A Background to Program Generators for Commercial Applications

Roger Clarke*

The emergence and key features of program generators are explained. Examples are given of the appearance and use of one particularly advanced product.

Keywords and Phrases: application generator, DELTA, macro language, macro processor, pre-processor, program generator, software portability, specification language, very high level language.
CR Categories: 41.2, 4.22.

INTRODUCTION

Since the dawn of programming better methods have been sought. One major focus has been upon efficiency in the use of processor and main memory resources, and in some circumstances these factors remain paramount.

Another focus has been on the manner in which programs are prepared. Varying degrees of maturity have been reached in the many aspects of languages. Their power has developed to such an extent in fact, that they offer far more than that needed by the vast majority of applications. As a result of this the problems arise of finding sufficient staff who are sufficiently highly trained to handle such languages, then constraining them to a narrow (and partly arbitrary) discipline in its use.

In order to combat such problems new methods of program preparation are emerging. These depend on parameter driven utility programs which generate a high level language program. Sub-problems may still require the power of the host language; for such cases it is necessary to be able to insert code into the appropriate location in the generated program. It is reasonable to view such program generators as preliminary attempts at future higher level languages. They are however identifiable products, and have some characteristics different from existing languages. This article will deal with them independently from questions of language design.

After a brief discussion of the reasons which stimulated the production of program generators, their emergence is traced and the concepts central to the theory are presented in stepped form. Examples are given based on one such product, and brief comments provided on the impact of the new tools.

THE STIMULUS

Modern theories of system and program development are poorly served by old languages and programming environments. Yet the enormous investment in software and in trained software development staff precludes a simple-minded revolution. One approach to provide a 'bridging' technology between old and new is to install a pre-processor before the compiler, to enable and/or require programmers to write in structured style, despite the weaknesses of the host language. In addition other deficiencies in the language can be catered for. An important product in this field was MetaCOBOL (see ADR, 1974a, 1974b), a commercial application of the 'Stage II' generator (Waite, 1974). It offered the ability to create additional verbs (case-construct, in-line PERFORM, initialise-table), to improve syntax (explicit ENDIF, a quasi-local variable feature), to recognise multiple alternative short forms, and to 'massage' the layout of the code for consistent presentation and indentation – critical factors in making programs readable and maintainable by persons other than the author.

The problem with conventional high-level languages, even when front-ended in this way, is that their power and complexity demand considerable expertise on the part of the programmer. Few problems arise in commercial programming that aren't capable of appropriate solution; but there are far too few suitably trained people to do the solving. Given that the vast majority of development groups work within a fairly small set of (partly consciously chosen) techniques, the full power of the host language could be foregone.

An additional problem is the matching of programming technology to the system analysis and design technologies that precede it in the application-software production-line. It is now fairly clearly established that multiple languages at different levels of abstraction are necessary (Hawryszkiewycz, 1981) and that therefore language translation problems will occur. In addition these languages can be expected to require some time yet before they stabilise, and the likelihood of multiple alternative languages at any given level of abstraction seems to be quite high. It is therefore desirable that the interface between the design and the programming syntaxes be supported by a powerful macro-language. Only in this way can the programmer/coder in all cases be provided with the means to perform simple, quick and efficient translation from the design documents/text files into compilable code.

THE DEVELOPMENT PATH OF PROGRAM GENERATORS

Progress has been achieved incrementally, and this

*Ueberlandstrasse 465, 8051 Zurich, Switzerland. *Manuscript received January 1982.*

article proceeds in a similar manner. The first necessary step was the realisation that commercial application development involved considerable repetition of effort, and on the other side of the coin, considerable code redundancy. Many functions were coded once per program rather than once per application, or even once for the entire installation. Several facilities have been used to overcome this wastage; for example Copy Libraries and Subprogram Calls remove localised and small-scale redundancies.

In addition to redundancy in processing code there is structural repetition. By this I mean that the majority of program structures are, or could be, formal variants of a set of models. To combat the wastage resulting from structural repetition requires a fundamental reorganisation of applications development, and investment in more effective supporting software.

The term in common use for such software seems to be 'program generator' and that term will be used in this article. Some more precise phrase such as 'parameter driven assembly of high level language programs' would be advantageous, but wordy.

PHASE 0 – REDEPLOYMENT OF STAFF

The prevailing nonsensical EDP convention of commencing to count at zero is conformed with by harking back to the most primitive, and sometimes the most effective manner of knowledge transfer. Experience in the development of commercial software is exchanged between projects in a planned manner through the assignment of staff with relevant 'know-how'. An even greater amount of experience sharing is achieved in less planned fashion thanks to the velocity of staff within the job market.

