The ModulaTor

Oberon-2 and Modula-2 Technical Publication

The ModulaTor
Erlangen's First Independent Modula-2 Journal! Nr. 5/Jun-1991

The EDISON Multiprocessor Language

by Guenter Dotzel, ModulaWare GmbH.

Edison is a minimalistic, block-structured, modular, real-time, multiprocessor language. It was developed in 1980 by Per Brinch Hansen of the University of Southern California, Los Angeles, CA as a successor of Pascal and Modula(-1). For more information see his publication (a) Edison - a Multiprocessor Language, (b) The Design of Edison, and (c) Edison Programs in Software, Practice & Experience, Vol. 11, p325-361 (1981), John Wiley & Sons.

Edison did not yet get much attention in the programmers community. It's the smallest multiprocessing language I know. It's definition including syntax summary and many examples takes only 35 pages. Like Modula, it doesn't include I/O-statements. Also Edison omits all unsafe features of Pascal-like languages namely pointers and variant records. Further omitted are subranges, files, unnamed (anonymous) types, goto-, case- and with-statement. Edison features only one loop construct, the while-loop which has an else-clause.

The only extensions are array- and record-constructors, procedural parameters, arbitrary length sets and retyped factors for system programming. Last but not least, for multiprocessor programming, Edison includes the synchronizing statement when ... do ... end and the concurrent statement cobegin ... also ... end. To show that Edison is powerful enough for most programming applications, Per Brinch Hansen wrote the Edison compiler, a single user operating system including file system and utility programs, an assembler (called Alva), a text editor and a text formatter in Edison.

Question: Why yet another language? Why EDISON?

Answer: Safety in programming of mission critical systems!

Safety in programming can be guaranteed only when (a) writing programs in a language like Edison which is well suited for verification and (b) by using only verified tools (Edison compiler). It is important to note, that verification is the only goal when security is a concern. If there exists any language with a complete and exact (formal) static as well as semantic definition, it is indeed possible to produce a validation suite (a large set of testing programs). But like testing, validation can only show the absense of errors but does not proof correctness.

How can safety be guaranteed with Edison?

1. The Edison language is small. 2. The exact definition of Edison's syntax and semantics is small. 3. Since the Edison compiler is written in Edison and generates code for an abstract machine (so-called e-code, which was designed to be ideal for the execution of Edison programs), the compiler itself is small. 4. Compiling an Edison program is equivalent to the transformation from Edison source code to e-code. 5. The correctness of the transformation process by compilation can be guaranteed by verifying

the compiler. 5. So it can easily be proved that the compiler compiles itself correctly to e-code. 6. The e-code of the compiler can be executed on any existing machine which is able to run an e-code-interpreter. 7. An e-code-interpreter is a small program with (7a) a formal specification in Edison (the correctness of this specification can be proved) and (7b) an interpreter program written in a portable assembly language called Alva. 8. The equivalence of 7a and 7b can be proved. 9. The Alva assembler is written in Edison. 10. To port any Edison program to a new machine, Alva has to be adapted to generate the codes and instruction format of the target machine's architecture. The correctness of this adaption can be proved. 11. The correctness of the transformation from Alva into the native code of the target machine can be proved.

EDISON Example: The producer/consumer relation

The example listed below is a complete compilation unit. All exported objects have an asterisk as prefix, importation into modules is automatic. All reserved words (keywords) are printed in bold. Comments are enclosed in quotes. The procedure main which is parameterized with a read- and a write-procedure contains two local modules semaphores and putget. Both modules have a body (initialisation part) which is executed when procedure main is called. Furthermore main declares two procedures producer and consumer which are going to be instantiated as concurrent processes in the body of main.

```
proc main (proc read (var ch: char); proc write (ch: char))

  module "semaphores"
   *record semaphore (value: int)

   *proc wait(var s: semaphore)
    begin
      when s.value > 0 do
        s.value := s.value-1
      end
    end "wait"

   *proc signal(var s: semaphore)
    begin
      when true do
        s.value := s.value+1
      end
    end "signal"

   *proc newsem(var s: semaphore; n: int)
    begin
      s.value := n
    end "newsem"

  begin skip end "semaphores"

  module "putget"
    const n=10 "slots"
    array table [1:n](char)
    var ring: table;
      head, tail: int;
      full, empty: semaphore

   *proc put(c: char)
    begin
      wait(empty);
      ring[tail]:=c;
      tail := tail mod n+1;
      signal(full)
    end "put"

                                                *proc get(var c: char)
```

```
        begin
          wait(full);
          c := ring[head];
          head := head mod n+1;
          signal(empty)
        end "get"

      begin
        head:=1;
        tail:=1;
        newsem(full,0);
        newsem(empty,n)
      end "putget"

      proc producer
      var x: char
      begin read(x);
        while x<>'.' do
          put(x); read(x)
        end; put(x)
      end "producer"

      proc consumer
      var y: char
      begin get(y);
        while y<>'.' do
          write(y); get(y)
        end; write(y)
      end "consumer"

    begin
      cobegin 1 do producer
      also 2 do consumer
      end "cobegin"
    end "main"
```

