PROCEEDINGS

Southern African Computer Symposium 1991

6th de Rekenaarsimposium van Suider Afrika 1991

DE OVERBERGER HOTEL, CALEDON

2 - 3 JULY 1991

SPONSORED BY

ISM

FRD

GENMIN

EDITED BY

M H Linck

DEPARTMENT OF COMPUTER SCIENCE • UNIVERSITY OF CAPE TOWN

PROCEEDINGS / KONGRESOPSOMMINGS

6th
SOUTHERN AFRICAN COMPUTER SYMPOSIUM

6de
SUIDELIKE-AFRIKAANSE REKENAARSIMPOSIUM

De Overberger Hotel, Caledon

2 - 3 JULY 1991

SPONSORED by

ISM
FRD
GENMIN

EDITED by

M H LINCK

Department of Computer Science

University of Cape Town

TABLE OF CONTENTS

FOREWORD

The 6th Computer Symposium, organised under the auspices of SAICS, carries on the tradition of providing an opportunity for the South African scientific computing community to present research material to their peers.

It was heartening that 31 papers were offered for consideration. As before all these papers were refereed. Thereafter a selection committee chose 21 for presentation at the Symposium.

Several new dimensions are present in the 1991 symposium:

* The Symposium has been arranged for the day immediately after the SACLA conference.

* It is being run over only 1 day in contrast to the 2-3 days of previous symposia.

* I believe that it is first time that a Symposium has been held outside of the Transvaal.

* Over 85 people will be attending. Nearly all will have attended both events.

* A Sponsorship package for both SACLA and the Research Symposium was obtained. (This led to reduced hotel costs compared to previous symposia)

A major expense is the production of the Proceedings of the Symposium. To ensure financial soundness authors have had to pay the page charge of R20 per page.

A thought for the future would be consideration of a poster session at the Symposium. This could provide an alternative approach to presenting ideas or work.

I would sincerely hope that the twinning of SACLA and the Research Symposium is considered successful enough for this combination survive. As to whether a Research Symposium should be run each year after SACLA, or only every second year, is a matter of need and taste.

A challenge for the future is to encourage an even greater number of MSc & PhD students to attend the Symposium. Unlike this year, I would recommend that they be accommodated at the same cost as everyone else. Only if it is financially necessary should the sponsored number of students be limited.

I would like to thank the other members of the organising committee and my colleagues at UCT for all the help that they have given me. A special word of thanks goes to Prof. Pieter Kritzinger who has provided me with invaluable help and ideas throughout the organisation of this 6th Research Symposium.

M H Linck
Symposium Chairman

SYMPOSIUM CHAIRMAN

M H Linck, University of Cape Town

ORGANISING COMMITTEE

D Kourie, Pretoria University.

P S Kritzinger, University of Cape Town.

M H Linck, University of Capè Town.

SPONSORS

ISM

GENMIN

FRD

LIST OF REFEREES FOR 6th RESEARCH SYMPOSIUM

NAME	INSTITUTION
Barnard, E	Pretoria
Becker, Ronnie	UCT
Berman S	UCT
Bishop, Judy	Wits
Berman, Sonia	UCT
Brink, Chris	UCT
Bodde, Ryn	Networks Systems
Bornman, Chris	UNISA
Bruwer, Piet	UOFS
Cherenack, Paul	UCT
Cook Donald	UCT
de Jaeger, Gerhard	UCT
de Villiers, Pieter	Stellenbosch
Ehlers, Elize	RAU
Eloff, Jan	RAU
Finnie, Gavin	Natal
Gaynor, N	AECI
Hutchinson, Andrew	UCT
Jourdan, D	Pretoria
Kourie Derrick	Pretoria
Kritzinger, Pieter	UCT
Krzesinski, Tony	Stellenbosch
Laing, Doug	ISM
Labuschagne, Willem	UNISA
Levy, Dave	Natal

MacGregor, Ken	UCT
Machanick, Philip	Wits
Mattison Keith	UCT
Messerschmidt, Hans	UOFS
Mutch, Laurie	Shell
Neishlos, N	Wits
Oosthuizen, Deon	Pretoria
Peters Joseph	Simon Fraser
Ram, V	Natal, Pmb.
Postma, Stef	Natal, Pmb
Rennhackkamp, Martin	Stellenbosch
Shochot, John	Wits
Silverberg, Roger	Council for Mineral Technology
Smit, Riël	UCT
Smith, Dereck	UCT
Terry, Pat	Rhodes
van den Heever, Roelf	UP
van Zijl, Lynette	Stellenbosch
Venter, Herman	Fort Hare
Victor, Herna	Stellenbosch
von Solms, Basie	RAU
Wagenaar, M	UPE
Wentworth, Peter	Rhodes
Wheeler, Graham	UCT
Wood, Peter	UCT

6TH RESEARCH SYMPOSIUM - 1991

FINAL PROGRAM

TUESDAY 2nd July 1991

10h00 - 13h00 **Registration**

13h00 - 13h50 **PUB LUNCH**

14h00 - 15h30 SESSION 1A

Venue: Hassner

Chairman: Prof Basie von Solms

14h00 - 14h30
"A value can belong to many types. "
B H Venter, University of Fort Hare

14h30 - 15h00
"A Transputer Based Embedded Controller Development System"
M R Webster, R G Harley, D C Levy & D R Woodward, University of Natal

15h00 - 15h30
"Improving a Control and Sequencing Language"
G Smit and C Fair, University of Cape Town