This method of knowledge transfer is entirely informal, too heavily reliant on individuals, and unmeasurable. Given the considerable variation between user applications across the various sectors of large and small primary, secondary and services industries, government enterprises and utilities and the public service it is difficult for tertiary courses to provide entrants to the information industry with directly useful applications experience.

Since formal education in such matters is difficult to come by, the interchange of staff between projects and employers will remain an important factor in knowledge transfer in all areas of computer applications. The possibility of formalising the process is greater in the more precise field of programming than in system analysis and design, yet even in this field the first steps were small and tottering.

PHASE 1 – COPY-A-PROGRAM AND AMEND

Plagiarism began with the selection of a program that bore some resemblance to the new one and the copying of the parts that seemed relevant and helpful. The method comprises Figure 1:

- selection of a model program;
- copying to a new file;
- leaving lines unchanged which are common to both programs;
- deleting lines particular to the old program;
- amending lines which are common but which contain terms particular to the program (such as the name of the program and the name of the driving file);
- inserting lines particular to the new one.

This approach can achieve significant gains:

- experience is explicitly transferred;
- it can take less time to prepare the source file;
- it can take less time to achieve a clean program;
- the resulting program is similar in style to its 'father'. It would be wrong to overlook the inherent problems.
- how is the program selected as suitable for 'fatherhood';
- how correct is 'father' as regards its original task;
- how relevant is 'father' to the new problem. Many mismatches between the two will be subtle, emerging only when testing reveals strange anomalies;
- no relationship is maintained between 'father' and 'son'. Subsequent changes in one are not easily associated with the other.

Nonetheless many organisations have profited from this technique.

PHASE 2 – COPY-A-SKELETON AND AMEND

A step which overcomes many of the deficiencies of Phase 1 is the formalisation of the 'father'. That task can require considerable investment depending on the suitability of the models available, the degree of difficulty of the program type involved, the ambitiousness of the project and the experience and competence of the staff assigned.

The preparation of the skeleton involves the following:

- define the program type to be supported;
- identify those parts of the sample program(s) common to the program type;
- define the variants of the program type which are to be catered for, and which are beyond the scope of that skeleton;
- assemble a 'first-cut' version of the skeleton from the sample(s);
- identify the variables as such. For example the driving file may have been CUST; it might be replaced with DFN (for 'Driving File Name'). In practice it is beneficial to use a string which is not legal in the source language;
- since few programs are direct analogues of one another, build in options which the programmer can select as appropriate. This might for example be

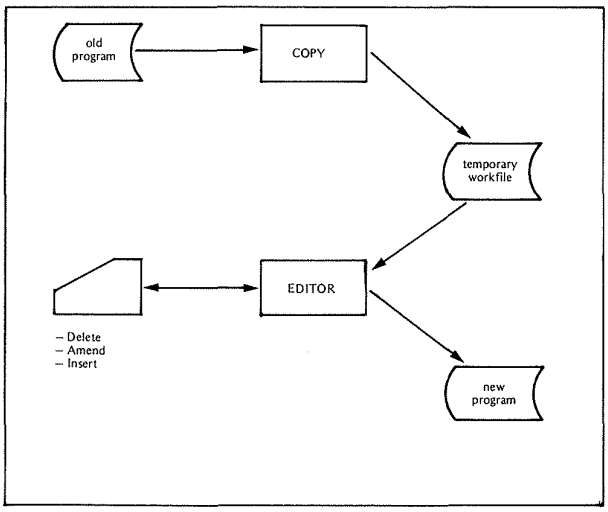

Figure 1. Copy-a-Program-and-Amend.

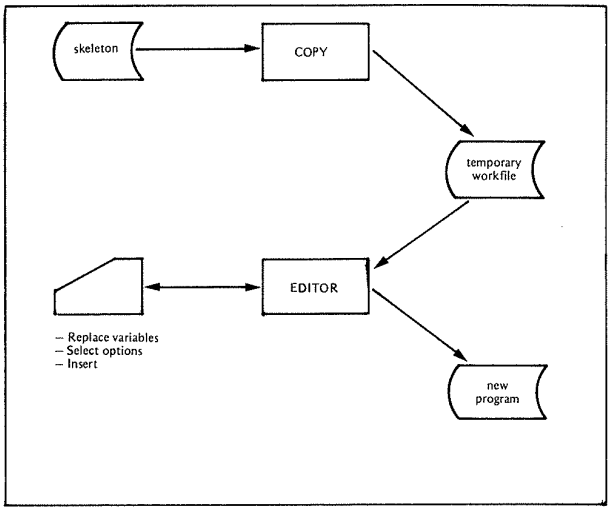

Figure 2. Copy-a-Skeleton-and-Amend.

achieved by the marking of optional lines as active or commented out;
- define the points within the skeleton at which programmers will under particular circumstances need to insert additional code.

