Abstract. This paper presents the DELPHI Project which deals with the design and implementation of a software engineering environment (SEE) to enhance software productivity, in particular, one that targets information systems (IS) for business data processing problem domain. A set of specification languages were designed to specify the IS in a modular manner. Collectively, these languages are called Data Processing Specification Outlining Language (DPSOL). DPSOL is customizable, executable, extensible and translatable. DPSOL can be used in rapid prototyping or in automatic translation to a target 3GL/environment. The language can be extended thereby enriching its capability for specification. This paper presents the expressive power of DPSOL to specify the IS and its automated process of checking referential integrity and computational consistency. Our approach is translational using templates instead of transformational. It was found that templating is a powerful concept in automatic programming, at least for IS. The templates can be edited and modified to generate customized 3GL source code. Moreover, documentation can also be automatically generated.

Index Terms. executable specification language, rapid prototyping, automatic programming, SEE, CASE tools.

Introduction

Information system in business is the largest single software application area that currently exists. Example of applications are payroll, accounts receivable/payable, invoice and inventory, all of which access one or more large databases. These information systems help the operation of the business and management decision making. The problem faced in developing business information systems is that, users who participate in the development phase are normally middle managers who know what they want but unfortunately cannot state clearly and completely what they want - although they claim to recognize the desired system if they see one.

This calls for a rapid prototyping system which allows the user to simulate the operation of the information system through a prototype. The prototype can then be revised and exercised until the user is able to "see" the needed information system.

The next step in developing information systems is to implement the approved prototype into the delivered system. This step suffers from deficiencies in preserving the correct functionality of the specification. Furthermore, the translation is painfully slow because of the large manual process of design and coding where most programs and/or modules are done from scratch. If the prototype goes to be the delivered system (as what some 4GLs do), it suffers from lack of portability and from handling large volume of transactions. Hence, it is still advantageous to translate the prototype in 3GLs [4].

Finally, maintenance of the information system becomes a problem when changes occur in the system requirements, access databases, another 3GL, operating systems or machines. The usual practice is to "patch-up" the system to accommodate the changes which usually results in unstructured information system that is unreliable and sometimes inconsistent.

New software paradigms are needed to solve the above mentioned problems. In Agresti [1], three classification of software paradigms were presented: the prototyping paradigm, the operational paradigm, and the transformational paradigm. To address the problems encountered in information systems, we propose a software paradigm that encompasses the properties of the three paradigms. We call this the prototyping/translational paradigm. We propose translational instead of transformational approach. The difference is discussed in later sections. This paradigm includes the rapid prototyping capability and a translation process that automatically translates the specification into the target 3GL/environment. Although rapid prototyping is not a new concept, the translation process is still very limited. The automatic generation of the source code from a given form solves the slow manual process of design and coding, and the maintenance problem to accommodate changes. Although several transformational systems already exist, see [5], none of them addresses information systems for business data processing problem domain.

The DELPHI project was undertaken at Software Brewers Inc., whose goal is to develop a software engineering environment (SGSA). As an interface to users we have developed an executable requirement specification language called Data Processing Specification Outlining Language (DPSOL). DPSOL helps a user develop application programs with little or no help from programmers. This paper presents the philosophy and rationale of the design of DPSOL, its ability to specify the desired information system, its executability by being able to prototype and its translatable by being able to automatically translate the specification into a target 3GL source/environment and/or documentation.

This paper is organized as follows. Section 2 presents the DELPHI's software development paradigm. Section 3 describes
the requirements specification language that supports the paradigm, the model of describing information system, syntax and semantics. An example requirements specification language is also given for an inventory system. Section 4 discusses the process of translating the specification into a 3GL source code/environment and/or documentation. Also discussed is the automatic translation of the specification using the concept of templates. Section 5 presents our conclusion drawn from the DELPHI Project and proposes some future directions in enhancing and extending the project.

The Prototyping/Translational Paradigm

Shown in Fig. 1 is the prototyping/translational paradigm adopted in the DELPHI Project. In this paradigm the system requirements are given in an informal or sometimes incomplete format. These are then describe in a more formal manner using a specification language. The specification language is express in a relatively high level language in terms of objects, processes and relationships.

There are two alternative paths in the paradigm, either the program specification is translated automatically into the target 3GL/environment or into a prototype.