SESSION 1B

Venue: Hassner C

Chairman:Prof Roelf v d Heever

14h00 - 14h30
"Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers" P Machanick, University of Witwatersrand

14h30 - 15h00
"Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface" L van Zijl & D Mitton, University of Stellenbosch

15h00 - 15h30
"Product Form Solutions for Multiserver Centres with Heirarchical Classes of Customers" A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig

15h30 - 16h00 **TEA**

16h00 - 17h30 SESSION 2A

Venue: Hassner

Chairman: Prof Derrick Kourie

16h00 - 16h30
"A Reusable Kernel for the Development of Control Software" W Fouché and P de Villiers, University of Stellenbosch

16h30 - 17h00
"An Implementation of Linda Tuple Space under the Helios Operating System" P G Clayton, E P Wentworth, G C Wells and F de-Heer-Menlah, Rhodes University

17h00 - 17h30
"The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System" K MacGregor, University of Cape Town & R Campbell, University of Illinois at Urbana-Champaign

19h30 **PRE-DINNER DRINKS**

20h00 **GALA CAPE DINNER**
(Men: Jackets & ties)

WEDNESDAY 3rd July 1991

7h00 - 8h15 BREAKFAST

8h15 - 9h45 SESSION 3A

Venue: Hassner

Chairman: Assoc Prof P Wood

8h15 - 8h45
"Concurrency Control Mechanisms for Multidatabase Systems" A Deacon, University of Stellenbosch

8h45 - 9h15
"Extending Local Recovery Techniques for Distributed Databases" H L Victor & M H Rennhackkamp, University of Stellenbosch

9h15 - 9h45
"Analysing Routing Strategies in Sporadic Networks" S Melville, University of Natal

SESSION 3B

Venue: Hassner C

Chairman: Prof G Finnie

8h15 - 8h45
The Design of a Speech Synthesis System for Afrikaans" M J Wagener, University of Port Elizabeth

8h45 - 9h15
"Expert Systems for Management Control: A Multiexpert Architecture" V Ram, University of Natal

9h15 - 9h45
"Integrating Simularity-Based and Explanation-Based Learning" G D Oosthuizen and C Avenant, University of Pretoria

9h45 - 10h15 TEA

10h15 - 11h00 SESSION 4

Venue: Hassner

Chairman: Prof P S Kritzinger
Invited paper: E Coffman

11h00 - 11h10 BREAK

11h10 - 12h40 SESSION 5A

Venue: Hassner

Chairman: Prof C Bornman

11h10 - 11h40
"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town

11h40 - 12h10
"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch

12h10 - 12h40
"Building a secure database using self-protecting objects" M Olivier and S H von Solms, Rand Afrikaans University

SESSION 5B

Venue: Hassner C

Chairman: Prof A Krzesinski

11h10 - 11h40
"Modelling the Algebra of Weakest Preconditions"
C Brink & I Rewitsky, University of Cape Town

11h40 - 12h10
"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch

12h10 - 12h40
"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal

12h45-12h55 GENERAL MEETING of RESEARCH SYMPOSIUM ATTENDEES

Venue: Hassner

Chairman: Dr M H Linck

13h00 - 14h00 LUNCH

FINIS 6th COMPUTER SYMPOSIUM

PAPERS

of the

6TH RESEARCH SYMPOSIUM

Improving A Control and Sequencing Language

Colin A. Fair* and Gabriël D Smit
Laboratory for Advanced Computing
Department of Computer Science
University of Cape Town

May 1991

Abstract

In a process control environment, batch processes, as opposed to continuous processes, are characterised by multi-product manufacturing lines producing relatively small quantities which often involves frequent product changes. One component of batch control systems is a programming language which is used to control and synchronise the operations of the plant. Initially low-level languages (e.g. ladder logic, boolean algebra and assembly language) were used, but have now been replaced by specialised high-level languages. These languages provide more functionality and are easier to use. This paper examines one such a high-level sequencing language (CASL) and identifies functionality, clarity and readability improvements that can be made to the existing language. An implementation of an upwardly compatible compiler for the improved language is described briefly.

1 Introduction

Batch processing provides manufacturers with the flexibility of producing many different products using the same basic equipment, compared to continuous processes which are used whenever large quantities of a product are required and the specification remains relatively constant [8]. The recipes (or sequences) for producing these batch products need to be updated continually in order to maximise the use of raw materials and plant equipment, thus producing cost-effective products. Batch process control has presented unique challenges to the process control fraternity [7, 8, 4, 10] compared to continuous control which is generally well understood and easier to implement [8].

Many low-level and unsophisticated systems have evolved over the years that allow the computer to automatically control a factory or industrial plant [6] and [4]. These systems were often difficult to understand and implement and the need arose for a high level language with concepts and constructs that could map onto typical plant operations. CASL[1] (Control And Sequencing Language) [3], is such a special-purpose language that is used for programming the various functions associated with batch process control. CASL forms part of the CYGNUS [2] process management and control system [2] and was developed in the seventies by ICI (Ltd) in England.

*Partially supported by the Computer Science Development Programme of the FRD
[1] Pronounced, "castle" or "cassel"
[2] Registered trade mark of AECI Limited

On current implementations, CASL source code is compiled into an intermediate language. The intermediate code is loaded into a code partition in memory from where it is executed by a run-time interpreter. Loading and replacement of code sequences are allowed during run-time. The interpreter interacts with the plant equipment by reading and/or setting attributes of plant equipment as represented in a real-time database. A separate task, the *scanner* is responsible for the communication between the plant equipment and the real-time database.