The development of a program using such a skeleton comprises Figure 2:
- selection of the appropriate skeleton;
- copying to a new file;
- the replacement of the variables;
- choosing the appropriate options;
- inserting additional lines particular to that program.

The scale of the effort involved varies widely. In the author's experience a file handling sub-program requires about six variables, no additional code, and about five minutes' work. For a reasonably flexible on-line master file maintenance program about 35 variables and 15 options were needed. The number of insertion lines varied directly with the amount and complexity of validation – between 50 and 2000 lines – giving a total development time between two hours and four days. If the average line rate seems high (600 lines/hour for simple programs, 100 for the more difficult ones), it should be recalled that this code is composed almost entirely of editing instructions directly translated from the specifications and containing virtually no control structures.

Advantages of this approach as compared with conventional programming are:
- experience has been invested in the skeleton, and is directly transferred to each program;
- less time is required to prepare the program;
- the new program requires testing only of the program-specific code (assuming that the particular combination of options was tested as part of the skeleton's development), hence less time is required to achieve a clean program;
- the resulting program's style is dictated by the skeleton.

There remain deficiencies:
- the selection of an appropriate skeleton for the task depends on criteria that are rarely fully understood;
- investment in some amount of abstract theorising and experimentation is a precondition of success. Installations which oppose abstraction *per se* and limit their techniques to those taught by their equipment and software suppliers are therefore ill-served by this method. It requires confidence on the part of the installation management that they can manage the risks involved;
- a sufficiently large volume of programs of each type is necessary to justify the investment. In the author's experience a breakpoint was already reached with three or four programs, but that is sensitive to the skeleton builder's experience in and flair for both skeleton building and the program types;
- an on-line development environment is essential, with suitable supporting software, in particular a full-screen editor with string replacement and line insertion capabilities (Clarke, 1982a);
- no continuing relationship exists between the skeleton and the programs produced from it. Subsequent corrections and improvements to the skeleton can only be included in each of its progeny by painstaking effort.

Efforts to overcome this last deficiency lead to the third phase.

PHASE 3 – SIMPLE PROGRAM GENERATORS

Once skeletons have been established it becomes attractive to have the benefit of the maintenance of those skeletons flowing more-or-less automatically to its progeny. The classes of maintenance include error correction (a skeleton is, like any program, 'clean' only until the next bug is found), efficiency improvement, the adaptation of existing facilities to new standards and to new run time environments, and the provision of additional features.

The step required to link programs to their skeleton is to store the instructions used in their preparation, and regard these rather than the generated high level language code as the source program. As Figure 3 depicts, the instructions to be stored comprise the assignment of values to variables, the selection of options and the insertion of additional source lines. A utility program is required to

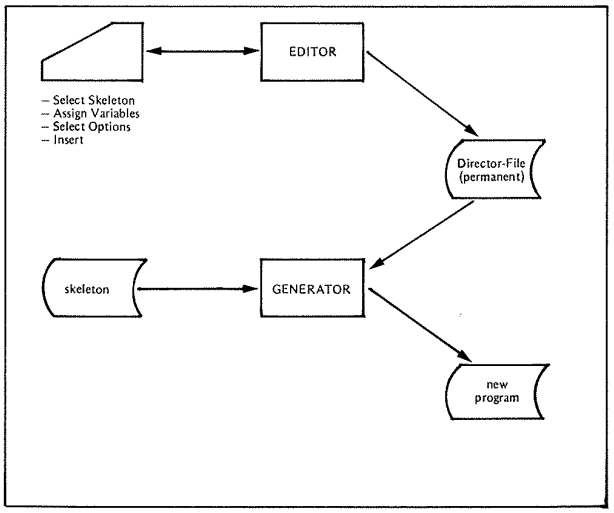

Figure 3. Simple Program-Generator.

merge the skeleton with the additional source lines, carrying out the (global) variable replacement and option setting as it goes. Such a utility is popularly termed a program generator; for the input I will use the term 'director file'.