Fig. 1. Prototyping/Translational Paradigm

The translation of the program specification into the target 3GL source code in the desired environment (host operating system or host machine) involves automatic programming techniques. This part eliminates the design and coding phases of the old waterfall model [1]. However, it is necessary that the translation process preserves the functionality of the specification. Because of this, translation into 3GL source code makes the system portable to several environments and able to handle large volume of transactions. These are the major weaknesses of 4GLs where their prototype cannot be ported to several environments except proprietary ones. They cannot handle also large volume of transactions.

Besides from translating 3GL source codes, the paradigm also supports translation of the specification into a prototype. The prototype is a working model of the system being designed. It is exercised to find out the characteristics and behavior of the system. This is a well known technique in engineering where a model is built first before the actual product is manufactured. As shown in Fig. 1, the prototype is validated to check if it satisfies the system requirements. Since the requirements are informal and sometimes incomplete, a series of iteration takes place (where the specifications are modified based on the results of exercising the prototype) until the user is satisfied with its performance. Notice that in the paradigm, it allows the target 3GL source code to be validated against the requirements. This process can be iterated several times until its performance reaches users satisfaction.

Maintenance is handled in the paradigm by expressing the changes in the requirements of the formal specifications or in the translated new specification into a target 3GL source code or into a prototype. Then, they can be exercised to find out if they satisfy the new requirements.

Maintenance also encompasses changes of computer machine, operating system or programming language. For these changes, the same specification is run through the translator by specifying only the new target source code and/or new environment. Thus an application designed using this paradigm can be ported from one computing environment to another with little or no effort.

Fig. 1 shows that the paradigm supports specification and prototyping on inexpensive microcomputer or workstations. The translation can be run on 286/386-based microcomputers (for their speed and memory) while the 3GLs can be run in a microcomputer, minicomputer or mainframe environments.

From this software paradigm, we developed a software engineering environment (SEE) aimed at building application programs for business data processing domains called 5GSA. 5GSA is a user-friendly front-end containing 3 key components:

- An "engine" which applies set of rules stored in a knowledge base in order to produce a program implementation.
- A set of accessible templates which serve as the knowledge base of the engine.
- A parser to transform the human-developed specifications into a structure that can be processed by the templates.

The specification languages consist of the Application Description Language (ADL), Data Definition Language (DDL), Page Definition Language (PDL), Program Specification Language (PSL) and Schema Change Language (SCL). These languages are collectively called the Data Processing Specification Outlining Language (DPSEL) and will be discuss more in the next section. The model on which DPSOL is based to describe information systems is discuss in the next section. Also discuss is the general description of the system and semantics of DPSOL.

The Requirements Specification Language

Other current works in requirements specification language focus on validating formal descriptions using mathematical rules while others concentrate on automatic transition from informal to formal descriptions. Our approach lets the user build the description through a formal specification language and then by means of a specify-validate-modify loop (either the prototype or the translated 3GL) the complete specification is derived.

Model of Information Systems

The most common representation of information flow is the data flow diagram (DFD). It is a graphical technique where data are shown to go through a sequence of transformations or processes. Following a path in the DFD represents information flow. Although some CASE tools have used DFD to
specify the information system to be designed, we extend DFD through DPSOL by specifying the information system in a more detailed information.

To illustrate this, let us discuss a hypothetical information system, see Fig. 2. In this DFD, there are 4 processes (P1, P2, P3, P4), 3 users (U1, U2, U3) and 2 files (F1, F2). U1 invokes P1 which in turn causes P2 to execute. P2 also initiates P4 which in turn writes to file F2. U3 requests P3 to read file F2 which afterwards prints a report to U2.

DPSOL views information system as consisting of a set of tuples, an object and its associated process. An object could be a user or a file. In the hypothetical information system given in Fig. 2, the following is the set of tuples: <U1,P1>, <U2,P2>, <U3,P3>, <F1,P2>, <U2,P3>, <F2,P3> and <F2,P3>. This is shown in Fig. 3.

