The Rapid Generation of a Class of Software Tools

D. INCE AND M. WOODMAN
Faculty of Mathematics, Open University, Walton Hall, Milton Keynes MK7 6AA

A crucial aspect of every computer project is the notations used to describe its evolving software. Computer-based systems have become so large and so complex that software tools are needed to process these notations. Some requirements for such tools are presented and an environment for their rapid synthesis is described. The environment is based on the use of a table-driven LL(1) parser that processes the notation. The notation itself is expressed as a form of attribute grammar.

Received June 1984

1. INTRODUCTION – NOTATIONS AND THE SOFTWARE LIFE-CYCLE

There is now a consensus that producing large, complex pieces of software is an engineering process. One consequence of this is the model of software production known as the software life-cycle. It consists of a series of phases: requirements analysis, specification, design, implementation, operation and maintenance. Splitting the production of software into a series of phases with well-defined end-products leads to control of complexity, enhanced project visibility and the establishment of models for resource planning and allocation.

Each phase of the software life-cycle requires a notation to describe the characteristics of the software which are important for that phase. It may be a graphical notation, a mathematical notation, natural language or a constrained subset thereof, or a programming language.

Typical examples of such notations are: natural language for requirements analysis, graphic notations such as structure charts used in system design[1] and data flow diagrams used for functional specification,[2] special-purpose mathematical notations such as Z[3] for use in functional specification and program-like notations such as schematic logic used in system design.[4]

We have now reached the position in many software projects where the size and complexity of such notations is such that human processing is inadequate. The most recent expression of this was the report of the British Government's Alvey Committee,[5] and its subsequent software engineering strategy.[6] They identified a need for: '...more tools to assist with software specification, design, testing, rectification and development',[6] and envisaged the development of integrated programming support environments which contain: '...a compatible set of specification, design, programming, building and testing tools, supporting a development methodology that covers the entire life-cycle'.[6]

There have been very few automated tools for the processing of life-cycle notations, apart of course from tools such as compilers and interpreters used in implementation. Those that have existed, for example: PSL/PSA[7] for specification, SREM[8] for requirements analysis and specification and PDL[9] for detailed design, are themselves the result of a large software development effort. They suffer from the major disadvantage that the software developer who decides to use them has often to make a major investment decision in replacing the life-cycle notation that he currently uses. This paper describes the Toolbuild environment, which rapidly produces software tools for the processing of a wide variety of software notations which can be expressed using LL(1) grammars. The environment does not suffer from the disadvantage outlined above. A developer who wishes to synthesise software tools for his own notations can use Toolbuild in a relatively painless way without expending a large amount of resource.

2. THE REQUIREMENTS FOR SOFTWARE TOOLS FOR PROCESSING LIFE-CYCLE NOTATIONS

There are number of general requirements for software tools which are to process life-cycle notations. They are as follows.

Software tools are needed to check fragments of the processed notation for syntactic correctness. This, of course, assumes that a formal definition of the notation already exists.

Software tools are needed to display the structure of fragments of the processed notation. If the notation is a linear one such as a program design language then this implies some degree of formatting. If the notation is a graphical one then it implies some degree of pictorial layout.

Software tools are needed which are able to perform some degree of semantic processing. For example, a software tool for processing a program design language should check that all declared program units are used and that all used program units are declared.

Since software maintenance can occupy as much as 60% of project cost,[10] a software tool is required to provide adequate retrieval and query facilities for staff engaged in this activity.

A major requirement of any software tool is that it should be compatible with existing and future software tools. For example, the output from a specification tool should be able to be processed as an input by a design tool. One of the major developments of the 1980s will be the construction of program support environments.[11] These represent a merging of development methodologies and tools in order to support all the activities of the software life-cycle. It is mandatory that any tool used in such environments should be compatible with other tools used.

The software tool produced must be able to store facts which are in a form suitable for processing by advanced knowledge-based systems. A recent expression of this desire occurred with the British Government's Alvey initiative.[6] This envisaged the production of third-generation integrated program-support environments, whose central feature would be a set of compatible software tools which used notational data and project data stored in a central knowledge base.

Software tools should be easily transportable from computer to computer.

3. THE TOOLBUILD ENVIRONMENT

In response to the requirements outlined in the previous section, a prototype environment for the rapid production

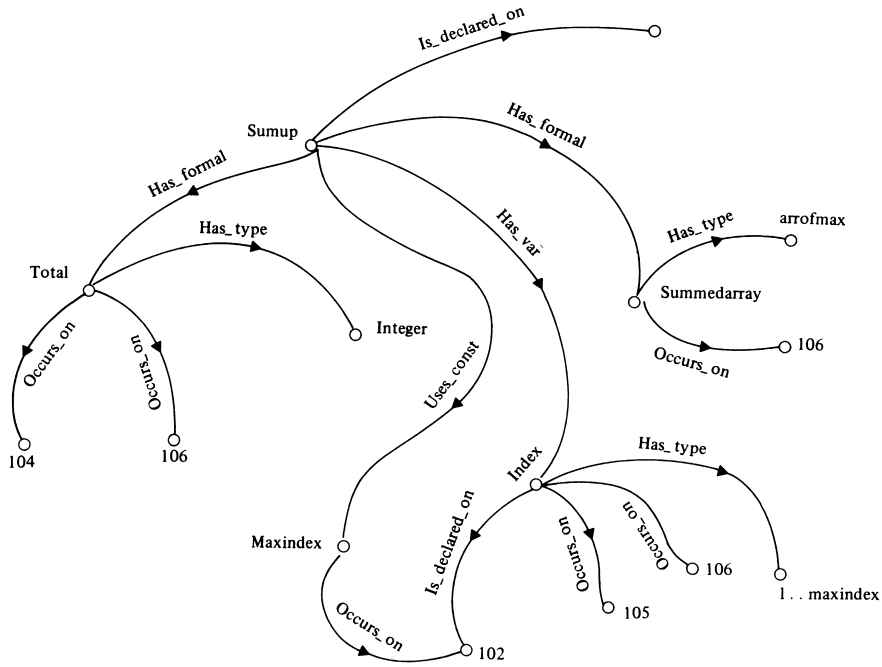

Figure 1. A possible semantic net for the Pascal procedure sumup.