The development of such a generator requires string handling capabilities. Nonetheless implementation even in COBOL requires under 10 days, and installations with expertise in more suitable languages should require yet less effort. Assuming that a small collection of 3-4 skeletons is created, then the breakeven point will be of the order of only two or three uses per skeleton – a point reached or reachable in almost any single new application.

An additional investment involved is the formalisation of the skeletons. A Phase 2 skeleton can contain loose comments of the form 'if both options A and B are selected, then datafields X and Y must be OCCURed twice, with consequent changes in Procedures P and Q'. This may have been the most cost-effective solution in Phase 2, but cannot be tolerated once a generator is implemented. Such problems are quite soluble, but require disproportionately high investment. (The pragmatic solution is to add this condition to the list of variants unsupported by the generator: 'For Priority Release' as the sales brochures say.)

When a skeleton is revised, all that is necessary to pass the revisions to its progeny is to re-run the generator against the director file. Late amendments in the file handling technique or the user interface no longer justify fears of excessive rework costs and delays.

Some limitations must be recognised of course:

- considerable unanimity is required as to what constitutes good programming style and appropriate program structure;
- the preparation of suitable skeletons requires familiarity with a wide range of program types as well as the ability to abstract;
- machine overhead is incurred by the generation run. The programs require far less testing, but the net effect is hard to measure and correspondingly easy to argue about. On a small software development installation (TI 990 with 256kb memory and five screens) the generator required about three minutes (elapsed) for a 500 line director file and a 1500 line skeleton. This compared favourably with the compile time of the generated program, despite the inefficiencies of COBOL string-handling;
- subsequent amendments to a skeleton must be made with rather more care than with a Phase 2 skeleton. It is necessary to generate and test first that program for which the change is required, then a range of sample programs appropriate to the population of the progeny, then all of the progeny;
- in addition to the normal 'where used' capabilities needed for copyfiles, datafiles and subprograms, the use of the skeletons themselves must be monitored. This is most easily achieved if the invocation of the skeleton is controlled from the director file itself;
- the use of an existing skeleton for a new project often involves additional investment. (Typically the original version assumed only one record type per file, while the new project must handle two or more.) Generally it seems better to allow skeletons to proliferate rather than invest too much too soon chasing the chimera of 'truly general' master programs;
- the method decreases the creativity involved in applications programming. Other sources of programmer job satisfaction must be substituted for that lost if low morale and high turnover aren't to rob the installation of the potential productivity gains;
- a stratification, or at least segmentation, of programming staff results, with differentiated education, experience and even psychological profiles. Given that the existing distinctions between systems and applications programming groups can result in friction, the addition of a 'methods programming' group could be an unwelcome additional ingredient in the political cauldron; and yet most systems programming staff are ill-suited to the work involved because of its strong applications orientation;
- the method invites the naive application of an inadequate tool to a different or more subtle problem. It is essential that in seeking productivity improvement we do not force development staff into under-investment in the problem comprehension and design phases and thereby trivialising their appreciation of the application.

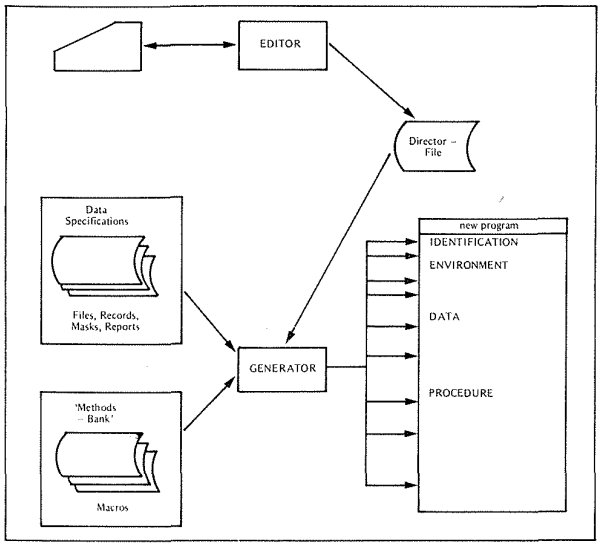

Figure 4. Sophisticated Program-Generator.

PHASE 4 – SOPHISTICATED PROGRAM GENERATORS

The 'merge-and-replace' type of generator remains trapped within the conceptual boundaries of its host language. There are two very important and related limitations that can be overcome only if the framework of the generation run is changed.

The sequential processing of a single skeleton has the result that a skeleton must resemble the program that is to be generated, with the exception that some symbols appear which would not be valid input to a compiler, some denoting locations for insertion, others awaiting replacement: the skeleton and the generated program are synchronous.