For the tuples <U1,P1>, <U2,P2>, <U3,P3> and <U2,P3>, DPSOL defines them as an object served by a process or processes. The specification language in DPSOL that does this is the ADL. In ADL, we can specify which among the users are served by what process. In our hypothetical example, U1 is served by P1 only while U2 is served by P2 and P3. We can also identify the intended users of the system and specify what particular process will serve that user. For example in an inventory system, the set of processes that serve the receive clerk is different from that of the warehouse manager. The indirect effect of this specification is that we can define at this level the set of valid operations or privileges for each user.

For the tuples <F1,P2>, <F2,P4> and <F2,P3>, DPSOL specifies them as the set of processes that manipulate files. The particular specification language that does this is the PSL. Moreover, the PSL integrates in one complete system all the processes involved. In our hypothetical example, PSL defines P1, P2, P3 or P4 and its associated files in one program. The effect of this definition is the linking of the different modules comprising the information system being defined. A side-effect of this definition is that the user will be restricted as to what files he is authorized to access, since ADL defines the tuple <user, process> and the files are defined through the DDL. PSL defines the integration of <file, process> tuples, the set of files a user can access is transitively derived.

DPSOL through DDL can define the structure of files in a relational data model. By identifying primary and foreign keys, different view tables can be derived as specified in the PSL. Maintenance of the base tables (files) from this view is allowed and DDL checks for referential integrity, i.e., does not produce an inconsistent database. DDL also checks for computational integrity, i.e., checks for values of derived field when the value of a data field in a record is changed. In PSL, the referential and computational integrity checks are done automatically because the files are specified through the DDL where the referential and computational checks are defined.

PSL is used to describe report formats and screen designs. In a DFD this is just a labeled report or message in a screen without the means to describe the format.

SCL is provided if there are changes to the logical schema of the database model. The SCL hides the effect of the changes done to the logical schema from the applications that uses the database. Thus the ADL and PSL are shielded from the changes to the logical schema through the SCL.

An Overview of the DPSOL Syntax and Semantics

DPSOL is a set of specification language for outlining descriptions of information systems in business. It is intended to be used by systems analysts. Because DPSOL can be interpreted into a prototype or translated into a target 3GL source code, it is an executable requirements specification language.

DPSOL may be used in combination with any type of programs such as batch, interactive or report and therefore only "basic" information about the system needs to be stated. All "derived" information is handled by the SGS knowledge base.

The objective of DPSOL is to be able to express in syntactically analyzable form the system description and be able to translate the description into a prototype or into the target 3GL source code.

System description may be divided into four major aspects:

a) System Input/Output Flow  c) Data Structure  d) Documentation

b) Data Derivation

DPSOL contains objects, processes and tuples which permit these different aspects to be described. The system input/output flow aspect of the system deals with the interaction between the target system and its environment. This includes interaction available on the terminal, screen/page attributes, users of the system, program types and files used.

The data derivation aspect specifies which data objects are involved in particular processes in the system. It is concerned with what information is used, updated and/or derived, how this is done, and by which processes.

The data structure aspect includes all the relationships and their capabilities which exist among the data used and/or manipulated by the system as seen by the users of the system.

The documentation aspect requires, in addition to the description of the system being designed, documentation of
the project as well as the users of the system.

The first 3 aspects of system description is achieved in DPSOL through ADL, DDL, PDL, PSL and SCL. The ADL and PDL specify the system input/output flow. The DDL provides the user to describe the data structure. In case, a change is made in the data structure, the SCL is used to modify such changes. The PSL describes the system's data derivation techniques. The next section presents a subset of an example specification of an inventory application written in DPSOL.

An Example Requirements Specification Language

Given below is an ADL for a sample inventory application.

```
1 application 'INVENTORY' for 'SOFTWARE BREWERS, INC';
2 var terminal
3 on 3270 record-level interaction,
4 page width is 132;
5 endvar.
6 var user
7 on 3270 WhseMgr, ReceiveClerk, IssueClerk;
8 endvar.
9 var file
10 PRODUCT title 'Product Master file';
11 RECEMASTE title 'Receipts Master File';
12 RECEDET title 'Receipts Detail File';
13 ISSUEMAS title 'Issues Master File';
14 ADJSMENT title 'Adjustment File';
15 endfile.
16 var form
17 PRODMAIN title 'Product Maintenance' browsed by IssueClerk;
18 maintained by WhseMgr;
19 RECEP title 'Receipt Maintenance' browsed by WhseMgr;
20 maintained by IssueClerk;
21 ISSMAINT title 'Issue Maintenance' browsed by WhseMgr;
22 maintained by IssueClerk;
23 ADJSMENT title 'Adjustment Maintenance' maintained by WhseMgr;
24 endform.
25 var report
26 PRODSTAT title 'Product Status Report' run by WhseMgr.
27 RECEPST title 'Receipt Status Report' run by WhseMgr;
28 ISSMAINT title 'Issue Status Report' run by WhseMgr;
29 endreport.
30 endapplication.
```