of notational software tools has been constructed. It is programmed in Pascal and runs under the TOPS 20 operating system for the DEC 20 range of computers. The main idea behind the environment is that of the processing and storage of life-cycle notations as binary relations configured as semantic nets. In order to illustrate the facilities of the environment, the paper will use as example notations either the programming language Pascal or a program design language for which notational tools have already been produced using Toolbuild.

3.1 Semantic nets

A semantic net is a graph structure which has been used extensively in artificial intelligence applications. For example, semantic nets have been used in natural-language processing,[12] as a representation of facts in an expert system for mineral exploitation[13] and as a model of memory.[14]

A semantic net is a binary directed graph whose nodes represent objects and whose edges represent relations between the objects. A fragment of semantic net which describes the Pascal procedure *sumup* below is shown in Fig. 1.

```
101    procedure sumup(var total: integer; summedarray:
       arrofmax);
102      var index: 1 . . maxindex;
103        begin
104        total:=0;
105        for index:=1 to maxindex do total:
               =summedarray[index]+total
107        end;
```

The numbers on the left in the program fragment are assumed to be line numbers on a program listing. In Fig. 1 nodes in the semantic net are shown as small circles while relations are shown as labelled arcs. For example, Fig. 1 shows that the node *index* is a local variable in the procedure *sumup*, and that it is declared on line 102, and has a type *1 . . maxindex* and occurs on lines 105 and 106.

It has already been shown that semantic nets are a convenient mechanism for ensuring system compatibility,[15] for controlling source code versions of a software system[16] and for the maintenance of program and design notations.[17] Moreover, a software tool, based on semantic nets, has recently been constructed which processes a program design language used in detailed design and which provided facilities for maintenance personnel.[18] Semantic nets are useful within the context of notations for life-cycle activities for two reasons. First, they are a convenient unifying mechanism for expressing the binary relations that are inherent in a fragment of any life-cycle notation which can be expressed in an LL(1) form. The notation may range from a simple one such as a job control language to a more complex one such as a programming language or specification language. Secondly, semantic nets are also useful in that they are a direct representation of the relations inherent in a fragment of life-cycle notation. This means that the processing of these relations as a semantic net is much more simple than the processing of the fragment of notation from which the net has been derived. For example, a semantic net, stored as a hashed table, and which holds instances of relations that describe calling sequences in a programming language fragment, is easier to process than the source of that fragment. This, of course, assumes that a relatively painless way of converting from the source representation to the semantic net is available.

The main constraint in using semantic nets arises from the fact that they are an extremely simple general representation of the relations inherent in a fragment of software notation. This simplicity may lead to processing and storage inefficiencies. For example, a semantic net implemented as a hashed table which contains instances

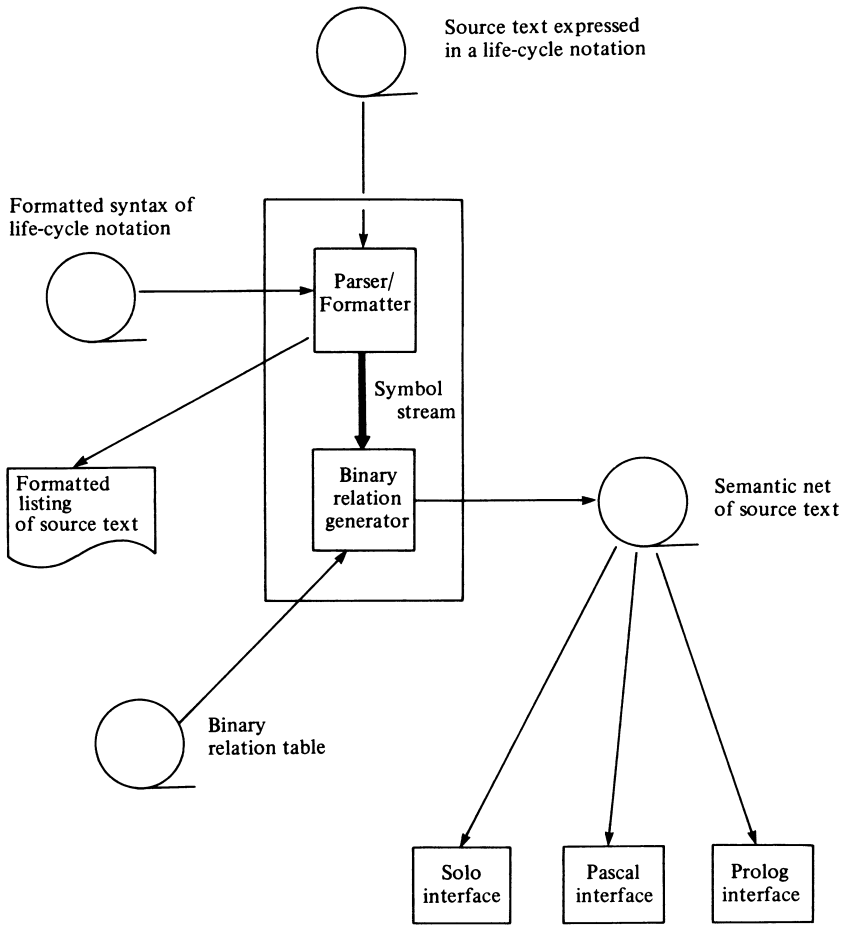

Figure 2. The architecture of the Toolbuild environment.

of a 1 to n relation such as *procedure calls* would be more inefficient than a circular list with backward, forward and parent pointers. However, little work has yet been performed on the information needs of the software developer and the structure of software engineering databases to provide evidence that a semantic net representation is too simple. A further constraint is that the notation processed, be it specification language or programming language, has either to be in an LL(1) form or can be translated into that form.

