Program Generators and Generation Software

P. A. LUKER AND A. BURNS*
Postgraduate School of Computing, University of Bradford, Bradford, West Yorkshire BD7 1DP

Program generation is a technique that is becoming increasingly popular, although many of the products so far developed have taken a very ad hoc approach. A definition of program generation is given together with an assessment of its applicability and a discussion of the basic structure of a program generator. An architecture is proposed. Considerations of dialogue design are given together with a number of operational criteria aimed at improving the user interface.

Received March 1984

1. INTRODUCTION

Recent years have seen a marked increase in the availability of computers, thanks largely to the dramatic cost reductions brought about by LSI and VLSI technologies. The potential user population has increased proportionally. Education has, however, failed to keep pace with such progress and there exist many people who now have access to some computer system, requiring the machine for some specific task, but neither have the time nor possess the skills to produce the software. The main skills necessary to produce any program can be summarised as follows:

(i) the ability to define the problem;
(ii) the knowledge and experience to define solution algorithms and associated data structures;
(iii) the knowledge and experience to code, test and debug the required program.

Until recently, people without the above skills, if they have been allowed near a computer at all, have had to make do with standard packages. Used in the right context these are valuable, but they are often inflexible, in that the user can exercise little control over what they do. Within their area of application, packages are usually designed to be generally applicable. There is an increasing need to change this position by providing the means to customise software to the users' requirements. One way of achieving this is to generate programs which are specifically suited to solve individual users' problems. The process of program generation is, however, non-trivial, and considerable investment, in terms of design and development, is necessary to produce a reliable and useful product.

Program generators have been in use for some years in specialised areas, such as discrete event simulation,[1,2,3] and program generators have been developed that enable computationally naïve users to perform continuous system simulation.[3,4,5,8] Commercially produced program generators, which perform many of the standard functions in data processing, are now on the market. From our experience of program generators, we have found many apparent advantages which are not restricted to the naïve user. It is also clear that there is no reason why program generation should be restricted to narrow application areas such as simulation.

For whatever application a generator is intended, many of the techniques used in its construction will be similar, if not identical, to those used in other generators. It would seem sensible, therefore, to try and economise in effort by identifying such areas of commonality. For example, this realisation has stimulated our work on general dialogue development systems.[19,20]

Our aim here is to draw on our implementation experience in addressing program generation in general. Many of the points discussed are illustrated in an example of a dialogue from the simulation program generator already referred to.

Although a user-friendly and well-structured interface will render some assistance in problem formulation for the inexperienced user, it is important to emphasise that this type of software will be of little use to anyone with only a vague problem definition. It is the user's responsibility to collect (and understand) all information required for the problem specification.

2. PROGRAM GENERATORS

A program generator (PG) is a piece of software which produces a program in some object language, which is tailor-made to fit a specific set of requirements. This description, however, is somewhat imprecise as it would equally apply to any compiler or assembler. The input to both a PG and a compiler is a specification of the user's requirements; for a compiler, though, this input comprises a set of statements each of which must conform to rigid rules of syntax. To make a PG attractive to the user, considerable freedom of expression must be allowed. However, it is also essential for the PG to exercise some control over the way in which a problem is specified, in order that inconsistencies and omissions can be trapped. Input to a PG must, therefore, take place via a dialogue with the user in which, within the imposed structure, the user may respond in a fairly free format.

The output from a compiler is (usually) low-level object code, whereas for reasons given later, PGs normally produce high-level language output. From these characteristics it is clear that program generators may be indistinguishable from pre-processors. Many continuous-system simulation 'languages' are, in fact, FORTRAN pre-processors. Strictly speaking, the term 'program generator' should apply only if output can be produced in more than one language (or dialect); however, it is now being widely applied to what are essentially high-level language pre-processors. For our purposes the term will be confined to the following strict definition. *A program*

* To whom correspondence should be addressed.