The ADL consists of the following fragments: application, terminal, user, file, form, report, batch and group. The application fragment specifies the name of the application to be maintain. The terminal fragment identifies the kind of terminal used, its interaction and attributes such as page width, colors, etc. The user fragment identify only the allowed user in a given application. The file fragment enumerates the files to be used. The form, report and batch fragments specify the types of program whether interactive, report or batch respectively. However, for each type, the user is allowed only with specific privileges or operations. As shown in the example, form PRODMAIN (in the forms fragment) can be browsed by the issue clerk and maintained only by the warehouse manager. The group fragment builds menus on the application by grouping the user and his privileges in a single menu, thus ensuring security of user access.

A sample DDL for RECEDET file.

```
1 define file RECEDET
2 ReceiptNo type numeric, pic '999999';
3 prompt is 'Receipt number';
4 ItemNo type numeric, pic '999999.99';
5 primary;
6 QtyRecvd type numeric, pic '999999.99';
7 prompt is 'Quantity received';
8 display is 'Z.ZZ9.99';
9 positive;
10 nonnegative;
11 increases PRODUCT QtyOnHand;
12 increases PRODUCT StockVal;
13 UnitPrice type numeric, pic '999999.99';
14 prompt is 'Unit Price',
15 display is 'ZZZ.ZZ9.99';
16 positive;
17 LineAmt type numeric, pic '999999.99';
18 prompt is 'Line Amount',
19 computes as QtyRecvd * UnitPrice;
20 increases RECEMASTE TotalAmt;
21 increases PRODUCT StockVal;
22 enddefine.
```

The DDL defines all the files in the database. Referential and computational integrity checks are supplied to produce a consistent database. A referential integrity check is shown in line 4, which states that 'ItemNo references PRODUCT'. This means, any value of ItemNo in file RECEDET must also appear in the PRODUCT file. A computational integrity check is done in lines 20-21. In line 20, TotalAmt in file RECEMASTE increases if LineAm is computed. Likewise in line 21, StockVal increases in the PRODUCT file.

Attributes are also added for each field names. Consider line 3, the prompt 'Receipt Number' will be displayed whenever a message is given concerning ReceiptNo. Line 9 which states, 'display is ZZZ.ZZ9.99' will be used as the displayed data type whenever the value of QtyRecvd is shown on the screen.

A PDL for Inventory System,

```
1 page RECEIPT
2 image ReceiptHdr at 3
3 beginimage
4 SOFTWARE BREWERS, INC
5 MATERIAL RECEIPT
6
7 RECEIPT NO. : -ReceiptNo
8 VENDOR NO. : -VendorNo -CompanyNam
9
10 QUANTITY UNIT
11 ITEM NO DESCRIPTION RECEIVED PRICE AMOUNT
12
13 endimage.
14
15 image ReceiptDetail at 15, repeats 5
16 beginimage
17 -ItemNo -Desc -QtyRecvd -UnitPrice -LineAmt
18
19 endimage.
20
21 endpage.
```
The PDL contains the pages of report formats or screen designs. These pages can be called by various programs and reports. Such technique allows modularity of pages. In line 1, the page is named RECEIPT, with single-multi images. The single image ReceiptHdr in line 2, is the heading of page RECEIPT, while the multi-image ReceiptDetail in line 4, is the detail of the record ITEM for each ReceiptNo. PDL also allows multi-paging where various subpages can be embedded in a main page.