The work described in Ref. 18 was limited, in that a special-purpose tool was designed for one life-cycle notation. This paper reports a major extension and generalisation of that work. It has involved the production of an environment where tools can be produced relatively quickly for a number of notations which can be expressed using an LL(1) grammar.

3.2 The architecture of the Toolbuild environment

The Toolbuild environment is shown in Fig. 2. It consists of a number of components. Two are concerned with constructing a collection of binary relation instances which represent a semantic net that describes a particular fragment of life-cycle notation; the remaining three are interfaces to the semantic net.

The first component is a syntax-directed parser/formatter. This processes the source text of the notation and the syntax of the notation which has been overloaded with formatting instruction.[19] The syntax is expressed in a version of BNF known as EBNF.[20] The purpose of the parser/formatter is to check fragments of life-cycle notation for syntactic correctness and to display a formatted listing of the text which accords with the formatting instructions in the overloaded (formatted) syntax. The parser/formatter is based on a modified version of a software tool originally intended for use with programming languages.[19] The extract shown below describes one simple rule for the syntax of the program design language in the form processable by the parser/formatter. It shows the overloaded EBNF description of a design library.

librarydesign =
  '**library**' _ *libraryid* {*newline*} **R I** *librarydescription* **R R**
  '**end_library**' _ *libraryid* {*newline*}

The symbols R, I and _ are the formatting instructions to the syntax-directed translator. Expansion of the non-terminal *librarydesign* sets the left margin; R performs a return to the current margin which has been set; I indents with respect to the current margin and _ inserts a space into the listing. As well as producing a formatted listing the parser/formatter also produces a symbol stream. This symbol stream consists of a sequence of elements which represent the occurrence of syntactic entities in the output text. Each element consists of the syntactic entity name, its value and its location (page number, line number) on the listing produced by the parser/formatter. For example, the occurrence of the syntactic entity *identifier* which has the value *actuator* and

which occurs on line 12 of page 3 of the listing would be represented in the symbol stream as:

identifier 'actuator' 3 12

This symbol stream is then processed by a binary relation generator, which produces a semantic net that holds relations which describe the processed text of the life-cycle notation. The semantic net is stored very simply as sets of tables of triples.

The binary relation generator, in order to produce the semantic net, requires information about which relations need to be generated and in what contexts. This is provided by a binary relation table which relates syntactic entities occurring in the symbol stream to instances of relations in the generated semantic net. The binary relation table consists of a series of entries. Each entry holds the name of the syntactic entity, together with the processing that is to occur when an instance of that entity occurs in the symbol stream. An example of part of a binary relation table entry is shown below.

```
if invardecs then
  begin
  issue(symbolname,'is_declared_on',line,3)
  issue(symbolname,'is_a','variable',1);
  top(symbolstack,context);
  issue(symbolname,'is_var_declared_in',context,1);
  insert(symboltable,symbolname)
  end
```

It describes the part of the binary relation table entry for objects of syntactic class *ident* (identifier). It shows that when an object of syntactic class *ident* is encountered in a variable declaration an instance of the relation *is declared_on* is generated. The left-hand object in the relation instance will be the current symbol in the symbol stream and the right-hand object will be the current line number in the symbol stream. The fourth parameter (3) associates the generation of the relation instance to a level number. When a binary relation instance is generated from a fragment of life-cycle notation the user is able to influence this generation at run time. This is achieved by having ten levels of generation (1 . . 10). By switching levels on and off the user can influence which instances are generated. Thus, if level 3 is switched off an instance of the relation *is_declared_on* will not be generated.

The fourth line generates an instance of the relation *is a* which shows that the value of *ident* is a variable. After this the top value on a symbol stack is retrieved and is placed in the variable *context*. This variable holds either a procedure, function or program name. The fifth line issues an instance of the relation *is var_declared_in* which describes the fact that the processed identifier is declared in *context*. Finally, the symbol is inserted into a symbol table for later processing by other entries in the binary relation table.

The parser/formatter and binary relation generator are a form of compiler generator which accepts the syntax of a software notation expressed as an LL(1) attribute grammar.[21,22] However, the major difference between Toolbuild and typical compiler generators such as those described in Refs 23 and 24 is that program code, abstract code or semantic trees are not generated. What is generated is a semantic net which describes the relationships between syntactic objects in the processed notations.