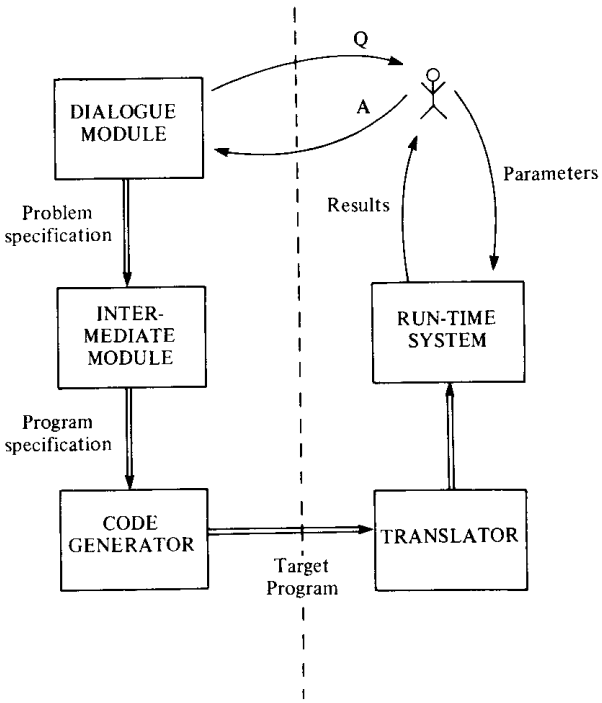

Figure 1. Outline of a program generator.

generator is a program which accepts as input human structured dialogues and produces, in a range of high-level languages, a well-structured and commented program which is tailored to meet the user's particular specification and requirements.

Applicability of the PG approach

A program generator consists of three major components, which are quite distinct: the dialogue module, an intermediate module and the code generation module. The relationship between these modules and the user is outlined in Figure 1. The dialogue module obtains from the user a specification of the structure of the problem to be solved. This structural definition is then used by the code generator to produce the corresponding target program in the desired language. This transformation may make use of an intermediate module (the architecture of a program generator is discussed in more detail in the next section). The target program is then submitted to the appropriate translator together with any data required by the program to obtain the desired results.

In order to assess the suitability of program generation it should be contrasted with alternative approaches which include:

(i) special-purpose programs written in a suitable language;
(ii) very general programs designed to cater for all likely requirements in a given application area;
(iii) fourth-generation languages.

Special-purpose programs bring us back to the initial premise that our naïve user does not possess the knowledge, time or skill needed to write a suitable program. Assuming that the user can specify the problem accurately and communicate this information to a programmer, then it is possible to obtain a program tailored to the user's needs. A program generator is designed to automate this process. It will produce syntactically correct programs and do so at a lower cost than conventional methods, if there is a significant number of applications within the one area. Moreover, by having the dialogue of the PG designed by an 'expert' in the field to which it will apply, rather than a programmer ignorant of this field, a more applicable and appropriate system should ensue.

Large general-purpose programs may have large sections of redundant code that artificially restrict the size of machine on which the application software can run. They may also involve the user in unnecessarily lengthy data entry. There is a clear distinction in this type of software product between parameters that a particular user or application may dictate and data which will change with each execution of the program. A PG requires the parameters to be specified only once, a general-purpose program may need these to be input on all occasions, although it is not fundamentally necessary.

Fourth-generation languages (FGL) are a recent development, and they are also aimed at increasing the productivity of the non-expert computer user. For example HOS[6] will produce software in a number of computer languages (COBOL, FORTRAN, Pascal) and supplies the users with a high-level language more geared to their experience and knowledge. PGs employ a *dialogue* to help users specify their particular requirements for some (previously designed) software product. FGLs, by contrast, are used to design and construct complete software systems; they provide the user with a *language* which, although flexible, must still be understood by the user.

By generating a program in a high-level language, all the features of that language can be utilised, thus obviating any temptation to duplicate them. Taking continuous system simulation as an example, there already exist many languages which are more than adequate for most modellers' requirements, providing such facilities as a choice of good integration routines, automatic sorting of the model equations, graphical output and many relevant functions. It would involve a sizeable investment of time and effort to incorporate such features into a general program.