As will be seen, CASL is a somewhat archaic language. The distributor of CYGNUS, AECI Process Computing, approached the Department of Computer Science at the University of Cape Town with a view to developing a local CASL compiler, while at the same time modernizing the language where possible while still maintaining upward compatibility.

In this paper we examine and evaluate CASL and identify weaknesses in its functionality, readability and module clarity; we design improvements that are, as far as possible, upwardly compatible with the old language; and we discuss some issues that arose during implementation of a compiler for the new language using compiler construction tools such as *lex* and *yacc*.

2 CASL Overview

This section introduces the concepts and major components of CASL. It is not intended as a language tutorial, but rather to provide the reader with some idea of the nature of the language.

2.1 CASL Operation

In a process control plant several operations (or processes) typically take place at one time. A batch process is defined by Rubin [10] as "having a recognizable start and finish, with one or more states or phases occurring during the operation." In CASL, these operations are represented by *jobs* (similar to tasks) which are executed (pseudo-)concurrently. A CASL job can be described as the execution of a *sequence*. A *sequence* is similar to a procedure in a conventional high level language and consists of a sequence or block of executable CASL statements. In addition to being CALLed (like a procedure call), a sequence may also be STARTed, in which case a new job is created. A job may consist of a number of sequences because of possibly nested CALLs to other sequences. However, there is only one active sequence per job (see Figure 1).

Sequences are re-entrant, therefore more than one job can access the same sequence simultaneously. A sequence is subdivided into *steps* by certain CASL statements. These steps play an important role in the concurrent execution of jobs. While it is the responsibility of the process engineer to subdivide a sequence into steps, in practice these steps occur naturally due to the distribution of the relevant CASL statements in the sequence code.

Pseudo concurrency is obtained by executing part of every job at least once in a given time interval (e.g. once per second). This concept is called a *heartbeat*. The part of a job that is executed during a single heartbeat is called a *phase* and corresponds with the execution of a single step in a sequence of code. In other words, one phase of every job is executed every heartbeat. For example, Figure 1 illustrates the conceptual structure of a CASL system being executed in a batch plant. There are n jobs in the plant (CASL

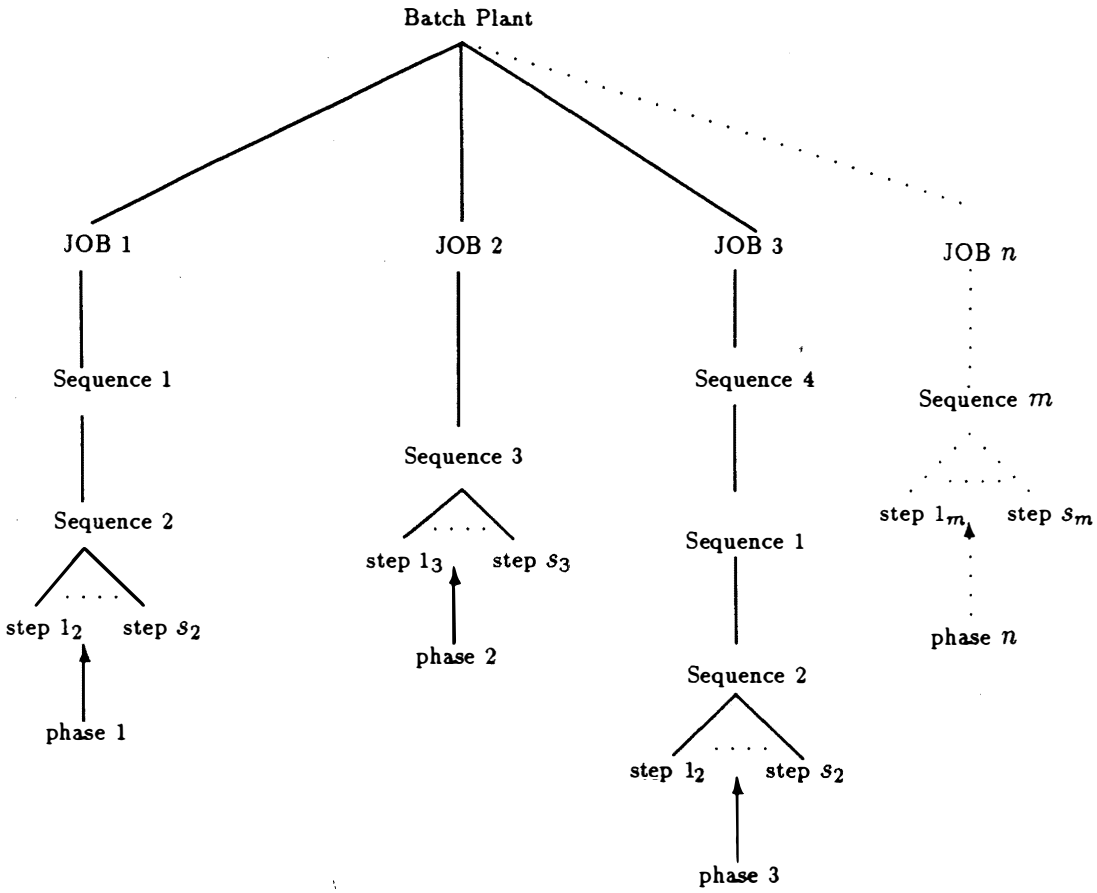

Figure 1: Hierarchical plant execution diagram

allows a maximum of 60), each with one or more sequences forming part to it. A new job is created when a sequence is STARTed, and this sequence is known as the base sequence. A job may consist of more than one sequence due to nested CALLs to other sequences, e.g. JOB 3 consists of Sequence 4 (the base sequence) which CALLed Sequence 1, which in turn CALLed Sequence 2. Note that Sequence 2 is accessed (being executed) by JOB 1 and JOB 3. In a single heartbeat n phases will be executed, e.g. in phase 1 step i_2 of Sequence 2 might be executed as part of JOB 1, step j_3 of Sequence 3 as part of JOB 2 in phase 2, step k_2 of Sequence 2 as part of JOB 3 in phase 3, etc.