The other limitation is that in a family of skeletons there will be a considerable amount of redundancy. In particular, file definition and file access routines will appear not merely in each skeleton, but even several times in each. It is desirable that code which is common to multiple skeletons be stored once only, in an independent sub-skeleton.

The instance of file handling is particularly impor-

```
.PROG-DEMO1, AUTHOR=ROGER
```

generates:

```
000100   IDENTIFICATION DIVISION.                    01000064
000200***************************                    01000065
000300   PROGRAM-ID.            DEMO1                01000066
000400   AUTHOR.                ROGER.               01000068
000500/                                              03000003
000600   ENVIRONMENT DIVISION.                       01000081
000700***************************                    01000082
000800   CONFIGURATION SECTION.                      01000083
000900   SOURCE-COMPUTER.    PRIME 550.              01000085
001000   OBJECT-COMPUTER.    SYCOR 585.              01000086
001100   INPUT-OUTPUT SECTION.                     23 DELTA
001200   FILE-CONTROL.                             23 DELTA
001300   I-O-CONTROL.                              23 DELTA
001400/                                              03000005
001500   DATA DIVISION.                            23 DELTA
001600   FILE SECTION.                             23 DELTA
001700/                                              03000007
001800* ------------------------------------------- 23 DELTA
001900   WORKING-STORAGE SECTION.                  23 DELTA
002000/                                              03000028
002100* ------------------------------------------- 23 DELTA
002200   PROCEDURE DIVISION.                       23 DELTA
002300   DX-MAIN SECTION.                          23 DELTA
002400   O-PROG.                                   23 DELTA
002500   P-PROG.                                   23 DELTA
002600   C-PROG.                                   02 DELTA
002700   STOP-RUN.                                 02 DELTA
002800       STOP RUN.                              01000131
```

Figure 5. The Minimum-Complexity Program.

```
. PROG-DEMO2, AUTHOR=ROGER
. SL=P-PROG
      DISPLAY "HELLO, USER! WHAT'S YOUR NAME?".
      ACCEPT WS-NAME.
      DISPLAY "CONGRATULATIONS " WS-NAME "!!!".
      DISPLAY "YOUR PROGRAM WORKS ALREADY!".
. SL=WORK01
  01   WS-NAME          PIC X(OB).
```

Figure 6. A Slightly More Complicated Program.

tant, because yet a further level of abstraction exists. In order to facilitiate the portability of applications software between differing machines, compilers, and file handling environments, it is necessary to store those parts of the program which are environment-dependent in separate 'sub-sub-skeletons' which can be exchanged in order to generate a new version of the application to run on, say, an interstate branch's much smaller and perhaps separately sourced installation. (The same problem occurs in relation to the handling of on-line terminals, although defining the interface between the logical and the physical sub-skeletons is made much more difficult by the absence of de facto standards.) See Clarke (1982b, 1982c) for further discussion of such matters.

It is not difficult to restructure the simple generator described in the previous section to include invocations of sub-skeletons, depending on some condition in the director file or the main skeleton. The problem is that the synchronisation between skeleton(s) and generated program is destroyed. In the case of a file handling sub-skeleton, the sub-skeleton will endeavour to insert code in a location (say the file access routines), while the skeleton still contains code that must be inserted at an earlier location.

The requirement is, then, that the director file be read sequentially, resulting in invocations of sub-skeletons, and the 'assembling' of an output file. The output file must

```
. ADD TESTMAC, 5, 2, N
```

invokes this Macro:

```
**PDL*8112311159/TESTMAC/02/  TEST-MACRO
.*
.*    Converts an alphanumeric field with contents in the form
.*    '9999.99' into a numeric field of the form 9999V99
.*
.*    The number of digits is freely-choosable.
.*
.*    Parameters:  01 — number of whole-digits
.*                 02 — number of decimal-digits
.*                 03 — whether a subroutine
.*                      is to be created (Y/N)
.*
.* ------------------------------------------------
. SL=WORK01
  01   WS-ALPHANUM- #01 #02.
       05   WS-AN- #01 #02-WHOLE     PIC 9( #01).
       05   FILLER                   PIC X.
       05   WS-AN- #01 #02-DECIMAL   PIC 9( #02).
  01   WS-NUM- #01 #02.
       05   WS-N- #01 #02-WHOLE      PIC 9( #01).
       05   WS-N- #01 #02-DECIMAL    PIC 9( #02).
  01   WS-NUM- #01 #02REDEF          REDEFINES WS-NUM-
                                      #01 #02 PIC 9( #01)
                                      V9( #02).
. IF-03. EQ. Y
. SL=SUBROUTINES
  CONV-AN- #01 #02.
       MOVE WS-AN- #01 #02-WHOLE    TO WS-N- #01 #02-
                                      WHOLE.
       MOVE WS-AN- #01 #02-DECIMAL TO WS-N- #01 #02-
                                      DECIMAL.
  CONV-AN- #01 #02-EXIT.  EXIT.
*************************************************
. IFEND
.* ------------------------------------------------
```