Care must be taken, however, with the choice of run-time system and translator, as execution errors, while their incidence may be reduced, can never be totally avoided. The naïve user should never be presented with 'meaningless' run-time diagnostics. Where possible, the logical names that the user has chosen for variables should be incorporated into the program. Languages with severe restrictions upon their identifier lengths should be avoided where at all possible. A program constructed by a PG is made more flexible by permitting the user to supply data to this target program during execution. The dialogue must, therefore, distinguish between those inputs which will be used to define the parameters to that program. However, the program generated is, ultimately, only as good as the compiler used.

From these (and other) observations, program generators would appear to be a suitable vehicle for several types of user and a number of application areas. These (not necessarily mutually exclusive) circumstances may be summarised as follows.

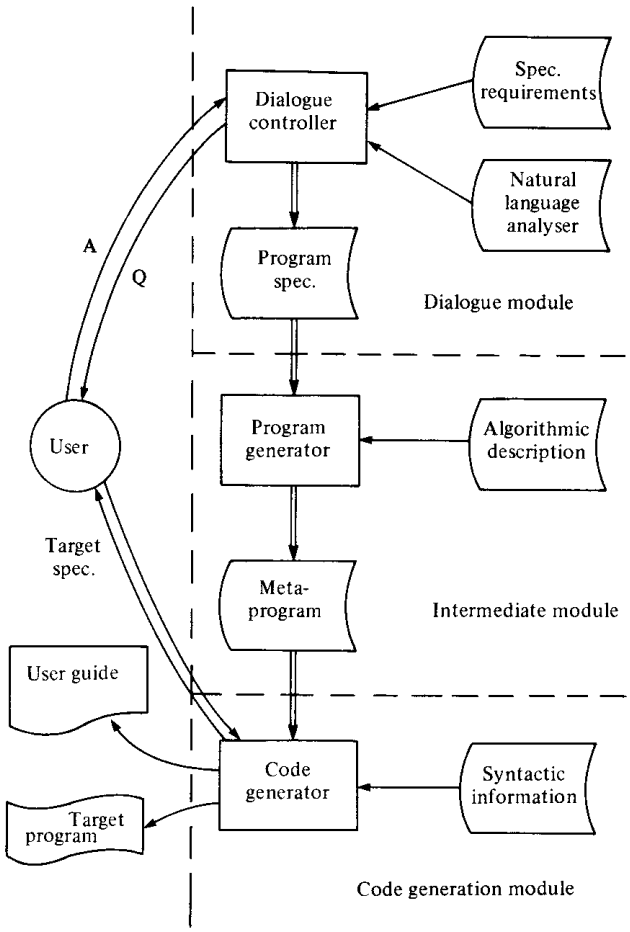

Figure 2. Detailed structure of a program generator.

(i) For people with insufficient programming expertise.

(ii) For more experienced programmers, as a means of generating syntactically correct programs for use either as generated or as a skeleton for fleshing out by the programmer. PGs have been found to be attractive to experienced users, but only if the dialogue is not too verbose.

(iii) Where a more general program would be too large and/or inefficient for implementation on a target computer.

(iv) Where a more general program would include ill-defined data structures which do not relate specifically to a user's problem. Debugging of this might be very difficult. For example, the use of a record structure which incorporates fields of which the user has no knowledge may cause problems. A generated program would have these fields removed.

(v) Where the user requires the program to run on different target computers, which have different parameters such as word-length and memory capacity. The problem need only be defined once.

(vi) Where the user requires more than one version of a program, either in different languages or in dialects of the same language. As with (v), the user can obtain these different codings from one problem specification.

3. STRUCTURE OF A PROGRAM GENERATOR

The major components of any program generator are illustrated in Figure 2. Not all the elements would necessarily be required for every generator, and different generators might need additional, specifically adapted components. Three main modules are identified here; the dialogue, intermediate and code-generating modules as mentioned in the previous section. In a number of applications the intermediate module can be omitted.

