the following conclusions:

1. Coding the generator in Prolog proved to be surprisingly easy. The language lends itself to code development in a top-down fashion, with initial emphasis on the declarative rather than procedural aspects of the task at hand. The resulting code tends to be almost self-documenting in many parts. For example, the rule for dealing with the concurrency operator:

```
analyse_concurrent (X || Y, Traces) :-
    analyse (X, Xtraces),
    analyse (Y, Ytraces),
    concurrent (Xtraces, Ytraces, Traces).
```

may be considered as a high-level statement of what is to be done, and the details of how to deal with the two traces (Xtrace and Ytrace) to obtain Traces is left until later.

2. The fact that Prolog questions may be asked of each part of the code at any stage of the software development greatly facilitates testing during development. Indeed, unlike conventional software development where testing forms more or less a distinct phase in the software cycle, in the case of developing Prolog code, testing tends to be much more interwoven with the actual coding effort. For example, once the rule for

```
concurrent (Xtraces, Ytraces, Traces)
```

is written down, the rule may be tested with various trial instantiations for Xtraces and Ytraces, until satisfaction is obtained that the rule is correct. This tends to enhance the reliability of the code.

3. The fact that CSP specifications can be so conveniently translated into Prolog facts is considered to be a particular advantage of the generator. This was made possible by the judicious definition of Prolog operators.
4. The project described above is already proving its practical value, both for teaching CSP and for generating traces for test purposes. The evidence suggests that it would be both feasible and worth while to extend the above ideas towards a trace generator for LOTOS specifications.

REFERENCES

1. 'Draft proposal of Estelle', *ISO/TC 97/SG 21/DP 9074*, June 1985.
2. 'Information processing systems—open systems interconnection—LOTOS—a formal description technique based on temporal ordering of observational behaviour', *ISO/TC 97/SG 21/DP 8807*, March 1985.
3. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985.
4. R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag, 1980.
5. H. Ural and R. L. Probert, 'Step-wise validation of communication protocols and services', *Computer Networks and ISDN Systems*, **11**, 183–202 (1986).

6. D. Gilbert, 'LOTOS to Parlog translator in Parlog', *Demonstration given at 3rd International Conference on Logic Programming*, Imperial College London, July 1986.
7. L. Van der Vegte, 'A LOTOS syntax checker', Paper presented at South African Computer Science Lecturers' Association, July 1986.
8. I. Bratko, *Prolog programming for Artificial Intelligence*, Addison-Wesley, 1986.
9. H. Ehrig, W. Fey and H. Hansan, 'ACT ONE: an algebraic specification language with two levels of semantics', *Bericht-Nr. 83-03*, Technical University of Berlin, 1983.