to generate this code:

```
WORKING-STORAGE SECTION.
   .
   .
01   WS-ALPHANUM-52.
       05   WS-AN-52-WHOLE       PIC 9(5).
       05   FILLER                    PIC X.
       05   WS-AN-52-DECIMAL     PIC 9(2).
01   WS-NUM-52.
       05   WS-N-52-WHOLE        PIC 9(5).
       05   WS-N-52-DECIMAL      PIC 9(2).
01   WS-NUM-52REDEF             REDEFINES WS-NUM-52
                                      PIC 9(5)V9(2).
```

Figure 7. Macro-Calls.

be addressable at multiple points, not merely at the (current) next record (a partitioned or segmented sequential file as distinct from purely sequential). For flexibility sub-skeletons should be able to be invoked conditionally, and iteration, nesting and even recursion should be possible. In addition parameter passing between different elements must be facilitated. Such a generator is complex, requiring modular construction for reliability, maintainability and extendability, and involving the investment of man-years of effort. Figure 4 depicts such a generator.

Examples of products which offer at least some of the requirements are: CPG, an American product of the late 1970's; CL/1, an Australian product released in 1979, MANTIS from CINCOM (IBM-specific, 1979); NoCode, an American product (1980); and the cutely-named 'The Last One', a UK product (1981). ADR's IDEAL is overdue for release. Philips' PET/X1150 development-machine incorporates generator-elements.

TABLE 1. Properties of Program-Generators.

The Product Provided
- capable of immediate use without initial investment by the user
- based on an interpretative language so as to be portable between software environments and machines
- includes standard macros for common functions which can serve as a starting point for the integration of the product into the user's particular environment
- is consistent with and capable of operation in parallel with other development environments, and in particular with the maintenance of existing applications by conventional methods
- is suitably documented and the documentation is well indexed
- education and introductory documentation are provided
- maintenance and support are provided
- on-going development of the product is guaranteed
- version control and upwards compatibility are assured

The Macro-Language
- standard macros are under user control
- additional macros can be written by the user
- offers DO-verb, and complex conditionals or decision table
- DO-verb and conditionals are nestable to an adequate depth
- offers computational and string handling capabilities
- capable of passing parameters
- parameters may be local or global, and 'typed'
- simple file reading capabilities
- additional locations can be defined
- all locations are accessible by any macro

Program-structure skeletons
- ability to generate the vast majority of program structures with simple parameterised invocations
- all control code for level breaks should be generated
- appropriate locations for insertion of program specific code
- the resulting code should be suitably modular and structured (within the constraints of the generated language)
- ability to specify exotic program structures in a convenient, auditable, powerful but compilable language

Outputs
- generates an industry standard language(s)
- is sufficiently flexible that variants within the standard, not-quite-standard and add-on compiler features can be handled
- generates code that is consistent in style with the prevailing installation standards no matter from which skeletons/macros it may be generated. This is important during the first years following its installation, since maintenance may be performed on the generated programs rather than the original source
- the code generated by all methods is consistent in apppearance
- generates documentation as an integral part of the code
- generates a where-used listing for macros/skeletons
- can generate skeleton JCL for testing and production purposes

Use
- simple to use for simple programs, in particular a simple report generator syntax such that trainees can quickly become productive and experience early positive feedback
- powerful for larger and more complex programs such that the productivity of experienced staff is significantly enhanced
- consistency of use for each type of standard program (i.e. the preparation of simple print programs, simple batch, complex batch, on-line enquiry, data capture and update programs should not differ more than is necessary)
- 'naturalness' of the language and its syntax, rather than obscure mathematical script
- completeness of syntax validation
- clarity of error messages
- accessibility of the documentation for reference purposes
- the capability to reflect user modifications and extensions

Mode of Processing
- can access multiple macros, including many level nesting and perhaps also recursion
- adequately efficient in its usage of machine resources (run time, file access, main storage)
- written in reentrant code and is actually shareable by many users
- capable of concurrent execution by an effectively unlimited number of users, e.g. suitably qualified workfile names, macros accessed in read-only mode
- allows definition of reference libraries and documentation options at run time