The Dialogue Module

The dialogue controller interacts with the user in order to obtain a complete and consistent specification of the problem, which will then form the program specification. The dialogue is conducted according to the information contained in the specification file. All user response is analysed by the dialogue manager (DM), the sophistication of which may vary considerably. Many dialogues, for example, will only require the user to select answers from a menu or respond in a simple Yes/No fashion; others will require the input of more complex expressions, such as arithmetic equations, the validity of which can only be tested by the application of domain-dependent rules.[7] More general program generators, allowing some freedom of expression, will be much more demanding of the DM, perhaps requiring a natural language analyser and all the complexity (and problems) that implies. However sophisticated, an important feature of the DM will be to distinguish between genuine user responses and requests for help or explanation. The degree to which a program generator is used will depend on the success of the dialogue design, in particular the appeal of the dialogue.

When the dialogue has been completed, the problem (and therefore the program) specification is also complete. The user should then be free to use this specification repeatedly in order to generate programs. This naturally requires the 'permanent' storage of the program specification file.

The Intermediate Module

The operation of this module is somewhat analogous to that of macro expansion with, here, the program specification being expanded by the inclusion of any algorithm necessary for the target program. The meta-program resulting from this unit will be in some meta-language which may, in fact, be an existing high-level language. Use of an actual programming language could, however, lead to certain inefficiencies, particularly if the target language is very dissimilar in nature. The meta-program should be easily readable both by human and machine, and should concisely describe the operations to be performed by the program. If an actual programming language cannot be used then it is envisaged that the meta-language would contain structures very similar to those in, say Pascal or Ada.

A further function of the intermediate module is to simplify/optimise the meta-program in the light of the specific information contained in the program specification. The algorithmic descriptions will be defined in terms

of parameters, the actual values of which will be obtained from the dialogue. Wherever possible, these parameters will be constrained within tested subranges; the first function of the program generator is therefore to test the program specification to see that it contains only valid values. For many descriptions, particular parameter values will greatly simplify the required algorithm, and this is reflected in the meta-program. For example the loop:

for *I*:= 1 **to** *parameter* **loop**
⟨*sequence_of_statements*⟩
end loop;

can be replaced simply by the *sequence_of_statements* if the parameter is given the value 1. More significantly, if it takes the value 0, the complete structure can be removed from the generated program.

If the intermediate module is omitted, then the program specification becomes the meta-program and the external subroutine would have to be introduced to the code generator module. In such cases the algorithmic content will probably be fairly low and involve the insertion of complete and invariant subroutines into the program rather than integrating the algorithms within the program text.

The Code Generation Module

Having taken in data from either the program specification or the meta-program, the function of the code generator is to produce two output files – the required target program and bespoke documentation to assist the user in understanding and running the program. This module also requires information, provided by the user, which specifies the target environment, in particular the target language and target machine. It would be possible for the user to include this information in the dialogue module, as part of the program specification. Greater flexibility is possible, however, if the user is asked for details concerning the target implementation by the code generator. This approach, though, is not ideal, as routines which are used by the dialogue module to process users' responses will also be required to analyse the implementation details input to the code generator.

Bespoke documentation is produced using a similar tool to that employed to construct the meta-program. In particular, parameters are replaced by actual values, and only those routines actually used will be documented. Cross-referencing between the program and documents via line numbers will be maintained consistently. Finally, information appertaining to the target environment and language will be selected.

The chosen language may not be able to represent a particular algorithm, or the requested hardware might be unsuitable because of, for instance, an inadequate dynamic range of number representation. As the separation of the problem and implementation details is felt to be both logical and desirable, any mismatch of problem to implementation would reflect an inappropriate choice of implementation on the part of the user and would not imply that the problem is inherently insuperable. It is preferable, however, that in the dialogue module all constraints on usage are expressed fully in order to give the user the opportunity to amend the specification. In this way the code generator could be driven by a simple menu of admissible targets.

Although the operation of a program generator is shown in Figure 1 as a mapping from dialogue to executable code, for many users in a number of applications the program itself is an important intermediate stage. Experienced programmers may use PGs to get close to the desired program and then add to this code particular functions that the generator is not able to provide. For these reasons it is important that the program which emanates from the code generator is well structured, adequately commented and laid out in a readable form.

Ancillary Problem Specification Module

One deficiency of many program generators is the inability of the system to save problem specifications which were only partially complete. This is especially limiting with problems which involve long specifications; it is unreasonable to expect users to be able to complete the entire specification in just one sitting. It is equally unreasonable to expect then to re-enter an entire specification in order to make only minor amendments. Consequently some kind of editing facility is required.