Edison program execution, compilation and self-compilation:

To illustrate, how the interpreter executes a program, we need four symbols A, B, C, I. A is program input, B is an e-code module, C is program output and I is the e-code interpreter (see illustration below). 1. To execute an Edison program, one executes it's e-code B which possibly reads A and produces C. 2. For compilation, A is an Edison source file, B is the e-code of the Edison-compiler and C is the e-code of A. 3. For self-compilation A is the source of the Edison compiler, B is it's e-code which is reproduced in C.

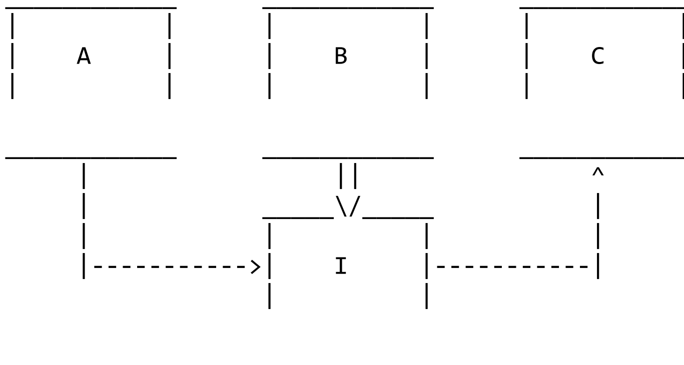

Fig. 0: How Elmar Baumgart, 22 years old, visualizes an interpreter

More about the Edison-future at ModulaWare

Edison will be available for a wide range of todays home- and personal computers, workstations, mini- and mainframes. The initial target machine is VAX/VMS (because it's the development platform) and INMOS Transputer T800 and T9000 (H1). The system is highly portable, because it is code is compatible across all 32 Bit architectures without recompilation. This is accomplished by not generating native machine code but e-code. E-code can be moved to other machines and operating systems by re-writing the interpreter. The interpreter is easily ported because it doesn't contain operating system specific code. This is due to the fact, that neither files nor any other I/O-mechanisms are part of the Edison language. The e-code contains a 32-bit byte-stamp, which tells the interpreter how the binary bytes are alocated in a 32-bit word. Furthermore, since the e-code contains information about the binary floating point number data format either ISO, IEEE-754-1985 (single and double) or VAX (single, double and g_floating), the interpreter is able to do the necessary format conversions before executing the e-code.

Of course, interpetative program execution is 2 to 10 times slower than native code. But with the advent of very fast processors such as the new H1-Transputer (150 MIPS, 25 MFLOPS, 80 MHz links, due to July, 1991) this fact can be tolerated for most applications not only for educational purposes. An Edison program on the H1 would run faster than a native-code compiled program on todays 30 MIPS RISC-workstation.

Implementation status trace

1982: Licensed the compiler and wrote some Edison programs (I gave up in favour of RT-11 (and later SHAREplus) operating system on PDP-11 because program development under the Edison operating system was too time consuming and also the system and the language was incompatible to everything). Dec-1990: Wrote a Modula-2 tool to be able to read source and data files from Edison's file system on the PDP-11/RT-11, copied the source files to VMS. Mar-1991: Studied the Edison language design issues and the compiler sources. Translated the Edison source code with a programmable editor partly automatically to Modula-2. Elmar, a new employee of ModulaWare took over the Edison project and for bootstrapping purposes he re-wrote the Edison compiler in Modula-2 within two weeks. The compiler has 5100 lines of Modula-2 source code. ModulaWare's Modula-2 compiler MVR V3.09 was used for compilation; this versions allows a set size of up-to 128 elements.

Apr-1991: The e-code interpreter was re-written from Edison to Modula-2. The Edison language was extended by the elementary data types REAL, LONGREAL, COMPLEX, LONGCOMPLEX and associated operations. Also generic constant real numbers, so-called RR-types (as in ISO-Modula-2) have been added. The Edison procedural program parameters were extended by the complete set of the ISO-Modula-2 standard library data types and procedures (from the 4th Interim Working Draft, dated May-1991). The Edison compiler (4966 lines) compiles itself to e-code within 40 minutes on a slow microVAXII (about 1 MIPS, 0.3 MFLOPS).

Jun-1991 First benchmarks, using the portable e-code interpreter written in ISO-Modula-2 (less than 1000 source lines): the following programs written in Edison were compared to their equivalent written in Modula-2 using MVR V3.13 with the /NOcheck compilation option.

(A) the Ackermann-function runs 11 times slower,

(B) the FFT-algorithm using single precision reals runs 23 times slower.

LECTOR/PS dw2:edison.dw2 10-Jul-1991 \251 1991 Guenter Dotzel.