Interface to its Environment
- ability to mesh with techniques used within the organisation (structured analysis, structured design, Relational Analysis, HIPO, structograms *a la* Nassi and Schneiderman, decision tables Jackson or Warnier Program Design Methodology, structured programming, etc.)
- interface with Data Dictionary software
- interface with formalised system requirements and system design utilities
- interface with screen definition facilities
- interface with report layout facilities
- interface with project planning and control
- interface with the testing and debugging facilities
- independence from its host machine, i.e. runs on many machines (and in principle on any machine)
- independence from its target machine(s)
- independence from supplier specific environmental software (operating system, file handler/database, languages, on-line monitor, data communications monitor, etc.)

The author has experience of a Swiss product, DELTA (see Clarke, 1982b, 1982c), which fulfils the requirements. It has enjoyed considerable success in German-speaking areas, and is available in both Britain and Australia. It had the market to itself following its market release in 1976, but a small flood of competitors is lining up to do battle. The generator package comprises an interpreter, a set of 'processors' (providing efficient performance of the most common facilities such as the basic program shell, and decision table and pseudo-code interpretation), a range of standard macros (providing file handling, a report generator, etc) and a macro language to enable the writing of further macros.

The distinction between a skeleton and a macro is important. A skeleton contains no control structures; the director file drives the run, but the generator itself performs all the decision-making. In the case of a macro the stored code is not just passive, but contains selection and iteration decisions, based on parameters supplied in the director file, and additional variables computed during the generation run.

This language is available to the software developer, so that he can go further than merely amending existing macros: he can also develop his own to match the requirements of the installation. The following examples of the use of a Phase 4 Generator are based on DELTA, because of the author's familiarity with that product, but also because it embraces all of the important concepts and mechanisms.

EXAMPLES

Figure 5 depicts the preparation of the minimum complexity program. The basic Processor is invoked using the command .PROG; a variety of optional parameters may be set. The result is a program shell containing the minimum set of commands consistent with the particular target compiler. The precise content of the generated shell is determined by macros supplied by the vendor but fully under the using organisation's control.

In addition so-called 'locations' are created into which lines of high level language code can be inserted. Each location is accessible in 'open-extend' mode, i.e. lines

```
. PROG-CUST, AUTHOR ROGER, WRITTEN JUL 81
. SL=REMARKS
*
*         ON-LINE DEMONSTRATION-PROGRAM (CUSTOMER
          FILE-MAINTENANCE)
*         ----------------------------------------
*
. *             Create program-structure:
. *
. ADD OLSTRUC, 1, (DSP, CRE, AMD, DEL), -
.                 (MSKCUST1, MSKCUST2)
. *
. * ----------------------------------------
. *
. *             Validation-code:
. *
. SL=VAL-01-DEL
. *
. *             Delete prohibited if current or previous year's
. *             Sales are other then zero:
. *
      IF    T01-SLSYTC = ZERO
      AND T01-SLSYTP = ZERO
          NEXT SENTENCE
      ELSE
. ADD VALERROR, 905, , SLSYTC
. * ----------------------------------------
. *
. *           Define Customer Logical-Record:
. *
. ADD LR-CU, UPDATE-ONPLACE, 1
. *
. * ----------------------------------------
```

The above depends on data definition files (which are the responsibility of the applications team), about 10 standard macros, 5 additional macros written and maintained by the installation standards team, and about 10 macros which generate the program structure and screen handling.

Figure 8. An On-Line Program Using DELTA.

are loaded successively into that slot. The process is directly analogous with a box of 80 column cards in which the permissible insertion points are marked with thick cardboard. Each new card (including new markers) can be inserted immediately before any marker. Figure 5 in itself cleanly compilable, although its execution would cause little excitement. Very slightly more interest would be aroused by the program generated by Figure 6, in which two locations have been used, that for basic processing, and the basic working storage location.

Figure 7 illustrates the next conceptual step, the invocation of macros. Great power can be achieved in the use of pre-written code through the nesting of macros. For example the author uses a single line invocation (together with separately prepared mask definitions) to generate an on-line update program with inquiry, creation, amendment and deletion capabilities, any number of masks and some 30 locations into which the more complex validation and file handling code can be inserted (Figure 8). The additional coding is also strongly supported by additional macros.