The remaining components of the Toolbuild environment are concerned with the provision of interfaces between the generated semantic net and potential users. Interfaces exist for the programming languages: Solo, Pascal and Prolog. Solo is a programming language developed at the Open University.[25] It is an interactive procedural language for the manipulation of semantic nets. It provides the range of facilities that would normally be expected in a procedural language. Facilities exist for: input/output; repetitive control; conditional control; the definition and invocation of procedures; the retrieval, deletion and modification of semantic nets; and pattern matching on semantic nets. The main use of Solo is in forming queries on the semantic net. It is envisaged that it will be most useful for staff engaged in software maintenance. An example of its use is shown below. It shows a Solo procedure PRCALLVAR which queries a semantic net holding objects and relations for the text of a software system expressed in the program design language. This procedure has two parameters /X/ and /Y/; it displays those procedures which are called by procedure /X/ and in which variable /Y/ is declared. A relation is described in Solo as:

left object — relation —> right object

This notation will be used throughout this paper.

```
TO PRCALLVAR /X/ /Y/
FOR EACH CASE OF /X/ — CALLS → ?A
CHECK /Y/ — IS_VAR_DECLARED_IN → *A
  IF PRESENT: PRINT *A; CONTINUE
  IF ABSENT: CONTINUE
DONE
```

PRCALLVAR iterates over each instance of the relation *calls* in which /X/ participates. Each time that an instance is found the right-hand side is placed in the variable A. For each value of A so found Solo then checks that it occurs in the right-hand side of a relation instance of *is var declared in* with the left-hand side being the parameter /Y/. If an instance is found the procedure is displayed. The asterisk in front of the variable A in the Solo fragment represents the value of A which has been pattern matched in the relation

/X/ — CALLS —> ?A

This is a simple example in the use of Solo. In general any query which can be expressed in first-order predicate calculus can be expressed procedurally in Solo. Although the Solo notation looks a little cumbersome it has achieved a great deal of success in teaching programming and artificial intelligence concepts to a wide variety of novice students.[25] The syntax of Solo is described in an appendix to this paper.

The Pascal interface consists of a set of procedures which perform pattern matching and retrieval of objects and relations held in the semantic net. An example of its use is shown below. It shows the Pascal procedure *checkdecused*, which processes the semantic net that represents a fragment of the program design language and prints out the name of those procedures which are declared but never called.

```
procedure checkdecused(programme:string);
var
  procs, lines:answers
  procname, linenumber:string;
```

```
begin
getobjects(programname,'has_proc',procs,maindatabase);
nextfrom(proc,procname);
while procname <>endofanswers do
  begin
  getobjects(procname,'is_called_on',lines,maindatabase);
  nextfrom(lines,linenumber);
  if linenumber=endofanswers then
    begin
    write('procedure');
    writestring(procname);
    writeln('is declared but not called')
    end;
  nextfrom(procs,procname)
  end
end;
```

Checkdecused uses the type *answers*, the constant *endofanswers* and the procedures *getobjects* and *nextfrom*. These are all part of the Pascal interface. *Answers* is a sequence of strings which consist of all the objects that participate in a relation on a right-hand side, given a left-hand side and the name of a relation. *Nextfrom* extracts from an object of type *answers* each string that occurs in it. *Getobjects* has four parameters: the first is a string on the left-hand side of a relation, the second is a string which is a relation name, the third is of type *answers* and the fourth is a data structure holding the semantic net made up of binary relations. When called *getobjects* places in the third parameter those strings which occur in instances of relations whose left-hand side is the first parameter, whose relation name is the second parameter and which occur in the data structure given as the fourth parameter. The constant *endofanswers* delimits the end of the strings that make up the sequence held in an object of type *answers*.

Checkdecused achieves its effect by extracting the name of all the procedures in the program *programname* and placing the sequence of procedure names in *procs*. The relation *has proc* which defines the division of the program into procedures is used as the second parameter in the first call of *getobjects*.

Each procedure is then extracted using *nextfrom* and a second call on *getobjects* places a sequence of line numbers on which the procedure is called in *lines*. The first element of this sequence is then extracted. If it is the constant *endofanswers* then there were no calls. The procedure has been declared in *progname* but not called, and its name is printed out.

It is envisaged that the Pascal interface would be employed in activities such as report writing or static analysis by users who wish to perform the type of sophisticated processing that would be difficult or awkward in Solo. A full description of the interface is an appendix to this paper.

The PROLOG interface consists of a Pascal program which converts a semantic net into a series of PROLOG clauses that represent the semantic net and which can be easily inserted into PROLOG programs. It is envisaged that users who wish to perform knowledge processing or wish to query the semantic net in a non-procedural way would use this interface.

4. APPLICATIONS OF THE TOOLBUILD ENVIRONMENT

There are six major applications of the Toolbuild environment. The first is as a basis for the rapid generation of automated tools which perform syntax checking and formatting of a wide variety of life-cycle notations.

The second application is as a basis for the rapid generation of automated software maintenance tools. Van Horn[26] has accurately described the lack of software tools for this phase of the software life-cycle:

> Information retrieval technology has not yet been rigorously applied to the problem of obtaining information about the structure, code and documentation of an existing software system. Yet the quality of software evolution depends critically on how well the evolver understands the system that he is evolving. We are not doing everything we could to facilitate that understanding

We hope that the Toolbuild environment is a first step towards the provision of portable maintenance tools which are able to process complex queries about a software system expressed in a variety of life-cycle notations.

A third application is as a basis for the rapid generation of tools for the automatic enforcement of standards. A description of a software system in a specific life-cycle notation should be readable, comprehensible and not over-complex. Normally, in a development environment the software developer will insist on notational standards which encourage these properties. Typical examples of such standards are:

> Vertically align **repeat**...**until** clauses.
>
> In a subroutine or procedure declaration the input parameters should be written first followed by the output parameters and then any input/output parameters.
>
> Subroutines or procedures should not contain control constructs which are nested to a depth greater than four.
>
> The ratio of commented code to executable code should be between 20 and 25%.
>
> In a design the fan-out from a program unit should be no higher than six.
>
> In a design the names of modules should be unique and follow a specific convention. For example, each name should consist of the system name followed by a string.

