GUIDE TO SOFTWARE

for

BIT-VECTOR ALGORITHMS FOR BINARY CONSTRAINT SATISFACTION

Julian R. Ullmann

email: jrullmann@acm.org

Table Of Contents

1 Programming language and compilers

1.1 Modula-2

After programming in low level languages from 1961 to 1975, and then in Pascal, the author switched in 1986 to Modula-2 (with the Cambridge compiler under Unix), then TopSpeed Modula-2 in 1989, then XDS Modula-2/Oberon-2 from 1999 onwards. Although it has not won widespread acceptance, Modula-2 continues to be eminently suitable for the present work. Modula-2 can easily be read and understood by Pascal programmers: Section 3 of this document provides brief notes to facilitate understanding. Moreover, excellent XDS Modula-2/Oberon-2 compilers are available as *freeware* from `http://www.excelsior-usa.com/xds.html`. For Linux and for Windows, there are XDS compilers that either produce x86 (e.g. Pentium) object code or C. The C code can be run via ordinary C compilers: the Excelsior website provides details. Thus the accompanying software can be run on many platforms without inordinate effort.

1.2 Brief notes on using XDS

After installing an appropriate XDS compiler, the associated TSCP should next be installed. TSCP stands for *TopSpeed Compatibility Pack*; this includes a valuable library of modules, for example for timing.

We recommend that the subdirectory organization of the accompanying material should be preserved, at least until the software works satisfactorily. We recommend that the four accompanying subdirectories be located in a subdirectory *BVmodules* so the path is something like *C:/modules/BVmodules/BVrand*, or *C:/modules/BVmodules/BViso*, etc. Section 2.1, below, provides further information about this subdirectory organization.

XDS works either from the command line or from an Integrated Development Environment (IDE) which is a graphic user interface. We recommend working initially from the IDE because this is easy. To run, for example, the program *BVRmain*, open the IDE, click on project at the top of the screen, then click on open, then browse to *BVrand/BVRmain* and click on it. Then click on the *open* icon and select successively *BVRmain.mod* then *dex.def*. Of course other modules can be opened subsequently. Then click on the man-running icon to compile, link and run the program.

Compile-time errors are reported in a *messages* window. Under XDS, each program has its own project (.PRJ) file. Run-time errors are reported helpfully if the project file includes:

```
- GENHISTORY +
- LINENO +
```

After a run-time error click on *tools* and then click on *history*.

2 Files and subdirectories

2.1 Contents of subdirectories

Subdirectory contents are as follows:

BVrand includes a project file and software implementing bit-vector algorithms, all with randomly generated constraint satisfaction problems, not with isomorphism. Within this subdirectory:

- **BVred** includes three different domain reduction procedures, together with procedures for preprocessing and forward checking.
- **quAndStacks** includes queue and stack modules.
- **Focus** includes focus search modules.

BViso includes a project file and software implementing bit-vector algorithms for randomly generated isomorphism problems, but not for focus search. Within this subdirectory:

- **BVisoRed** includes three different domain reduction procedures, together with procedures for preprocessing and forward checking.
- **unary** includes three different modules that apply unary constraints.
- **quAndStacks** includes queue and stack modules for use with isomorphism modules.

isoFocus includes a project file and software implementing focus search with randomly generated isomorphism problems. Within this subdirectory:

- **funa** includes three different modules that apply unary constraints for use with focus search.

common includes modules that export to many other modules.

There is much overlap of contents of some of these subdirectories. For example there are two *quAndStacks* subdirectories. Many modules are very similar to others. This duplication could easily be avoided, but the resulting software would be more difficult to understand. Where have had to choose between avoidance of duplication and ease of comprehension, we have chosen ease of comprehension.

The subdirectory *BVrand* includes a focus search subdirectory and does not have a separate project file for focus search. The *isoFocus* subdirectory is not a subdirectory of *BViso* and does include a separate project file for focus search. Again, this is intended to make the software easier to read.

We have not included software that reads constraint satisfaction or isomorphism problems from files. Yet again, if this were included, comprehension would be more difficult. We have omitted many modules containing straightforward programming that is mainly not of research interest.

2.2 Program parameters

In subdirectory *BVrand*, the definition module *dex.def* declares and sets values of parameters such as the number of variables. The name *dex* is short for *declarations*. The number of random trials is declared as a constant in *BVRmain.mod*. Similarly, *gdex.def* in subdirectory *BViso* sets values for parameters for isomorphism programs. When any such parameter value is changed, at least one module has to be recompiled, and the program has to be re-made (i.e. re-linked).

2.3 Comments and commentary files

In Wirth's Modula-2, comments are enclosed between (* and *). We use such comments sparingly within the program text. Excessive use of comments clutters the text and can make it harder to read.

XDS Modula-2 ignores the contents of a line to the right of '`--`'. This can be used for comments. We have used this to label lines of program text, for example by putting `--sc` at the right hand end of a line. These labels refer to comments in a separate commentary file. At a line commencing `--sc` in a commentary file there is explanation of the line ending `--sc` in the associated program module. For a program module named *foo.mod*, the associated commentary file, if it exists, is named *foo.txt* and is in the same subdirectory. There are no commentary files for modules, such as queue and stack modules, that hopefully do not require explanation. And there are no commentary files for modules that are very similar to others that do have commentary files.