A hierarchy of self-supplied macros is one of a range of ways in which the program structure can be generated. A processor is supplied for normal batch processing programs, another generates structures in a manner consistent with Jackson's Program Design Methodology, and an interpreter is available to generate more exotic forms from a structured 'pseudo-code'.

A number of processors are also supplied as part of the basic product to achieve run time efficiency in the handling of certain standard functions. Chief among these is the File Processor which, with the aid of one or more macros generates all code necessary for definition of and access to each file. It also includes facilities for integrating the file processing into the program structure. In COBOL this involves entries into at least the following locations: SELECT, FD, RECORD-DESCRIPTION, OPEN, CLOSE, File handling Subroutines and the calling of the file access routine(s). Macros for the various file types are supplied, and can be used in that form or extended to suit the user's particular requirements.

A further point of importance about Figure 8 is the machine-independence of the DELTA source file. It was written and tested on a PRIME 550, then re-generated on that machine in the form appropriate for a SYCOR (Data 100) Model 585. Differences between the file definition, file handling and (very differently conceived) screen handling methods were catered for with little difficulty. Implementation of precisely that program on further machines involves the preparation of file and screen macros appropriate to the new target machine and/or target environment. Clarke (1982b) discusses this example at greater length.

PROPERTIES OF PROGRAM GENERATORS

Table 1 contains a list of factors to be considered when assessing alternative products or designing one's own. Since this article is tutorial rather than analytical this point is not discussed further.

IMPACT OF PROGRAM GENERATORS

The benefits brought by a sophisticated program generator include the faster development of cleaner products, quicker and more reliable maintenance and enhancement, the opportunity for genuinely portable applications, and shorter lead times for trainees.

The development process, the organisation of development teams, and the organisation and operation of the supporting 'methods programming' team are significantly affected.

TOWARDS APPLICATION GENERATORS

The focus of this article, and indeed of the products which it discusses, is the generation of independent programs. The design of a collection of programs to fulfil a complex of purposes is viewed as a separate exercise. In order to generate an entire application from an application specification, a logically complete and precise statement of the requirements would be needed in a set of consistent and compilable syntaxes. Implementation parameters (e.g. the physical allocation of records and the gathering of functions into programs) would also be required.

CONCLUSION

In the near future only specialist 'methods programmers' will deal at the level of detail of present high level languages. The vast majority of commercial development will be done by programmer coders using utilities to capture the parameters for input to program generators.

In the period 1982-1987 many of these generators will be machine specific, generating a special language code, and be subject to myriad intended and unintended restrictions. Later more of them will achieve substantial machine independence and generate industry standard languages. A very few such second generation products are already on the market.

Somewhat further in the future it seems reasonable to anticipate effective application generators which will operate on one or more system design languages to produce executable code directly.

ACKNOWLEDGEMENT

The assistance of colleagues at Brodmann Software Systeme AG, and of Herrn Peter Buchmann, Claude Reibel and Dr. Reinhold Thurner of SODECON AG, the suppliers of DELTA, in the preparation of this article is gratefully acknowledged.

REFERENCES

ADR (1974a): *Macro-Writing for the MetaCOBOL User*, Doc Nr P502M, Applied Data Research, Princeton, NJ.

ADR (1974b): *Macro-Writing for the MetaCOBOL Specialist*, Doc Nr P551M, Applied Data Research, Princeton, NJ.

CLARKE, R. (1982a): Editors for Software Development, *Aust. Comput. Bull.*, 6, Feb 1982, pp. 21-25.

CLARKE, R. (1982b): Generating Self-Contained On-Line Programs Using DELTA; submitted to the Ninth Australian Computer Conference, Hobart, August 1982.

CLARKE, R. (1982c): Generating Transaction-Oriented On-Line Programs Using DELTA, submitted to the *Aust. Comput. J.*

HAWRYSZKIEWYCZ, I.T. (1981): 'Some Trends in System Design Methodologies', *Aust. Comput. J.*, 13, Feb. 1981, pp. 13-23.

WAITE, W.M. (1974): *Implementing Software for Non-Numeric Applications*, Prentice-Hall, NJ.

BIBLIOGRAPHICAL NOTE

The author has been active in commercial data processing since 1971 in functions ranging from systems analysis and project-leadership through research into the privacy implications of the information industry, to the technology of software development. He has worked for and with a variety of industrial, commercial and consulting organisations, including more recently 1½ years with The Stock Exchange, London, and three years with a software house in Zurich. He completed an M.Comm. (Accounting and Information Systems) in 1975 following nine years' part-time study at the University of New South Wales.