There are two obvious routes which can be followed – an editor can be incorporated into the dialogue module or the user could apply a standard system editor to the program specification file. The latter option is not satisfactory for inexperienced computer users – the very people for whom program generators are primarily intended. Therefore, the dialogue module should contain an editor which would read the program specification from a file and allow the user to make any changes or additions as necessary. The editor should be very simple, to suit the user and prevent the software from becoming unwieldy. Considerable care has to be taken over the inclusion of any amendments to the problem specification to prevent the introduction of inconsistencies caused by conflict with unaltered portions. During an editing session a point may be reached at which the remainder of the original specification becomes invalid, and it is important that the main dialogue software resume control of the problem specification. Examples of a simple editor and its interaction with the dialogue module are given by Luker.[8]

For experienced users, the option of using a system editor for amendments might prove more attractive. Knowing the 'language' of the problem specification would also enable the expert to 'program' the specifications directly rather than via the dialogue. This is not necessarily a desirable or advisable feature owing to the greatly increased scope for errors. Experience with such users has shown the value of re-analysing the problem specification for errors or inconsistencies whenever it is read from the file. The overheads incurred in obtaining this extra security are negligible.

4. DIALOGUE DESIGN

There is as yet no established discipline for dialogue design, as is obvious from observing different interactive software in use. Steps have been taken to identify key areas in man–machine interaction,[9,10,11] but ad hoc construction still tends to be the norm. Inevitably the

application area exercises considerable influence over the design of a dialogue, but there are certain points which seem to be generally pertinent. A more detailed discussion is given by Robinson and Burns.[12]

Robustness

Above all other factors, the dialogue must be robust. Under no circumstances should an error in the host language's run-time system be invoked by user input. The dialogue module must maintain control at all times. While this is, perhaps, an obvious design criterion some of its implications might not be; for example, to avoid any error being caused by the entering of a number which is out of range, the PG must read all numbers as characters and reconstruct them character by character, testing the bounds at each stage. Another area which might give rise to problems is memory management. Static data structures should not cause any difficulty, but dynamic data structures can cause a run-time failure owing to lack of space in the free space area (heap). To retain control of memory allocation, the dialogue module must organise and maintain its own free space area, although this can lead to inefficiencies.

Scope of the Dialogue

In developing program generators for a range of applications, much thought has been given to the design and structure of the dialogue and the degree of freedom accorded to the user in terms of control of the dialogue. There are two extremes between which the user's level of control may be pitched: compete control and no control whatsoever. If the user has complete control then the dialogue is driven by commands from the user; at the other extreme, the user is merely a passive responder to prompts from the dialogue module. Our experience has led us to the conclusion that the latter extreme is closest to the 'optimal' solution. By identifying and using a natural problem definition sequence, a discipline is imposed upon the user by the dialogue – the 'correct' sequence must be followed. As a result it is much easier for the dialogue module to ensure that all entities requiring definition have been defined and that all requisite initialisations are performed.

It is always to be remembered that this software will be used by inexperienced people. Not only will they be inexperienced in computing but they will probably not be familiar with the logic of structuring a solution properly. It is worth repeating here that if the user does not understand the problem to be solved then something beyond a program generator is required. Assuming that the problem is understood, however, then a rigidly structured dialogue merely guides the user through a logical definition sequence designed by a domain specialist. For all applications it should be possible to identify such a sequence.

Frequently, the path taken through the dialogue depends upon the user's responses to earlier questions. In cases where a portion of the dialogue becomes irrelevant, the user should not be asked to provide any information, not even 'nil returns'!

Freedom of Expression

This is not always in the best interest of the user. The more latitude allowed in the dialogue, the more scope there is for errors in correctly analysing the user's intentions, both syntactically and semantically. The dialogue module will probably be very large without the inclusion of a general natural language analyser of the standards demanded by query languages[13,14] or artificial intelligence.[15,16]

Dialogue Layout

The dialogue should be designed to suit the device(s) on which it will be displayed. It is useful (and comforting) to have as much information on the screen as possible and, furthermore, a long sequence of very short prompts not only clears the screen quickly but is also irksome and unattractive. Owing to the predominance of video terminals with their low information capacity, it is desirable to identify distinct modules into which the dialogue may be partitioned. For each of these modules it should be possible for the user to obtain a summary of the relevant part of the specification. These summaries can be tailored to fit different screen formats.