A commentary file is designed to be open in one window whilst the associated program text is open in another window. For this reason, for use in XDS IDE windows, commentary files have no format and no line-wrap.

3 Very brief notes on the language Modula-2

3.1 Modules

The principal difference between Pascal and Modula-2 is that a Modula-2 program may consist of many modules that can be compiled separately. A program named *programExample* has a main module headed by:

```
MODULE programExample;
```

and ending

```
END programExample.
```

This module will be in a file named *programExample.mod*, which in XDS will be located in the same directory as the program's project file (named *programExample.prj*). All reserved words, e.g. MODULE, and all standard identifiers, e.g. TRUE, are in upper case.

The main module cannot export identifiers but may import identifiers from any number of other modules. Each of these typically implements a data structure and procedures that operate on it. An exporting module is written in two parts: the definition module, which begins e.g.

```
DEFINITION MODULE stacks;
```

and the implementation module, which begins e.g.

```
IMPLEMENTATION MODULE stacks;
```

In this example the definition module is in a file named *stacks.def* and the implementation module is in a file named *stacks.mod*. In XDS the project file tells the system where to find exporting modules that are not in the same subdirectory as the main module. The definition module includes all CONST, TYPE and VAR declarations that may be imported by other modules. The definition module also includes copies of the headings of all procedures that may be imported by other modules. These are copies of headings because complete head-and-body declarations of these procedures must be included in the implementation module.

Bodies of exporting implementation modules are executed before commencement of execution of the main module. The body of an implementation module may be empty. Definition modules of exporting modules must be compiled before modules that import from these are compiled. A definition module must be compiled before the corresponding implementation module can be compiled.

There are two ways to declare identifiers imported from other modules. If we write

```
IMPORT stacks;
```

then identifiers imported from *stacks.def* can be referenced as e.g *stacks.push* and *stacks.pop*. In *stacks.pop* the exporting module name *stacks* qualifies the exported identifier *pop*. Alternatively, imported identifiers can be referenced without qualification if they are explicitly declared for example by

```
FROM stacks IMPORT push, pop;
```

In this case the importing module can simply reference, for example, *pop*.

3.2 Control structures

After THEN or DO in Pascal there is a single statement that may optionally be a compound statement. After THEN or DO, etc, in Modula-2, there is a statement sequence terminated by a reserved word such as END or ELSE. Modula-2 includes LOOP, for example

```
LOOP
  statement1; statement2; etc;
  IF condition THEN EXIT END
  anotherStatement; yetAnotherStatement; etc
END;
```

Statements enclosed by LOOP and END are repeated until EXIT transfers control to the statement that follows END.

A FOR statement has an optional BY part:

```
FOR x:= a TO b BY k DO statementSequence END;
```

This executes *statementSequence* for $x = a$, $x = a + k$, $x = a + 2 * k$, etc. The increment k may be negative. There is no DOWNTO in Modula-2. In a FOR statement that does not have BY, the increment is $+1$.

In Modula-2, `HALT` terminates program execution.

3.3 Procedures

Modula-2 has proper procedures and function procedures. A proper procedure such as

```
PROCEDURE push(k);
BEGIN
  INC(top); stack[top]:= k  (* INC(top) means top:= top + 1 *)
END push;
```

does not assign a value to the procedure name. Procedure execution terminates, and control is returned to the statement following the call, either at the end of the procedure body or when a RETURN statement is obeyed. It is legitimate for a proper procedure to have no parameters in brackets.

A function procedure such as

```
PROCEDURE matches(VAR A,B: ARRAY OF REAL; n: CARDINAL): BOOLEAN;
VAR
  i: CARDINAL;  (* CARDINAL simply means non-negative integer *)
BEGIN
  FOR i:= 0 TO n DO
    IF A[i] # B[i] THEN RETURN FALSE END; (* # means NOT EQUAL TO *)
```

```
  END;
  RETURN TRUE;
END matches;
```

does assign a value to the function-procedure name. This value is specified after RETURN. A function-procedure terminates only at a RETURN statement. Another rule is that the heading of a function procedure must include '(' and ')', but there may be nothing between these. The type of the value returned by the function procedure is specified after a colon at the end of the procedure heading. As in Pascal, VAR in a procedure heading signifies call-by-reference.

As in this example, an array parameter need not be of a fully declared type. In the body of procedure *matches*, the bounds of array A are 0..HIGH(A). HIGH is a standard function in Modula-2, which includes other standard functions such as DEC(i) which means $i := i - 1$. Another is VAL(INTEGER, i) which returns the value of variable i converted to type INTEGER. If p is a pointer so that, for example, $p\uparrow$ is an array, then VAL(CARDINAL, p) returns the pointer as a non-negative integer. In program text $p\uparrow$ is written
`p^`

A proper procedure or a function procedure may have a parameter that is itself a procedure. This facility could easily be used to remove much duplication in the present work, but sometimes at the cost of making programs harder to understand.