Although notational standards are still a controversial topic, the point made in this paper is that whatever standards are chosen there is a very high probability that the Toolbuild environment will be able to enforce them. This enforcement can be done in two ways. First, the formatted syntax processed by the parser/formatter can enforce layout- and syntax-related standards. The first two standards shown above would come into this category.

A second category of standards can be enforced by the Pascal or Prolog interfaces. By careful choice of relations produced by the binary relation generator, standards such as the last three above can be enforced. For example, the standard which relates the ratio of commented lines to executable lines can be enforced by generating relations of the form:

SYSTEM — occurs_line → line_number
END_SYSTEM — occurs_line → line_number
comment — starts _on_line → line_number
comment — ends_on_line → line_number

and by using either the Pascal interface or Prolog interface to calculate the ratio present.

At present there are very few reliable experiments which indicate which particular standard is beneficial. A software manager who devises standards often does so iteratively using common sense and experience. The Toolbuild environment easily allows this form of development. Whenever a standard is introduced, removed or modified in a life-cycle notation either the syntax of the notation is modified or the binary relation generation rules changed and new Pascal or Prolog code written to enforce the standard. This can involve much less work than modifying a special-purpose one-off standards-enforcement tool.

A fourth application of the Toolbuild environment is as a basis for the rapid generation of static analysis tools for the detection of errors and anomalies. They detect anomalies such as uninitialised variables, incorrect common block alignment and incorrect parameter alignment. Such tools have in the past been mainly used in FORTRAN environments[27, 28] but there is no reason why they cannot be constructed for other life-cycle notations. Such static analysers can take man-years to construct, or rely on non-portable software such as general-purpose database management systems.

A fifth application is in the rapid generation of tools for metric calculation. If software engineering is to become a mature engineering discipline its practitioners are going to have to be able to judge the quality of a piece of software as it progresses through its life-cycle. One way of doing this is by examining the text of software specification and design notations and evaluating structural metrics.

There is a serious shortage of research work concerned with deriving quality metrics from design and specification notations. One view is that with design in particular, we now know enough about what constitutes a good design in structural terms to be able to establish metrics; that the only bar to their derivation is a lack of automated tools for the processing of design notations.[29]

Toolbuild is capable of producing software tools which are able to extract structural properties such as module fan-in and fan-out and complexity of shared data. From these properties design metrics can be calculated. Moreover, the ease with which such tools can be produced is compatible with the way in which structural metrics research will be initially carried out. Toolbuild will enable the metrics researcher to modify, delete and include new relations and objects in the generated semantic net and relate behavioural properties of the software such as reported bugs to these relations and objects. The researcher would gradually refine the structural properties monitored until an adequate correlation was achieved.

A final, more tentative application of the Toolbuild environment is in the domain of knowledge processing. A semantic net which has been derived from a fragment of life-cycle notation represents facts about the software that the notation describes. For these facts inferences can be made. Semantic nets have been used in a number of expert systems, see for example Ref. 13, which have been successful in a number of limited domains. Little research has yet been performed in the use of knowledge-based systems in software projects. Indeed, it is still an open question as to whether the same degree of success can be achieved for semantic net-based systems in a slightly richer domain as software development. However, the Toolbuild environment will at least provide an experimental testbed for resolving some of the number of unanswered research questions that remain.

5. USING THE TOOLBUILD ENVIRONMENT

In order to develop a set of tools for a specific life-cycle notation the software developer has to carry out a number of activities. First, the syntax of the notation has to be defined in EBNF. Secondly, this syntax has to be transformed to a formatted syntax[19] which determines the layout of fragments of the notation. Whatever other tools the developer requires, he has already carried out the actions needed to produce a processor for syntax checking and formatting.

The next step is to define which binary relations are to be generated from fragments of the notation, and construct the binary relation table. The choice of relation depends on the tools that are required over and above the already constructed syntax checker and formatter. For example, if the developer wishes to use Solo, Pascal or Prolog to extract maintenance information he must choose relations which reflect the structural properties which maintenance personnel wish to query. For example, relations which indicate calling sequences, hierarchy structure and variable usage would be defined. If the developer wishes to construct a standards enforcer then he would choose relations which reflected the structural, syntactic and layout properties that are to be enforced.

If the developer only requires formatting, syntax checking and query processing for maintenance personnel no further work need be done. Maintenance staff can use the Pascal, Solo or Prolog interfaces to query the generated semantic net.

However, if the developer wishes to develop further tools such as static analysers or standards enforcers then programs will need to be written which use one of the Toolbuild interfaces to examine the generated semantic net. The processed data for these tools is in such a simple form, i.e. a set of binary relations, that the process of writing these programs, while not trivial, is not a particularly difficult or time-consuming task.

6 DISCUSSION AND CONCLUSIONS

In gauging the effectiveness of the Toolbuild environment it is first necessary to re-examine the six requirements presented in the second section of this paper. First, the Toolbuild environment is able to check for syntactic correctness. The parser/formatter processes fragments of life-cycle notation which are defined in EBNF. However, the syntax must be expressed as an LL(1) grammar.