Scanner-Syntax:

```
letter: 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' |
  'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' |
  'u' | 'v' | 'w' | 'x' | 'y' | 'z'
digit: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

```
apostrophy: '"'
special_character: ''' | '(' | ')' | '*' | '+' | ',' | '-' | '.' | ':' | ';' |
  '<' | '=' | '>' | '[' | ']' | '_'
graphic_symbol: letter | digit | special_character | apostrophy | space
comment: { letter | digit | special_character | space | new_line_character }
separator: space | new_line_character | '"' comment '"'
name: letter { letter | digit | '_' }
numeral: digit { digit }

Parser-Syntax:

type_name: name
constant_name: name
procedure_name: name
variable_name: name
field_name: name
control_symbol: 'char' '(' numeral ')'
constant_symbol: numeral | character_symbol | constant_name
character_symbol: graphic_symbol | control_symbol
constant_declaration: constant_name '=' constant_symbol
constant_declaration_list: 'const' constant_declaration { ';' constant_declaration }
enumeration_symbol: constant_name
enumeration_symbol_list: enumeration_symbol { ',' enumeration_symbol }
enumeration_type: 'enum' type_name '(' enumeration_symbol_list ')'
record_type: 'record' type_name '(' field_list ')'
field_list: variable_list
range_symbol: constant_symbol ':' constant_symbol
array_type: 'array' type_name '[' range_symbol ']' '(' element_type ')'
element_type: type_name
set_type: 'set' type_name '(' base_type ')'
base_type: type_name
type_declaration: enumeration_type | record_type | array_type | set_type
variable_group: variable_name { ',' variable_name } ':' type_name
variable_list: variable_group { ';' variable_group }
variable_declaration_list: 'var' variable_list
parameter_group: [ 'var' ] variable_group | proc_heading
parameter_list: parameter_group { ';' parameter_group }
procedure_heading: 'proc' procedure_name [ '(' parameter_list ')' ] [ ':' type_name ]
procedure_body: { declaration } 'begin' statement_list 'end'
complete_procedure: procedure_heading procedure_body
preprocedure: 'pre' procedure_heading
postprocedure: 'post' complete_procedure
library_procedure: 'lib' procedure_heading '[' expression ']'
procedure_declaration: complete_procedure | preprocedure | postprocedure | library_procedure
module_declaration: 'module' { [ '*' ] declaration } 'begin' statement_list 'end'
declaration: constant_declaration_list | type_declaration |
  variable_declaration_list | proc_declaration | module_declaration
function_variable: 'val' procedure_name
field_selector: '.' field_name
indexed_selector: '[' expression ']'
type_transfer: ':' type_name
variable_symbol: variable_name [ type_transfer ] | function_variable | variable_symbol selector
selector: field_selector | indexed_selector | type_transfer
constructor: type_name [ '(' expression_list ')' ] [ type_transfer ]
expression_list: expression { ',' expression }
factor: constant_symbol | variable_symbol | constructor | procedure_call |
  '(' expression ')' | 'not' factor
multiplying_operator: '*' | 'div' | 'mod' | 'and'
term: factor { multiplying_operator factor }
adding_operator: '+' | '-' | 'or'
simple_expression: [ sign_operator ] term { adding_operator term }
relational_operator: '=' | '<>' | '<' | '<=' | '>' | '>=' | 'in'
expression: simple_expression [ relational_operator simple_expression ] [ type_transfer ]
assignment_statement: variable_symbol ':=' expression
argument_list: expression { ',' expression }
```

```
procedure_call: procedure_name { '(' argument_list ')' ]
conditional_statement : expression 'do' statement_list
conditional_statement_list: conditional_statement { 'else' conditional_statement }
if_statement: 'if' conditional_statement_list 'end'
while_statement: 'while' conditional_statement_list 'end'
when_statement: 'when' conditional_statement_list 'end'
process_statement: constant_symbol 'do' statement_list
process_statement_list: process_statement { 'also' process_statement }
concurrent_statement: 'cobegin' process_statement_list 'end'
statement: 'skip' | assignment_statement | procedure_call | if_statement |
  while_statement | when_statement | concurrent_statement
statement_list: statement { ';' statement }
program: { initial_declaration } complete_procedure
initial_declaration: constant_declaration_list | type_declaration
```

IMPRESSUM: The ModulaTor is an unrefereed journal. Technical papers are to be taken as working papers and personal rather than organizational statements. Items are printed at the discretion of the Editor based upon his judgement on the interest and relevancy to the readership. Letters, announcements, and other items of professional interest are selected on the same basis. Office of publication: The Editor of The ModulaTor is Guenter Dotzel; he can be reached by tel/fax: [removed due to abuse] or by mailto:[email deleted due to spam]

ModulaWare home page   The ModulaTor download

Webdesign by www.otolo.com/webworx, 14-Jul-1998