Verbosity in the dialogue must be avoided at all costs. The information given to the user must be accurate, helpful but concise. There should be no attempt to make the dialogue chatty or amusing, since this becomes exceedingly tedious with use.

Experience Levels

It is difficult to design a dialogue which is sufficiently informative for the novice and not too demanding of the experienced user's patience. Consequently, more than one level of dialogue should be included.[17,20] For many generators, two dialogue modes are employed: a full version and an abbreviated one. Not only are the prompts shorter in the latter but the user can list items on one line, separating them by commas, and so curtail many of the prompts.

The dialogue can ascertain the current user's level of experience either explicitly, by asking, or implicitly, by assuming an experienced user and changing mode if the user indicates that further explanations are required. This should, however, be done with care, for a dynamically adapting interface may well disturb the conceptual model the user has of the system.[10]

Helpfulness

A dialogue which provides insufficient information for the user must be deemed to have failed. Even in full dialogue mode, some users might get into difficulties on certain points. Should this occur, then the user must be able to obtain further information by either typing 'help', which the dialogue manager must recognise or, preferably, by pressing a pre-assigned 'help' key on the keyboard.[9,18] A general-purpose dialogue control system which incorporates user levels, help facilities and backtracking has also been developed for the language Ada.[19] The next section illustrates these features in a program generator for simulation.

So, even though there may be one, two or more

```
EQUATIONS
        1? X'=1E-456*X

??? ERROR.
THE NUMBER IS TOO LARGE FOR THE CYBER.
X'=1E-456*X
          ∴
        1? X'=-123456789012345*X
WARNING! THE CYBER ONLY PROVIDES PRECISION TO
14 DIGITS.
WARNING! THE HP2100 ONLY PROVIDES PRECISION TO 10
DIGITS.
```

Figure 3. Numeric error-handling example.

```
        2? U'=K1*X-K2*Y
        3? U'=K3*Y-K4*U

??? ERROR
THIS VARIABLE HAS ALREADY BEEN DEFINED WITH
THIS CONDITION.
U'=K3*Y-K4*U
∴.∴
```

Figure 4. Structural error handling example.

```
        3? ∴
        2? Y'=K1*X-K2*Y
        3? U'=K3*Y-K4*U
        4? %
*** EXPERIMENT SPECIFICATION ***
```

Figure 5. Example of backtracking.

dialogue modes there is an extra dimension of information available which itself may be multi-layered. This can be handled quite easily by having a number of random access files, one for each level of information. Should more detail be required at the current position the corresponding record in the file for the next level of information can be accessed. In order to keep memory requirements to a minimum, the dialogue text and other information for users should be kept on files. This has the additional benefit of making the dialogue more amenable to change.

Backtracking

A useful feature in all program generators is the provision of a backtracking facility. By typing a special key in response to any prompt, the dialogue module returns the specification to the state it was in when the *previous* prompt was issued. This can be repeated, so that the user may step back through the dialogue and redefine the problem from any point. It is useful, while backtracking, for the dialogue module to remind the user of the previous input.