3.4 Sets

A set of discrete values can be declared by, for example,

```
TYPE
  smallInts = SET OF [-5..+19];
  charSet   = SET OF CHAR;  (* CHAR means character *)
VAR
  small: smallInts;
  ucAlphabetic, pChars: charSet;
```

After these declarations we could have, for example,

```
small:= smallInts{-3, 2, 5..9, 15};
  (* Set contents are enclosed between '{' and '}'.
     The type name smallInts must be included here
     to enable type checking *)
IF i IN small THEN doSomething END;
  (* The reserved word IN is self-explanatory *)
small:= small + smallInts{4, 11};
```

```
  (* Here '+' means set union *)
small:= small * smallInts{-1, 2, 4, 7, 12};
  (* Here '*' means set intersection *)
INCL(small, 14);
  (* This standard procedure includes 14 in small *)
EXCL(small, 2);
   (* This standard procedure removes 2 from small *)
pChars:= charSet{};
  (* The type name must precede '{'.  {} means an empty set *)
ucAlphabetic:= charSet{'A'..'Z'};
  (* Again the type name must precede '{' *)
```

A PACKEDSET is physically a bit vector composed of one or more words, in which one bit is associated with each possible member of the set. '1' signifies that the associated member is now in the set; '0' means that the associated member is now absent. After the set declarations

```
TYPE
  domainType = PACKEDSET OF [0..VUB];
VAR
  C, D, E: domainType;
```

we can apply any of the Modula-2 set operations to these sets, e.g,

```
  C:= domainType{}; (* The type name must precede '{'.
    {} means an empty set *)
  INCL(C,v); C:= D + E; C:= D * E; EXCL(C,y);
  D:= D - E; (* set difference, which removes
    from D each value that is also in E *)
  IF NOT (v IN E) THEN doSomething END;
```

Thus in Modula-2 we can work with bit-vectors without seeing their underlying implementation in one or more words.

3.5 Pragmas: Conditional compilation

If at the beginning of a module we have for example

```
<* NEW SINGLE + *>  (* An example of a compiler option or switch *)
<* NEW APPEND - *>  (* An example of a compiler option or switch *)
```

Then in the program

```
<* IF SINGLE THEN *> aStatementSequence;
<* ELSE *> bStatementSequence;
<* END *>
```

```
   (* Here  aStatementSequence is compiled and
      bStatementSequence is ignored by the compiler *)
<* IF APPEND THEN *> cStatementSequence;  <* END *>
   (* cStatementSequence is ignored by the compiler,
      which behaves as if cStatementSequence were
      absent from the program text *)
```

4 Naming conventions in this software

In the accompanying modules, identifiers chosen by the programmer almost always start with a lower case letter. Identifiers imported from library modules start with an upper case letter. A procedure identifier that starts 'mc' signifies that the procedure builds something; here 'mc' is short for 'make'. For example procedure *mcMatrices* constructs bit matrices. Procedure *mcGraphs* constructs a pair of random graphs for use in subgraph isomorphism experiments.

A procedure identifier that starts *see* displays contents of a data structure. For example, procedure *seeDomains* outputs the current contents of domains. Procedure *seeDcards* outputs the current cardinalities of domains. Modules include *see* procedures that are not called. These (mainly trivial) procedures have been used during program development and have been retained because they may be useful to anyone who investigates detailed behavior of modules.

Identifiers that start with *nb* or *noOf* signify *number of*. For example, *noOfScopes* is the number of scopes. *nbTrials* is the number of trials.

In modules, `iMj` means M_i^j. And `jMi` means M_j^i.

These modules send output to a file named *peep* via a module named *peepa*. This file serves as a *stdout* file. The call `peepInt(i, 4)` writes integer i to the file *peep* occupying four character positions. Other procedures exported by module *peepa* have explanatory comments in *peepa.mod*. For example `meepInt(m, i, 4)` writes to the file *peep* a text-string m followed by integer i occupying four character positions. *meep* procedures output a text message and then a value. *peep* procedures output a value without a preceeding text message. The file *peep* is located in the same subdirectory as the project file. In output files, values are sometimes represented by letters instead of numbers because this may be easier to read.

These modules perform a prescribed number of trials. Results produced by each trial are sent to module *anna* (which is short for analyser). Module *anna* gives us the average, the minimum, the maximum and the standard deviation of results in various categories. For example, if *nodesVisited* is *Category 1* then `annaCard(1, nodesVisited)` updates Category 1 using the number of nodes visited in each successive trial. Similarly, if *totalTime* is *Category 2* then `annaLngReal(2, totalTime)` updates Category 2 using the total time for each successive trial. Here *LngReal* is an abbreviation for LONGREAL which is a

64-bit REAL in the XDS implementation. In many cases the statistical distribution is far from normal, so standard deviation is merely a better-than-nothing indication of the extent of variation of the variate.