Secondly, the Toolbuild environment is capable of formatting fragments of life-cycle notations. Moreover, this formatting is under the control of the project manager. By including formatting instructions which overload the syntax of the life-cycle notation preferred layouts can be imposed. A weakness of the Toolbuild environment is that it is at present only capable of processing linear notations. It cannot syntax-check or format graphic notations. However, this is not a

particularly serious drawback. Toolbuild is capable of being used as a sophisticated back-end processor to simple graphics software which constructs and displays graphic notations. Indeed, in an earlier project we have used semantic nets to represent data flow diagrams used in system specification. By processing a linear 'compiled' version of the graphics produced by the graphics software, Toolbuild is capable of responding to many of the requirements outlined in this paper.

Thirdly, the Toolbuild environment is also capable of satisfying the requirements for semantic processing. By constructing programs using the Pascal or Prolog interfaces a large amount of semantic processing can be performed. This ranges from simple semantic processing such as checking that a called program unit has been declared, up to the generation of sophisticated structural metric values.

Fourthly, the Toolbuild environment is capable of satisfying the maintenance requirements. Indeed, it represents one of its strengths. Maintenance staff are able to interrogate text of a life-cycle notation; procedurally using Pascal or Solo, or non-procedurally using Prolog. Moreover, the use of Solo allows queries to be constructed and processed interactively in the same way that a sophisticated DBMS query language would be used. The derivation of maintenance information should not be confined to the use of the three interfaces which have been constructed. The simple method of storage chosen for the semantic net should mean that provision of interfaces to languages such as LISP and to relational database management systems should not be difficult. The experience with the Prolog interface supports this, as it only took five man-days to construct.

The requirement for compatibility with other software tools is satisfied in the respect that a developer using the Toolbuild environment for a series of consecutive life-cycle notations would use the same facilities and storage method for the semantic nets which represent fragments of these notations.

The requirement for storage of notations in a form suitable for processing by knowledge-based systems is satisfied to some degree by the Toolbuild environment. Semantic nets have been used with some success in a number of expert systems. However, it is still not clear whether more sophisticated systems based on richer data structures such as the frame[30] would be better suited for knowledge processing in a software project. One strand of our current research is concerned with investigating how Toolbuild can be parameterised to generate a variety of data structures.

The Toolbuild environment satisfies the requirement for portability. All the processors, together with the interpreter for Solo, are written in standard Pascal.

We are confident that the Toolbuild environment satisfies the requirements outlined in this paper. The final question to be addressed is how quickly software tools can be produced using the environment. Experience so far has been good. A syntax checker and formatter for a program design language[31] was constructed in two man-weeks. A syntax checker and formatter for Modular-2 was produced in four man-weeks. A binary relation generator which produced semantic nets for the program design language was produced in three man-days. This work was carried out by the authors of the paper, who are fully conversant with the system. However, we feel that developers who are unfamiliar with the environment should not take considerably more time to develop tools using Toolbuild.

Future developments for the system which are feasible have already been mentioned: the use of the system as a back-end for graphics processors, the generalisation of the system so that it can generate arbitrary data structures, its use in knowledge-based processing. One particularly important extension is the use of the system in constructing interactive programming environments. It has already been shown[32] that formatted syntaxes can be used to build such environments; one future enhancement to the Toolbuild environment would be to include facilities to construct interactive preparation systems for a variety of software engineering notations.

However, the most immediate task is to speed the system up. The majority of the processors are of production quality speed, apart from the binary relation generator. If the system is to be seriously used on major software projects this is where work will have to be carried out over the next year.

REFERENCES

1. W. P. Stevens, G. J. Myers and L. L. Constantine, Structured design. *IBM Systems Journal* **13**, (2) 115–139 (1974).
2. T. De Marco, *Structured Analysis and System Specification.* Englewood Cliffs, NJ: Prentice Hall (1979).
3. J. R. Abrial, *The Specification Language Z* : Basic Library, Oxford University Programming Research Group Interim Report (1980).
4. M. A. Jackson, *Principles of Program Design.* London: Academic Press (1975).
5. *A Programme for Advanced Technology*: *The report of the Alvey Committee.* HMSO (1983).
6. *Alvey Programme*: *Software Engineering Strategy.* Department of Trade and Industry (1983).
7. D. Teichrow and E. Hershey, PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems. *IEEE Trans. on Software Engineering* **3**, (1) 41–48 (1977).
8. C. Davis and C. Vick, The Software Development System. *IEEE Trans. on Software Engineering* **3** (1) 67–84 (1977).
9. S. Caine and K. Gordon, PDL – a tool for software design. *Proceedings National Computer Conference*, AFIPS, 271–276 (1975).
10. B. P. Lientz and E. B. Swanson, *Software Maintenance Management.* Reading, MA: Addison-Wesley (1980).
11. A. I. Wasserman, Towards integrated software development environments. *Scientia* **115**, 663–684 (1980).
12. D. E. Walker (ed.), *Understanding Spoken Language.* New York: North-Holland (1978).
13. R. O. Duda, P. E. Hart, K. Konolige and R. Reboh, *A Computer-based Consultant for Mineral Exploitation.* Technical Report, SRI International (1979).
14. M. R. Quillian, Semantic memory. In *Semantic Information Processing.* Cambridge, MA: MIT 216–270 (1968).
15. D. C. Ince, A compatibility software tool for use with separately compiled languages. *SIGPLAN Notices* **18** (9), 31–34 (1983).
16. D. C. Ince, A source code control system based on semantic nets. *Software – Practice and Experience* **14** (12), 583–594 (1984).