5. EXAMPLE DIALOGUE FROM A PROGRAM GENERATOR

The following examples have been included to illustrate some of the features of dialogues that have been discussed. They are taken from MODELLER, a program generator for the continuous system simulation of a model and the experiment to be performed on it. No knowledge is required of numerical algorithms for

```
RUN TYPE ? )
SELECT THE TYPE OF EXPERIMENT REQUIRED FROM:
  SR: SIMPLE RUN
  RF: RUN FUNCTIONS AND POST-RUN CALCULATIONS
  PS: PARAMETER SWEEP
  PO: PARAMETER OPTIMISATION
  BV: BOUNDARY VALUE
EXPERIMENT (SR, RF, PS, PO OR BV)? )
THE FIVE TYPES OF EXPERIMENT AVAILABLE ARE:
SR: A SINGLE, SIMPLE SIMULATION RUN OF THE
  MODEL.
RF: A SINGLE RUN OF THE SIMULATION FOR WHICH
  YOU CAN SPECIFY RUN FUNCTIONS TO BE DETECTED
  DURING THE RUN AND/OR ANY CALCULATIONS TO
  BE PERFORMED AFTER THE RUN. THE RUN
  FUNCTIONS AVAILABLE ARE PEAK ATPEAK AND
  CROSS. YOU CAN OBTAIN MORE INFORMATON ON
  THESE AT THE APPROPRIATE STAGE OF DEFINITION.
  POST-RUN CALCULATIONS USE FINAL VALUES OF
  VARIABLES AFTER THE SIMULATION RUN OR RUN
  FUNCTION VALUES.
PS: A PARAMETER SWEEP IN WHICH ONE OR TWO
  PARAMETERS THAT YOU SPECIFY ARE SWEPT
  THROUGH A RANGE OF VALUES, WITH A SIMULATION
  RUN FOR EVERY VALUE (AND CONBINATION OF
  VALUES). SUMMARY OUTPUT IS AUTOMATICALLY
  PRODUCED TO SHOW HOW SOME FUNCTION OF THE
  MODEL (SPECIFIED BY YOU) CHANGES AS THE
  PARAMETERS VARY.
PO: PARAMETER OPTIMISATION IN WHICH ONE OR
  TWO SPECIFIED PARAMETERS ARE VARIED
  AUTOMATICALLY IN ORDER TO MAXIMISE OR
  MINIMISE (YOU CHOOSE) THE COST FUNCTION YOU
  SPECIFY.
BV: BOUNDARY VALUE PROBLEMS VARY ONE OR TWO
  SPECIFIED PARAMETERS IN ORDER TO MATCH FINAL
  VALUES OF ONE OR TWO SPECIFIED VARIABLES AS
  ACCURATELY AS POSSIBLE.
NOTE. ALL DATA FOR THE MODEL AND EXPERIMENT
  ARE SPECIFIED LATER.

SELECT THE TYPE OF EXPERIMENT REQUIRED FROM:
  SR: SIMPLE RUN
  RF: RUN FUNCTIONS AND POST-RUN CALCULATIONS
  PS: PARAMETER SWEEP
  PO: PARAMETER OPTIMISATION
  BV: BOUNDARY VALUE
EXPERIMENT (SR, RF, PS, PO OR BV)?
```

Figure 6. Three information levels for the same prompt.

solving differential equations, or for controlling some of the experiments provided for, such as optimisation. On completion of a specification, a program may be generated in a continuous system simulation language, or FORTRAN. A full description of MODELLER has been published elsewhere.[21]

The dialogue must be robust, with all erroneous responses trapped by the dialogue manager. The first examples are taken from the part of the dialogue which controls equation specification. Extracts shown are contiguous. Figure 3 illustrates a warning of a loss of precision in both the host and target machines as well as a number too large to be represented on MODELLER's host (a CYBER). The next example in Figure 4 shows an error invoked by a variable having been defined on the left-hand side of two equations.

In Figure 5, the user backtracks to the previous equation in order to correct the mis-typed entry. Backtracking is allowed to any previous prompt. In MODELLER, all subsequent prompts are repeated, and consequently the responses have to be re-entered.

This may seem an unnecessary burden on the user, but it does not allow the opportunity for a different path through the dialogue being fed inappropriate responses. MODELLER does provide its own editor, which would normally be used to amend a specification, in preference to deep backtracking. Currently, MODELLER does not remind the user of the previous response at that point in the dialogue. As has been noted, such a feature would be useful.

MODELLER has three levels of prompt. The dialogue extracts so far have all been in 'expert mode' with a minimum of information. To obtain the next level of information, the user types '>' (analogously, '<' requests less information). The final example, Figure 6, shows a sequence of increasing information for the same prompt. The dialogue does not lock into its most explanatory mode, it reverts to the intermediate mode (in which all dialogues begin).