A sample program for Receipt Maintenance is shown as follows,

```
1 interactive program RECEIPT;
2   view
3     PRODUCT;
4   endview;
5 dialog
6   page
7     RECEIPT
8   endpage;
9 rule
10   Totahit increment by Lineht;
11 endrule
12 endprogram
```

The PSL is identified with the type of a program to be executed, the files & pages to be used and the rules to be validated. In the example, the type is 'interactive' (line 1), the file is PRODUCT (lines 2-3) and the page is RECEIPT (lines 4-7). Lines 8-9 show the rule to be validated. This means, the value of Lineht is always added to the current value of Totahit. Other rules are also supported such as initialization, sort, format, report break, protected field entry, mathematical computations, etc.

### Translation to Prototype or 3GL

In simple terms, automatic programming is concerned with mapping an informal or formal requirements specification into an implementation-specific program. Current transformation approaches take as input a specification written in a very high level language and by a series of transformations produce the target 3GL source code or low-level implementation. Our approach is translational rather than transformational. This means that the input requirements specification program is translated incrementally to the target 3GL source code. By a process of assembling these incremental produced fragments, the complete program is derived. The translation process obeys a set of rules which directs the translation. The set of rules is stored in a repository which maintains consistency in all of the translation processes.

### Templating Concept

Although templating is not a new concept, it has been extended in [8] to automatically generate 3GL source code given an input specification program. This technique was adopted in the DELPHI Project. The translation process selects the appropriate set of templates which are created for a particular target language.

Templates are outline of program objects and logic sets which produce a string of characters or fragments of code in a particular 3GL. The "blackboard" is a global data structure which contains the evolving code fragments. Initially, there will be fragments of code being written in the "blackboard". As more templates are called, these fragments of code assemble into several program modules. And then finally the modules are assembled into a single complete target 3GL source code.

As shown in Fig. 4, execution of templates may call various templates, which in turn may call other templates and so forth to any depth. Each template at some point is expected to terminate its execution and return control to the template that called it. During execution of a called template, the calling template is temporarily halted. When execution of the called template is completed, execution of the calling template resumes at the point immediately following the call of the template. Note that the fragments of code generated by the called templates are stored in a "blackboard" reference through a name fragment. This pool of name fragments can be manipulated by any template either the calling template or any other called templates.

![Template calling sequence control](image)

Templates are written in a templating language. The language resembles like the structure of a 3GL. However, unlike 3GL, the templating language was primarily designed to manipulate symbols [8]. The symbols take the form of name fragments (prefixed with "^" or global strings (prefixed with "**")) and global strings (prefixed with "**, e.g., "**global string**"). The former is a variable name that can contain another variable name, while the latter is a dynamically created variable which is updated by a conventional assignment statement or by printing text into it. These symbols are reference by names, not on addresses. Consider for example, the variables CNT, Extractl and ScPicOut found in the following template statements.

```
!move "CNT + 1 to "CNT
!move "[ScPic] Extract"CNT to "Datatype
```

These statements increment "CNT, evaluate "[ScPic] Extract"CNT and assign its value to "Datatype in the "blackboard". Let us assume the values 0, Out and PIC X(3) for CNT, Extractl and ScPicOut, respectively. If we want to determine the value of "[ScPic] Extract"CNT, evaluation proceeds in the following manner. "CNT has a value 1 and concatenated to produce "[ScPic] Extractl. Then, "Extractl is evaluated into Out resulting to "ScPicOut. "ScPicOut becomes PIC X(3). Hence, "Datatype has now a value of PIC X(3).

The templating language supports the following features. It allows dynamic variable referencing. This feature is important because we do not know exactly what variables are needed by templates; code may change according to a certain variation. Another striking feature is the heavy use of dynamic global strings which survive through invocation. This is needed since code fragments must float prior to being collected and formed
into a complete source program. A third key feature is the ability to invoke templates from other templates. This is important so that templates can be modular. The last feature provides support for format, alignment and adjustment of multi-line print blocks. Formatting is crucial, since source code must be properly aligned for clarity and correctness, as with RPG. Furthermore, if templates can be used to produce source codes, it must also produce documentation from the specification program. Detailed discussion of templates and templating language is found in [8].

Reusable Logic Sets

Reusable logic refers to the process of reusing code fragments, objects and logic sets where they are considered as software building blocks. A logic set is an outline of how a program or a part of a program is written to solve a particular function. In programming, outlines occur many times when particular functions must be coded, such as when generating business reports, scheduling algorithms for operating systems and searching a problem space in expert systems. Because these functions recur in particular problem domains, they are included in the set of reusable components.