Phases are not preempted, but run to completion. Mutual exclusion is therefore guaranteed within a step. Synchronisation and scheduling of jobs are obtained with special CASL statements that provide the following operations:

- Start a new job.
- Delay the current job for a given time period.
- Place a given job on hold. The held job could be the current or any other job, and it may be put on hold either immediately or when it reaches a specified point (label) in its sequence.
- Reactivate a given job that has been put on hold.
- Terminate a given job.

The data on which sequence logic operates can be separated into three parts: process database data; PLIST data; and job-local data. By accessing and/or changing attributes associated with plant addresses in the *process database*, the sequence logic is able to read the plant state and control plant equipment. A PLIST *data block* is a block of data (or parameter list) that have been grouped together and given a name. A PLIST can be passed as a parameter to a sequence, or can be referenced explicitly as a global data block. A limited number (only four) *job-local* variables are provided. They have fixed names (**X1** to **X4**) and are local to a job. A reference to such a variable from within any sequence will reference the corresponding variable in the job executing that sequence.

2.2 A CASL Module Definition

A CASL program is usually split into a number of modules. Each module consists of the six sections described next. Some of these sections are optional. An example of a module is given in Figure 2. It has the following components:

TITLE and LET definitions

The **TITLE** statement defines a module title, while **LET** definitions define text replacement strings (parameterless macros).

PLIST structure definitions

PLIST structure definitions appear in the **PLISTDEF** section and are like record definitions in a named type declarations in Pascal and *struc* definitions in a *typedef* in C. Every PLIST must have a unique name and contains a list of field definitions. Fields are declared to be of one of five types (or *modes* in CASL terminology), namely **INT** (integer), **NUM** (real), **BITS** (bit flags), **PLAD** (plant address), or **MESS** (message — actually string), and must have a name that is unique within the module. An array is defined by placing the size of the array after the mode keyword. For example

```
PLIST REACTOR = PLAD OUTLET, INLET, NUM(2) REQVOL, AREA, INT ERRFLAG
```

defines any **REACTOR** PLIST to consist of two plant addresses (**OUTLET** and **INLET**), two arrays of two real numbers each (**REQVOL** and **AREA**), and an integer (**ERRFLAG**).

PLIST instantiations

The section headed by the keyword **PLISTNUM** is used to declare the actual PLISTs (PLIST instantiations) for the module. The **PLISTNUM** section of Figure 2, for example, defines three PLISTs with the **REACTOR** structure, namely **REACTOR(1)**, **REACTOR(4)**, and **REACTOR(9)**, and one PLIST with the **BATCHREC** structure. Each instantiated PLIST is created with a list of variables that correspond to the fields defined in the PLIST-structure.

In addition to the instantiation number (the numbers in parentheses), each PLIST must be identified with a unique integer identifier. These identifiers are used by the interpreter as indices to reference the PLIST items in the data partition of the sequence database in memory. Each integer must be unique in a project in order to prevent a conflict from occurring during access. A maximum of 255 PLISTs may be declared in CASL.

```
TITLE "CASL/TST/18FEB90" EXAMPLE MODULE

LET SCREEN = OCT 100;              %OCP dialogue
LET NL = 10;                       %NEWLINE character for printouts

PLISTDEF                           %Plist definition section
PLIST REACTOR = PLAD OUTLET, INLET, NUM(2) REQVOL, AREA, INT ERRFLAG
PLIST BATCHREC = INT BATCHNUM, SHIFTNUM, MESS SEQTITLE, SEQID

PLISTNUM                           %Plist declaration(instantiation) section
1   :  REACTOR(1)
5   :  REACTOR(4)
13  :  REACTOR(9)
100:   BATCHREC(1)

SEQNUM                             %Sequence definition section
10:   MAINSEQ
20:   SUBSEQ REACTOR, BATCHREC
30:   ANOTHERSEQ BATCHREC

DATA                               %Plist initialisation section
REACTOR(4) = '2XV326, '2XV315, 50.0, 60.0, 0.0(2), 22
BATCHREC(1) = 0(2), "Transfer of raw materials for process 10", "X21208"

SEQUENCE MAINSEQ
10:
      IDENT1 "Mainseq"
      DEVIN SCREEN
      DEVOUT SCREEN

20:   X1 = ME                      %Set local variable X1 to current job no.
      START SUBSEQ REACTOR(2), BATCHREC(1)
      GOTO 10
ENDSEQ

SEQUENCE SUBSEQ AREACTOR, ABATCH
      IDENT1 "Subseq"
      :
      ABATCH.BATCHNUM = 100
      OPEN AREACTOR.INLET
      :
ENDSEQ
```

Figure 2: A sample CASL module.