6. CONCLUSION

Program generation appears to be a genuinely useful technique for particular users and a number of types of application. Although program generators take time to construct, their ability to supply error-free tailored programs will continue to make them important software tools for the foreseeable future. Moreover, by adopting a standard approach, as outlined above, the initial difficulty in obtaining the PG design can be greatly reduced. An examination of Figure 2 reveals that many of the elements isolated within a program generator will, in fact, be similar or identical in all program generators. It is, therefore, possible to envisage a system which takes as input a 'Specification Requirement' and an 'Algorithm Description' and produces a complete program generator.

As with so many uses of information systems, the human–computer interface is an all-important one with PGs. The dialogue must be designed with the end user in mind, and should use one of the standard approaches now being developed. More than one mode of usage should be given, so that people with a range of expertise can use the same package. The opportunity to ask for help at any point of the specification must also be incorporated.

Further developments are needed in order to understand fully the essential features of the dialogue, although much experience is now available. Software support, in the form of natural language analysers and program code generators, is also needed. Present work involves the investigation of both of these areas.

REFERENCES

1. S. C. Matthewson and J. A. Allen, A commentary on the proposal for a simulation model specification and documentation language. *Proceedings UKSC Conference on Computer Simulation*. IPC Press (1978).
2. E. Subrahmanian and R. L. Cannon, A generator program for models of discrete-event systems. *Simulation*, pp. 93–101 (1981).
3. P. A. Luker and J. Stephenson, Simulation without programs. *Proceedings UKSC Conference on Computer Simulation*. IPC Press (1978).
4. P. A. Luker and J. Stephenson, Program generation for continuous system simulation. In *Simulation of Systems*, edited Dekker, Amsterdam, North-Holland (1979).
5. P. A. Luker and J. Stephenson, Towards a comprehensive modeller. *Proceedings Summer-Computer Simulation Conference, Seattle* (1980).
6. P. R. Mimmo, *Mathematically Provable Software: A Major New Technology*. HOS, Cambridge, Massachusetts (1982).
7. A. Burns and J. Robinson, *The specification of Interactive Ada Programs*. University of Bradford Report 43 (1984).
8. P. A. Luker, Computer assisted modelling of continuous systems. *Ph.D. Thesis*, University of Bradford (1982).
9. R. L. Wexelblat, *Design of Systems for Interaction between Humans and Computers*. BCS81, edited R. D. Parslow, pp. 259–272 (1981).
10. B. R. Gaines, The technology of interaction – dialogue programming rules. *International Journal of Man–Machine Studies* **14**, (1), 133–150 (1981).
11. J. Martin, *Design of Man–Computer Dialogues*, Prentice-Hall, Englewood Cliffs NJ (1973).
12. J. Robinson and A. Burns, A dialogue development system for the design and implementation of user interfaces in Ada. *Computer Journal* **28** (1), 22–28 (1985).
13. P. Subatier, *The New Generation Natural Language Systems*. BCS81, edited R. D. Parslow, pp. 253–258 (1981). Heyden & Sons Ltd., London.
14. J. Hendler, T. Kehler, P. Michaelis, B. Philips, K. Ross and R. Tennant, Issues in the development of natural language front-ends. National Computer Conference, Chicago (1981).
15. E. A. Felgenbaum, Themes and case studies of knowledge engineering. In *Expert Systems*, edited D. Michie, pp. 3–25 (1979).
16. E. Shortliffe, *Computer-based Medical Consultations*. MYCIN, Elsevier, New York (1976).
17. B. Negus, M. J. Hunt and J. A. Prentice, DIALOG! A scheme for quick and effective production of interactive applications software. *Software–Practice and Experience* **11**, 205–224 (1981).
18. A. Burns, Enhanced input/output in Pascal. *SIGPLAN Notices* (1983).
19. A. Burns and J. Robinson, A prototype dialogue development system. *Ada U.K. News* (1984).
20. J. Robinson and A. Burns, The use of multi-level adaptive user interfaces in improving user–computer interaction. Symposium on Empirical Foundations of information and Software Science, Atlanta. Plenum, New York (1984).
21. P. A. Luker, MODELLER – computer-assisted modelling of continuous systems. *Simulation* **42** (5) (1984).