Current reusability approaches are limited to the use of subroutine libraries & macros, modular programming and object-oriented programming systems (OOPS). Libraries allow the reuse of routines for programs written in a particular language, however they are highly inflexible since they work only with specific data types or global variables. Macros on the other hand, do not have the notion of objects and cannot reuse logic sets. The reusable code fragments are tightly linked to an environment and to write one must begin from scratch.

Modular programming initiated by MODULA-2 and ADA [2] enforce modularity and information hiding so that procedures or packages can be reuse by other programs. OOPS, such as Smalltalk and C++ [3, 7] follows a similar approach. They define low level objects such as semaphores & data structures and high level objects such as files & CPU. However, these approaches cannot reuse logic sets and do not automate coding of instructions to objects. Finally, they are a replacement of, rather than an add-on to, available expertise.

CASE tools feature visual programming, rapid prototyping, and translation to 3GL [4], but only a few support reusability [6]. While some of these tools allow the reuse of program code, they do not support the reuse of the logic behind the code.

DELPHI's approach to reusability is through the use of templates. They were extended by allowing the reuse of code fragments, objects and logic sets (see Fig. 5).

As shown in Fig. 5 the reusable logic sets act as a knowledge base containing the rules for assembling templates to produce a target 3GL source code given the specification of the application to be implemented. The set of rules once selected will invoke templates for reusable fragments and templates for reusable objects.

Mapping to Low-level Implementation

Mapping the description of a system into a prototype is achieved as follows. Using as input the DPSOL, a pattern matcher directive (PMD) is triggered to scan the system description written in DPSOL. A PMD is a preprocessor within the prototype compiler that locates text patterns and execute some instructions when such patterns are detected. Consider for example our sample inventory system, specifically the program for Receipt Transaction Maintenance, when the PMD scanned line 1, it found a keyword 'interactive'. This keyword will execute sets of templates specifically designed for interactive programs. The execution will create repository of information pertaining to a particular system. This repository is then accessed by the prototype templates which assembles and expands small fragments of code to a bigger code fragments. Through an integrating template, code fragments are further assembled to form the 5GL prototype language. This is then ready to be interpreted. At this stage the user can view and test the system prototype. Further modification can be made by changing the system description written in DPSOL and regenerating the new prototype.

The translation of the DPSOL into a low-level specific software system particularly in the target 3GL/environment starts when the repository is accessed by the target 3GL translator. By following the same process of assembling and expanding code fragments as in prototyping, the source code is produced in a specific target 3GL/environment. The generated 3GL source code can then be compiled and executed in its target environment (mainframe, miniframe, or microcomputer).

As an example let us discuss the translation of a DPSOL procedural component, in particular a flow-of-control statements, into a target 3GL source code. Shown below is a DPSOL IF statement which will be translated to COBOL,

\[
\text{IF } B < C \text{ and } A > C \text{ then}
\]

\[
\text{Set } B \text{ to } C; \]

\[
\text{EndIf;}
\]

Initially, a sequence of template calls is produced by following the grammar definition of DPSOL. That is, the parser searches for the production rules until a given terminal symbol is encountered or a production rule is satisfied. If satisfied, a template call is created invoking the template that correspond to the triggered production rule or terminal symbol. This process repeats until the DPSOL program is parsed and all template called are completely generated.

As an example let us use a grammar definition following a subset of DPSOL specification language. The grammar shown below is used to parse DPSOL fragments for IF statements.
A template call can be interpreted in the following manner. A template name is distinguished with a call consists of a template name and a set of parameters. A parameter), variable names and/or template names. A template call is generated, i.e., %Factor([ l ], [Varl ]). Hence, by following the same process, the IF statement produces an parser rules, which is found in the STATEMENT production. The parser rule now found a terminal symbol, @VARDEC, therefore a expression consisting of 18 template calls shown created. The 'stream' declaration allows duplication of values particular statement or expression. Such assumption takes care of referencing the values uniqueness of values for production rules triggered in a production once a template is created. The 'stream' declaration allows duplication of values in a triggered production rule and referencing is not allowed for these values.