Sequence identification section

All sequences used in a module, whether they are defined in the module or just referenced in it, must be numbered and listed in the **SEQNUM** section. Each entry in the section consists of a number followed by a colon, the sequence name, and a list of PLIST-structures that defines the kind of parameter list(s) associated with the sequence. This is similar to procedure prototypes found in other languages. The integer identifiers are used by the interpreter as indices into the code partition of the sequence database. Our sample module identifies three sequences, one (**ANOTHERSEQ**) of which is external to the module. **MAINSEQ** does not make use of a PLIST, while **SUBSEQ** requires two PLISTs, namely a **REACTOR** type PLIST and a **BATCHREC** type PLIST.

Initialisation of PLISTs

The **DATA** section contains the optional initialisation of variables in instantiated PLISTs declared in the **PLISTNUM** section. The PLIST is identified by its name and instantiation number and is followed by a list of data values. The list of values must have a one to one correspondence with the appropriate PLIST-structure definition in the **PLISTDEF** section.

The list of data values may include a replication factor. This is specified as an integer in parentheses following a data item. For example, in Figure 2, the variables in the PLIST REACTOR(4) are initialised as follows:

```
REACTOR(4).OUTLET     = '2XV326      REACTOR(4).AREA(1) = 0.0
REACTOR(4).INLET      = '2XV315      REACTOR(4).AREA(2) = 0.0
REACTOR(4).REQVOL(1) = 50.0          REACTOR(4).ERRFLAG = 22
REACTOR(4).REQVOL(2) = 60.0
```

Sequence definitions

A sequence definition starts with the keyword **SEQUENCE** followed by the sequence name and a list of formal parameter names. These formal parameters refer to PLISTs of the kind defined in the **SEQNUM** section. The sequence definition is terminated with the keyword **ENDSEQ**. CASL labels and executable statements are enclosed between these two keywords.

Executable statements

CASL executable statements can be sub-divided into a number of functional groups. *Assignment* statements change the contents of PLIST and local variables. Assignments can be made to variables of any type except **MESS**. However, type conversion is only allowed between integers and reals. *Plant control* statements allow the manipulation of **PLAD**'s and their attributes in the database. For example, the value of an attribute may be changed with the SET statement. *Program execution control* statements direct the flow of control through a sequence. These include an IF THEN ELSE statement, a simple looping construct, a subroutine (sequence) call, as well as a **GOTO** statement. *Text input and output* statements read and write text data from and to the current input and output devices.

Job control statements can be used to affect the state and execute positions of not only the current job, but also other jobs. We have mentioned most of these statements under job synchronisation. A further example is the **DIVERT** statement which is similar to a **GOTO**, except that it causes another (specified) job to **GOTO** the given label. *Timing* and *error notification* statements form the last two groups of CASL statements.

3 CASL: A Critique

CASL can be evaluated as a batch process control language and as a high level language. In the former, one would ask how well does the language serve the process personnel in specifying control sequences that must be performed in the plant. In other words, how good is the mapping from CASL statements to plant operations? In the second evaluation one would consider such aspects as the clarity and simplicity of the language concepts, clarity of module syntax (i.e. readability), naturalness for the application, ease of module creation and use [9].

3.1 CASL as a Process Control Language

CASL seems to be a good batch control language. The abstraction of concurrency concepts namely a heartbeat and phase, allows the CASL programmer to concentrate on the content of a module rather than on concurrency. Re-entrancy of sequence code and the ability to pass PLISTs as parameters to a sequence maps well onto the batch processing environment. A particular sequence (recipe) may be defined and by simply passing a different PLIST (set of ingredients), a different batch of a products may be produced. This together with the instantiation of multiple PLIST definitions, the interface with an Operator Communication Package, and the standard interface to the plant database, all contribute to the success of CASL as a process control language — evidenced by the more than 60 installations of CASL in South Africa.

3.2 CASL as a High Level Language

When judged as a high level language, CASL has a number of deficiencies. In the following sections we examine these short-comings and propose solutions.

Some of the unfortunate features of CASL cannot be addressed without making the new language incompatible with the old. We first discuss those issues that can be addressed while maintaining upward compatibility and then point out those to which the solutions, we believe, have more serious implications.

Local Variables

CASL allows only four local (fixed-named, and typeless) variables per job. The main reason for this restriction was the severe memory restrictions the original system had to cater for. This is no longer the case and CASL can be enhanced with proper local variables.

Proper local variables can be introduced through a special kind of PLIST, namely a **LOCAL** PLIST. **LOCAL**s are declared at the start of a sequence, before the executable statements. The same format is followed as that of a PLIST declaration in the **PLISTDEF** section of the module. For example,

```
SEQUENCE MAINSEQ
LOCAL PLAD(16) locplad,
      INT      locint1, locint2, locint3,
      MESS     messloc
  :
X1 = locint1 + locint2 * locint3
  :
```

```
ENDSEQ
```

The same usage and naming convention applies as for normal PLISTs, except for the scope of the variables. While the names of local variables must be unique, their scope are limited to only the sequence in which they are defined. Therefore, a local variable in sequence A is not visible to sequence B in the same job, and neither is a local variable in sequence A of job J visible to sequence A in job K. This is achieved by ensuring that every time a sequence is invoked, an instantiation of the local PLIST is created. This is not unlike invocation records and stack frames in conventional programming languages.