17. D. C. Ince., The provision of procedural and functional interfaces for the maintenance of program design and program language notations. *SIGPLAN Notices* **19** (2), (1984).
18. D. C. Ince, A program design language maintenance tool based on semantic nets. *IEEE Computer Society Workshop on Software Maintenance, Monterey*, pp. 160–164 (1983).
19. G. A. Rose and J. Welsh, Formatted programming languages. *Software – Practice and Experience* **11**, 651–669 (1981).
20. N. Wirth, What can we do about the unnecessary diversity of notations for syntactic definitions? *Comm. ACM* **12**, (11) (1977).
21. D. R. Milton and C. N. Fischer, LL(k) parsing for attributed grammars. *Proc. 6th International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science no. 71, pp. 422–430. Springer Verlag, Heidelberg (1979).
22. P. M. Lewis, B. Hutt and E. Zimmerman, Attributed translations. *Journal of Computer and System Sciences* **9** (3), 279–307 (1974).
23. D. R. Milton, L. W. Kirchhoff and B. R. Rowland, An LL(1) compiler generator (SIGPLAN Symposium on Compiler Construction). *SIGPLAN Notices* **14** (8), 52–157 (1979).
24. U. Kastens, B. Hutt and E. Zimmerman, *GAG – A Practical Compiler Generator*. Lecture Notes in Computer Science no. 141. Springer Verlag, Heidelberg (1982).
25. M. Eisenstadt, A user friendly software environment for the novice programmer. *Comm. ACM* **26** (12), 1058–1064 (1983).
26. E. C. Van Horn, Software must evolve. In *Software Engineering*, pp. 209–226. Academic Press, London (1980).
27. J. C. Browne and D. B. Johnson, FAST: a second generation program analysis system. *Proc. 3rd International Conference on Software Engineering* (1978).
28. L. D. Fosdick and L. J. Osterweil, *Some Experiences with Dave – A Fortran PROGRAM analyser*. NewYork, AFIPS, pp. 909–915 (1976).
29. D. C. Ince, The evaluation of software design. *Design Studies* (to be published).
30. M. Minsky, A framework for representing knowledge. In *The Psychology of Computer Vision*. McGraw-Hill, New York (1975).
31. M. Woodman, A program design language for software engineering. *SIGPLAN Notices* (to appear).
32. G. Rose and T. Roper, Generation of program preparation systems for formatted languages. *Proc. IFIP '83, Paris* (Sept. 1983).
33. British Standards Institute. *Method of Defining Syntactic Metalanguage*. BS 6154 (1981).

APPENDIX

This appendix describes the syntax of both Solo and the Pascal interface using the British Standards Syntactic Metalanguage[33] augmented with EBNF.[20] The notation, briefly, is as follows:

, used as an explicit concatenation symbol
; used as a production rule terminator
(* *) used to include comments – used here for semantics
? ? used to enclose special sequences
' used to delimit terminal symbols
* used to indicate repetition
– used for syntactic exception