By applying the grammar definition to our example IF statement, the sequence of template calls is done as follows. The parser will search for an 'IF' keyword in the production rules, which is found in the STATEMENT production. The parser will evaluate if the condition for STATEMENT production is satisfied. In this case it is not, so the CONDITION production rule is evaluated. Again, it is not satisfied, so the parser searches for COMPARISON, then to EXPRESSION, and finally to FACTOR. The FACTOR production rule found a terminal symbol, @VARDEC, therefore a template call is generated, i.e., %Factor([1], [Var1]). Hence, by following the same process, the IF statement produces an expression consisting of 18 template calls shown as follows.

```plaintext
1  Factor ([1], [Var1])
2  Expression ([1], [Factor1])
3  Factor ([2], [Var2])
4  Expression ([2], [Factor2])
5  Comparison ([1], [Expression1], ['+'], [Expression2])
6  Factor ([3], [Var3])
7  Expression ([3], [Factor3])
8  Factor ([4], [Var4])
9  Expression ([4], [Factor4])
10 Comparison ([1], [Expression3], ['+'], [Expression4])
11 Condition ([1], [Comparison1])
12 MoreCompare ([1], [AND], [Condition1])
13 MoreCondition ([2], [Comparison2], [MoreCompare1])
14 Factor ([5], [Var5])
15 Expression ([5], [Factor5])
16 Statement ([1], [Var2], [Expression5])
17 ThenClause ([1], [Statement1])
18 ElseStatement ([2], [Condition2], [ThenClauses])
```

A template call can be interpreted in the following manner. A call consists of a template name and a set of parameters. A template name is distinguished with a "" sign. The parameter consists of the sequence number of the template (first parameter), variable names and/or template names. A template name when passed as a parameter is done by concatenating its name and the sequence number. Consider, line 1, the template name is FACTOR, the parameters are 1 (sequence number) and Var1 (variable name for B, assuming the variables were already declared). Likewise in line 2, the template name is EXPRESSION, and the parameters are 1 (sequence number) and FACTOR1 (a call to template FACTOR with sequence number 1).

Note that the expression is written in postfix format with a call to a template at the end. This format is used because of the prevalence of different operator hierarchies among 3GLs. In the example, Comparison1 (B NOT > C) and Comparison2 (A NOT < C) are assembled first. Condition1 is set to Comparison2. A call to MoreCompare passes a logical operator and a condition. MoreCompare1 becomes "AND" concatenated with Condition1. Condition2 which is a compound condition, is set to the concatenation of Comparison1 and MoreCompare1 enclosed in parenthesis. Statement1 is passed to ThenClause1. Then Statement2 is assembled by concatenating Condition2 and ThenClause1. The equivalent COBOL code will then be,

```cobol
IF (B NOT > C) AND
A NOT < C)
COMPUTE B = C.
```

So to assemble a complete COBOL program, the translator starts by generating source code fragments for assignment, flow-of-control, file definitions, image definitions, and others. These code fragments will be assembled to form procedures and modules. Finally, a main integrating template for a particular application will assemble these procedures and modules into the main source program in the target 3GL/environment. Note that assembling of fragments of code is done in a bottom-up postfix manner.

**Conclusion & Future Directions**

We have presented DPSOL, an executable requirements specification language, which is used for specifying information system in the business data processing problem domain. We adopted the rapid prototyping approach instead of the automatic translation of the informal requirements into a formal specification language. In this approach, the user first tries out an initial specification based on the informal and sometimes incomplete requirements. By a series of specify-validate-modify loop, the user is able to exercise the prototype until he "sees" the required application system.

Having the desired specification, it can be used in generating the target 3GL/environment: Our approach is translational instead of transformational because of the simplicity and flexibility in generating very specific low-level implementation. Currently, we have sets of templates to generate the following 3GL and environments:

- **GICS COBOL**, **ANS COBOL 74**, **ANS COBOL 85**, **COBOL 74**, **RPG System/36**, **Turbo C**, **VSAM**, **ISAM**, **CICS**

**Future Directions**

- More 3GL and environments sets of templates are being designed and implemented currently. For the above target 3GL/environments, documentations can be automatically generated from DPSOL either in English or French.
Work is underway is studying how "reverse engineering" can be done on old and previous source code which is being used in some applications. If the specification can be extracted from a given source code, then conversion to another 3GL or machine or operating system can be done automatically.

References