Interface variables

CASL allows only PLISTs to be passed as parameters to sequences. It is not difficult to extend this to allow any variable to be passed as a parameter. We call these additional parameters *interface variables*. They may be used in combination with PLISTs. For example,

```
PLISTDEF PLIST SOMEPLIST = ...

PLISTNUM 1:  SOMEPLIST(1)

SEQNUM
10:  MAINSEQ
20:  SUBSEQ SOMEPLIST, (INT, PLAD)

SEQUENCE MAINSEQ
LOCAL INT COUNT
   :
   CALL SUBSEQ SOMEPLIST(1), (10, COUNT, 'HV-100)
   :
ENDSEQ

SEQUENCE SUBSEQ APLIST, (INT ivint1, count, PLAD valve)
   SET valve.MVE = 100
   APLIST.SOMEFIELD = ...
   :
ENDSEQ
```

Interface variables can be implemented as yet another type of PLIST – at each CALL or START an instantiation is made of an interface PLIST containing the values of the actual parameters. This instantiation of the interface plist is destroyed once the called sequence terminates. Interface variables are therefore passed by value.

Looping Constructs

CASL does not have any flexible looping construct apart from the GOTO statement and the REPEAT ... FINISH loop, which is restricted to a maximum of 255 iterations. One of the reasons for this is the fear that a loop can take long to complete (e.g. because of a large repeat count) and if it does not contain an end of phase statement, the job may overrun its time slice in the heartbeat. (A backwards GOTO forces an end of phase, therefore loops constructed from GOTOs do not have this problem.) In spite of this, it is our opinion that

more powerful looping constructs should be provided. Suitable programming practices such as putting conditional phase terminators inside loops can prevent overruns. As a compromise between simplicity and versatility, we argue for a **FOR** loop and a **WHILE** loop, namely

```
FOR <counter> = <initial_value> TO <final_value> STEP <step_value> DO
    ⋮
NEXT <counter>
```

and

```
WHILE <condition> DO
    ⋮
ENDWHILE
```

The semantics are similar to those in Pascal or Modula-2. The loop counter may be a PLIST item or a local variable. The **STEP** value is optional, with a default value of one. Loops may be nested.

Real Numbers

CASL supports a 16-bit NUM which stores a floating point number in a (non-standard) internal format. Values can range from 0.00003 to 65504.0 with an accuracy of one part in 2048. In order to obtain better accuracy and to make more efficient use of floating point co-processors where available, 32-bit IEEE format floating point numbers should be supported. Due to the current design of the intermediate language, this has the implication that all other basic data types must be expanded to 32-bit values.

Foreign Language Procedure Calls

CASL allows RTL/2 procedures to be called from within a sequence with the statement

```
RTL n
```

where n is an integer identifying the procedure to call. Parameters and results can only be passed through the 4 local variables. There is no reason why these procedures cannot be identified symbolically. Support should also be provided for other foreign languages such as C. Foreign language procedure call statements could therefore be defined, e.g.

```
XCALL C procedure_name
XCALL RTL procedure_name
```

Proper parameter passing as well as external *function* calls should be allowed as in

```
AREACTOR.AREA(1) = CALC_AREA(R_LENGTH, R_BREADTH)
```

Symbolic Labels

CASL allows only numeric labels in the range 0–255 since the original implementation required any label to be identifiable by a single byte. Since memory restrictions are no longer a big issue, the new language should allow the use of symbolic (alpha-numeric) names for labels.

Macros and File Inclusions

The LET text replacement facility allows for the definition of parameterless macros. Nested macros are not supported. Similarly files may be included with a file inclusion command, but only one level of inclusion is allowed. It is a relatively simple matter to expand LET definitions to allow for parameters and nested replacements and to allow nested file inclusions.

Boolean expressions

CASL does not support parenthesis to force precedence of operators in boolean expressions. This is unnecessary restrictive and simply reduces the readability of CASL programs.

All the issues discussed so far can be (and have been) addressed by implementing the proposed changes while still maintaining upward compatibility with the old language. Solutions to the issues in the following sections, however, will have more serious implications.

Layout and readability

CASL does not use an explicit statement terminator or separator like the semicolon in C and Pascal respectively. While CASL statements are always terminated by a newline character, not all newline characters terminate statements. One of the reasons given for not using explicit statement terminators [5] is that most users of CASL don't know programming languages and therefore statement terminators other than a newline would be "unnatural" for them and an unnecessary complication. We argue, however, that with the CASL syntax the way it is, the use of newline characters actually complicates matters: a set of rules is required specifying where newlines must, may, and may not be used. For example, newlines are allowed after any comma and before (but not after) ANDs and ORs in conditional expressions. It would have been much simpler to state newlines may be used anywhere where spaces can be used. It is somewhat embarrassing to have to explain to someone that she may write

```
IF  A.SIZE >= B.SIZE                    but not    IF  A.SIZE >= B.SIZE  AND
    AND A.SIZE >= 0   THEN                             A.SIZE >= 0       THEN
```

Had it not been for the syntax of two statements, namely the CALL and START statements, it might have been possible to allow free-format CASL programs without explicit statement terminators (while still using an LALR(1) parser). The problem lies in the fact that the two fragments

```
CALL CLEAN_UP TANK(1) = 0
```

and

```
CALL CLEAN_UP
TANK(1) = 0
```

are both legal (non-free-format) CASL fragments, but mean completely different things. In the former TANK is a PLIST with a single field that is initialised to 0 in the CALL. In the latter, TANK is an array variable (a member of a PLIST — if there is only one instantiation of a PLIST, the field name is enough to identify the variable) that is assigned the value 0.