Solo is a novice programming language suitable for artificial intelligence applications. A Solo database consists of semantic nets constructed by a user together with procedures for the manipulation of these nets. The procedures will also have been written by a user. A more informal description of Solo is contained in Ref. 25. The syntax of the language is shown below:

```
solo session = {solo command} finish command ;
finish command = (quit | bye) , command terminator ;
command terminator = ' ; ' | ? UK 7-bit character carriage return ? ;
solo command = (top level command | edit command) , command terminator ;
top level command = common command | top only command ;
edit command = common command | edit only command ;
top only command = edit | help | simple check | simple test | to ;
edit only command = complex check | complex test | for | while | until ;
common command = comment | describe | disable | dir | dump | enable | forget | input | kill | let | list | note | plot | print | save | write | user proc call;
bye = 'BYE'
    (* Is used to save the current Solo database on the user disc and terminate the current SOLO session *);
quit = 'QUIT'
    (* terminates the current Solo session and leaves the database as it was when the session was started *);
simple check = 'CHECK' , ' ' , pattern ;
complex check = simple check , command terminator , if present , command terminator , if absent ;
if present = 'If Present:' , edit command – conds , ';' , control command ;
if absent = 'If absent:' , edit command – conds , ';' , control command ;
pattern = node , link , node | node , link , wildcard | wildcard , link , node
    (* if pattern starts with a wildcard it can't end with one and vice versa *);
link = '—' , relation , '→' ;
simple test = 'TEST' , ' ' , node , ' ' , '=' , ' ' , node
    (* inspects the nodes to see if they are the same *);
complex test = simple test , command terminator , if yes, command terminator , if no ;
if yes = 'If Yes:' , edit command – conds , ';' , control command ;
if no = 'If No:' , edit command – conds , ';' , control command ;
    (*If the TEST is within a user procedure, then it is followed by two sub-lines which indicate what action Solo is to take depending on the success/failure of the TEST, i.e.,
      If Yes: (procedure); control statement
      If No: (procedure); control statement *)
control command = continue | exit | next case ;
exit = 'EXIT'
    (* used by either CHECK or TEST to terminate execution of the current user procedure *);
continue = 'CONTINUE'
```

```
    (* used by either Check or TEST to tell Solo to
    continue execution of the current user procedure at
    the next statement *);
next case = 'NEXTCASE'
    (* is a control statement used by either CHECK or
    TEST to tell Solo to return to the higher level FOR
    procedure in which CHECK or TEST is included*);
conds = check | test | for | while | until;
describe = 'DESCRIBE' , ' ' , node
    (* gives the description of a node in a semantic net*);
dir = 'DIR'
    (* gives the full list within the current Solo database
    of
     (i) The set of Solo procedures.
     (ii) The set of user defined procedures.
    (iii) The set of nodes for which descriptions exist *);
disable = 'DISABLE' , ' ' , switch
    (* will turn OFF one of the switches which define the
    user's working environment *) ;
enable = 'ENABLE' , ' ' , switch
    (* will turn ON one of the switches which define the
    user's working environment*) ;
switch = inverse | protection | printout | printer |
    outfile | trace ;
trace = 'TRACESTATS' (* traces all procedures *) |
        'TRACEPROCS' (* traces user procedures *) |
        'TRACEPLOT' (* traces output to plotting
        subsystems*) ;
inverse = 'INVERSE' (* allows pattern matching on first
          node *) ;
protection = 'PROTECTION' (* prevents novice access
             to procedures *) ;
printer = 'PRINTER' (* to all terminal output goes to
          printer *) ;
printout = 'PRINTOUT' (* all uses of NOTE and
           FORGET result in verification *) ;
outfile = 'OUTFILE' (* all terminal output is also
          copied to file *) ;
dump = 'DUMP'
    (* will list the complete set of user procedures and
    the entire semantic net to the terminal and the
    printer or output file if the appropriate switches are
    enabled *) ;
edit = 'EDIT' , ' ' , user procedure
    (* activates the screen editor *) ;
edit session = { up | down | insert | replace } , done;
up = ? function key or control code for cursor up ?
    (* moves you to previous line *);
down = ? function key or control code for cursor down?
    (*moves user to next line if not on last line *) ;
insert = ? function key or control code for insert ? , edit
    command
    (*inserts a new line before current line *)
replace = ? function key or control code for replace ? ,
    (edit command | if present | if absent | if yes |
    if no | do)
    (*replaces current line and any sub-lines *) ;
done = ? function key or control code for done ?
    (* finishes edit session and returns to Solo top level*);
forget = 'FORGET' , ' ' , pattern
    (* will remove the specified relation(s) from the
    semantic net*);
help = 'HELP' , ' ' procedure name
    (* Help may be obtained on any Solo procedure by
    typing:
    HELP >SOLO procedure> *) ;
input = 'INPUT' , ' ' , variable {variable}
    (* Will halt the activated procedure and ask the user
    to supply a value for the variable *);
kill = 'KILL' , ' ' , user procedure
    (* Will remove the user procedure from the SOLO
    database *) ;
let = 'LET' , ' ' , variable = node
    (* will assign a value to a variable *) ;
list = 'LIST' , ' ' , user procedure
    (* will display the specified user procedure *) ;
note = 'NOTE' , ' ' , node , link , node
    (* stores a new relation in a semantic net *);
print = 'PRINT' , print list
    (* evaluates print list and prints on new line *) ;
write = 'WRITE' , print list
    (* evaluates print list and prints on current line *);
plot = 'PLOT' , print list
    (* evaluates print list and sends to graphics
    sub-system *) ;
print list = { ' ' , (quoted string | simple node)} ;
quoted string = '"' , { character } , '"' ;
parameter = '/' , id character , '/' ;
node = simple node , {' 'S' , ' ' , node};
simple node = identifier | parameter | variable ;
variable = '*' , [id character] ;
user procedure = identifier ;
identifier = id character , 11 * [id character]
    (*names may be up to 12 characters, etc. *)
    (*Assume definitions of capitals, printable characters,
    etc.*)
character = printable character ;
id character = capital | '.' | '_' | '(' | ')' ;
save = 'SAVE'
    (* Will make a copy of the current SOLO database
    on the user disc *);
for = 'FOR EACH CASE OF' , ' ' , pattern , command
    terminator , do ;
do = 'Do:' , ' ' , edit command ;
to = 'TO' , ' ' , user procedure , [ ' ' , parameter];
while = 'WHILE' , ' ' , pattern , command terminator,
    do ;
until = 'UNTIL' , ' ' , pattern , command terminator ,
    do ;
user proc call = user procedure , {' ' , node} ;
```

The interface to Pascal is similarly described. It consists of a series of procedures for pattern matching and the sequential retrieval of relations which match a particular pattern. The syntax shown below assumes the standard definition of Pascal declarations and Pascal statements. The procedures assume that a relation is given as the triple

subject verb object

```
pascalinterface = 'procedure' 'userprocedure' user-
procbody;
userprocbody =
    declarations 'begin' {pascalstatements | interface-
calls} 'end' ';' ;
declarations = {pascaldeclarations | interfacetyperefs} ;
interfacetyperefs =
    'dictionary'(*Data structure holding semantic net*) |
    'string'(* Large packed array of characters *)
    'answers'(* List of results obtained by pattern
                matching procedure*);
interfacecalls =
```

```
'initialisedictionary' '(' dictionaryid ')'
(* Sets up net data structure *) | 'buildnet'
'(' binrelationfile ', 'dictionaryid ')'
(* reads file of triples into the semantic net *) |
'inserttriple' '(' subjectstring ',' verbstring ',
    ' objectstring ','dictionaryid ')' (* Inserts a single
    triple *) |
'readstring' '(' stringid ')' (* Read string from
terminal *) |
'writestring' '(' stringid ')' (* Write string to
terminal *) | 'nexttfrom' '(' listofresultsid ',
' resultstringid ')'
(* returns the next result from the list of strings *)
 | 'getsubjects''(' subjecttobematched','givenverb',
    ' givenobject , ',' dictionaryid ')'
(* constructs list of answers satisfying relations with
verb and object *) |
'getverbs' '('givensubject ', ' verbtobematched ',
' givenobject ', ' dictionaryid ')'
(* constructs lists of answers satisfying relations with
subject and object *) |
'getobjects' '(' givensubject ', ' givenverb ',
' objecttobematched ', ' dictionaryid ')'
(* constructs list of answers satisfying relations with
subject and verb *);
```