Even without free-format the (poor) syntax of CALL and START statements give problems to an LALR(1) parser. As indicated before, PLISTs may be passed as parameters with these statements, and furthermore, they may be (optionally) initialised at that time. One could therefore have the following

```
PLIST REACTOR = PLAD OUTLET, INLET, NUM(2) REQVOL, AREA, INT ERRFLAG
PLIST BATCHREC = INT BATCHNUM, SHIFTNUM, MESS SEQTITLE, SEQID
   ⋮
CALL SUBSEQ REACTOR(1) = '2X, 'V3, 5.0, 6.0, 0.0(2), 22, BATCHREC(1)
```

which initialises the fields of **REACTOR(1)** to the same values of **REACTOR(2)** in the sample module of Figure 2. To further complicate matters, the initial values may be references to other PLIST fields, e.g.

```
PLIST BOB = INT(20) A
   ⋮
START X1 BOBSEQ BOB(1),BOB(2)=1,,3(5),BOB(4).A(2),(7), 10(5), BOB(4)
BOB(3).A = 10            % a statement following the call
```

The work of not only the parser, but also the human reader would be much easier if the initialisation of a PLIST is enclosed in braces ({ }) and repetition factors are enclosed in square brackets ([]). The above example could then be written as follows:

```
START X1 BOBSEQ BOB(1),
                BOB(2) = {1, , 3[5], BOB(4).A(2), [7], 10[5] },
                BOB(4)
BOB(3).A = 10            % a statement following the call
```

IF Statement

The IF statement has the structure

IF <*condition*> <*optional_then*> <*statements*> **END**

This allows for the following potentially confusing code

```
IF X1 = 1            % if X1 is equal to 1
   X4 = X1           % THEN assign X1 to X4
   PRINT X1          % etc.
   ⋮
END
```

The **THEN** should be mandatory.

LET Definition

Unlike other CASL statments, **LET** statements are terminated with an explicit terminator – a semi-colon. While this allows for definitions such as

```
LET CLEAR_FLAGS =
   X4 = 0
   REPEAT 70
      X4 = X4 + 1
      FLAG(X4) = 0
   FINISH;
```

readability can be improved if an explicit continuation character is used and no termination character (to fit in with the rest of the language), e.g.

```
LET CLEAR_FLAGS =  \
   X4 = 0          \
   REPEAT 70       \
      X4 = X4 + 1  \
      FLAG(X4) = 0 \
   FINISH
```

String Manipulation

CASL does not support string assignments directly. In order to assign a string to a (message) variable, one has to use a special version of the **WRITE** statement, e.g.

```
WRITE 255 MESSVAR, "New text string"
```

(Every **WRITE** statement has a priority which determines the urgency of the **WRITE** statement. The priority range is from 0-9. The **WRITE** statement used with the special priority of 255 indicates a string assignment.) It would be much simpler if one could write

```
MESSVAR = "New text string"
```

PLIST initialisation

PLIST instantiation and PLIST initialisation is performed by two separate statements (**PLISTNUM** and **DATA**). This is an unnecessary fragmentation of information and should rather be combined, for example

```
PLISTNUM
2 :  REACTOR(4) = '212XV326, '212XV315, 50.0, 60.0, 0.0(2), 22
10:  BATCHREC(1) = 0(2),
                   "Transfer of raw materials for process 10",
                   "X21208"
```

PLIST and SEQUENCE numbers

As mentioned already, each instantiation of a PLIST must be assigned a unique integer identifier which is used at run-time to reference the PLIST items. Similarly a SEQUENCE must be assigned a unique integer identifier by the programmer in order to reference the sequence at run-time. We propose the introduction of a project file that controls the execution of the project. This file would free the programmer from the need to know and define SEQUENCE and PLIST numbers and serve to centralise the definitions and declaration

Problem	Solution implemented
• Four fixed-named local variables	General local variables via LOCAL PLIST.
• Only PLISTs as parameters	Interface variables
• Limited looping construct	FOR and WHILE statements
• Accuracy of real numbers	32-bit IEEE format floating point numbers
• Only numbered RTL procedure calls	Support for symbolic calls to C procedures
• Only 256 numeric labels	2^{32} symbolic labels
• Limited LET definitions	Parameters; nested invocation; nested file includes
• Boolean operator precedence	–
• Lack of free format	–
• START and CALL statement syntax	new syntax
• Optional THEN in IF statement	–
• Multiline LET with terminator	–
• No string assignment	–
• PLIST instantiation and initialisation	–

Figure 3: Summary of Improvements

of data. Currently every module is required to include a PLISTDEF and PLISTNUM section to define and declare those PLISTs that are used in the sequences contained in the module. The SEQNUM section is required to associate the sequence with a number and provide a prototype against which to check usage of that sequence.

The project file would contain all PLIST definitions and declarations for the entire project as well as all the SEQUENCE prototype definitions. It would then be the compiler's responsibility to assign appropriate numbers. Individual modules would be compiled against the project file and the system would ensure that dependent modules are recompiled as appropriate.

4 Implementation Issues

A new CASL compiler has been developed that implements the improvements described before that could be made while maintaining upward compatibility. Figure 3 summarizes the improvements identified and implemented. Those that were not implemented were either because of the compatibility issue, or because the initiators of the project saw no immediate need for it.

In the rest of this section we describe some of the problems that had to be solved when implementing the compiler using PCLEX and PCYACC [1].

4.1 Language Specification

Initially an attempt was made to make the language free-format without explicit statement terminators. However, as explained in a previous section, this was not possible. Newlines were therefore encoded as statement terminators, with the same restrictions and exceptions as were present in the original CASL.

As has been indicated before, the complete **START** and **CALL** statements cannot be implemented using an LALR(1) parser (which is what PCYACC produces). In particular, the problem is when PLISTs are initialised in these statements. Fortunately it turned out that in practice, **START** and **CALL** statements are not often used with PLIST initialisation. This, coupled with the desirability of a grammar-based parser was judged to be a good enough reason to change the syntax of these initialisations to the format described earlier. This is the only case in which the new language is not upwardly compatible with the old.

4.2 Preprocessor and Lexical Scanner

PCLEX was used to generate a lexical scanner from regular expressions defining the CASL tokens. An attempt was made to incorporate file inclusions and macro definitions and calls directly into the scanner and parser. This could be achieved by changing the source of the scanner input at appropriate times. However, because PCLEX generates a scanner that uses the more efficient, so-called *flex* algorithm, this was not possible. Instead of reading input a character at a time, a *flex* scanner reads input a line at a time and then examines the characters in the internal buffer. Redirecting input source under these circumstances became too complicated and cumbersome, and a separate, hand-coded preprocessor was developed using a public domain C preprocessor as a base.

A minor problem arose during the implementation of the lexical scanner. In CASL two or more strings separated only by a newline are considered as one string. A string token was therefore defined accordingly. However, this caused the scanner's internal buffer to overflow on long strings. The buffer size of the scanner could have been extended, however, one would not be able to guarantee that it is big enough since there is no limit on the lengths of strings (in some applications strings of more than 1K bytes are defined). The problem was solved by moving the recognition of long strings to the parser. Rules were defined to recognise lists of strings only separated by newline characters and to concatenate these strings and store them as a single string.

Tag name to Plant Address conversion

The convention for referencing plant addresses in CASL is by tag name, which is usually application dependent. The standard CYGNUS syntax requires that a tag name begin with a prime (') followed by up to nine significant characters, drawn from the letters **A–Z**, digits **0–9** and **@**. The layout characters **-** and **/** may be used freely, e.g. **'FIC-410/A**, **'@MASSFLOW**, and **'77-HV-101/A**. A tag name that is used in a CASL module must be present in the CYGNUS database in order to be valid. Since each PLIST item is represented internally by a 32 bit value (16 bits in the original CASL), tag names, which are the "values" of **PLAD** variables, need to be converted to 32 bit values. In other words, a **PLAD** is a 32-bit representation of a tag name.

In the old version of the CYGNUS system the conversion from tag name to PLAD, was performed by the translator. This was achieved by simply passing the tag name to a special dictionary task, which looked up the name in a dictionary and returned an associated 16 bit number that served as a kind of index in the database. This number was then stored in the PLAD item and used for subsequent database access. The disadvantage of this approach is that if the database is changed, all modules referencing the database have to be re-translated.

In the new compiler, the tag name to **PLAD** conversion is left to the loader, to be performed at load time. The compiler records every occurrence of a tag name and passes

this information to the loader. Multiple occurrences of the same tag name will result in a chain of pointers being formed from the last occurrence to the first. The loader converts the tag name and inserts the packed value into the intermediate code. The loader thus becomes responsible for the validation of tag names.

Run time conversion and validation is not required since there is a restriction that prevents the database from being updated while a sequence that uses it is running.

5 Conclusion

We have examined CASL, a specialised high level language for batch control, and concluded that, while it serves well as a batch control language, it could do with some improvements as a high level language. We have identified specific areas for improvements to functionality, clarity and readability and have proposed specific improvements. The majority of these improvements can be implemented while keeping the new language upwardly compatible with the old. Finally, we have implemented an LALR(1) compiler that incorporates all the upwardly compatible improvements. Except for one infrequently used construct, the new compiler will compile all existing CASL code in order to protect the large investments made in CASL.

References

[1] Abraxas Software, Inc., 7033 SW Macadam Ave., Portland, Oregon, 97219, USA. *PCYACC.*

[2] AECI Process Computing, P.O. Box 1823 Halfway House South Africa. *CYGNUS PROCESS AND MANAGEMENT SOFTWARE.*

[3] AECI Process Computing, P.O. Box 1823 Halfway House South Africa. *CASL USER MANUAL*, 8 edition, Jan 1987.

[4] C. Charalambous and A.J. Conning. Distributed sequence control utilising a high level sequencing language in conjunction with PLCs. In I.M. Macleod and A.D. Heher, editors, *Software for Computer Control 1988. Proceedings of the fifth IFAC/IFIP Symposium*, pages 27–36, 1988.

[5] R.W. Dehning. Private Communication. APC Research Associate.

[6] M.A. Keyes, D.M. Norris, and J.G. Patella. A hybrid approach to batch control. In *Generic Control for Batch Processing*, pages 91–96, Purdue, 1990.

[7] D.B. Leach. Specifying a batch-process control system. *Chemical Engineering*, 93(23):115–122, 1986.

[8] L.E. Massey and J.J. McCarthy. Batch process automation for the "90. In *Generic Control for Batch Processing*, pages 17–37, Purdue, 1990.

[9] Terrence W. Pratt. *Programming Languages: Design and Implementation.* Prentice Hall, New Jersey, second edition, 1984.

[10] S.E. Rubin. Supervisory batch control using personnal computers. In *Generic Control for Batch Processing*, pages 103–106, Purdue, 1990.