

## Towards Quality of Software in TUG

CHIA-CHU CHIANG

*Department of Computer Science  
University of Arkansas at Little Rock  
Arkansas 72204-1099, U.S.A.  
E-mail: cxchiang@ualr.edu*

The quality of software depends on the effectiveness of the software development process. Existing software development processes are still not flexible or practical enough for developing a system that requires a mix of existing processes to be used in the production of that system. In this paper, a formal specification language, called TUG (Tree with Unified Grammar), is presented to support a software development process that accommodates conventional software development, operational specification, rapid prototyping via software transformations, software reuse, software testing, and program proofs of correctness. The development process with TUG can not only allow for a mix of existing development processes applied to a given system but also can be adapted to one of existing development processes. The combination of existing development processes takes advantage of the strengths of each process in the system. The software development process with the aid of the TUG specification language can aid the production of reliable and reusable programs.

**Keywords:** executable specifications, formal method, formal specifications, proofs, rapid prototyping, reuse, software transformations, TUG, waterfall model

### 1. INTRODUCTION

The conventional software development process has been criticized as not appropriate for developing evolutionary software. Errors are frequently not detected until the test phase. Alternative software development processes, such as rapid prototyping, operational specification, incremental development, and software reuse, respond to the needs of the conventional software development process. However, these existing software development processes are not flexible or practical enough for developing a system that requires a mix of existing development processes to be used in the production of that system. For example, a system that has a poorly understood user interface and poor decision support functions may require a process that combines rapid prototyping and incremental development. However, if the system has fully understood user decision support functions, the poorly understood user interface may cause the process to be the equivalent to a rapid prototyping process. A flexible and practical software development process should not only allow for a mix of existing development processes for developing a given system but should also be adaptable to one of the existing processes [1]. A combination of existing software development processes takes advantage of the strengths of each development process in a system. In this paper, we present a formal specification

---

Received February 11, 2003; revised June 11 & December 30, 2003; accepted February 2, 2004.  
Communicated by Meng-Chang Chen.

language, called TUG, to support a software system to be developed through the integration of existing software development processes, such as operational specification, rapid prototyping via software transformations, software reuse, and static and dynamic analysis of software.

The TUG specification language is a formal specification language based on DCGs (Definite Clause Grammars). The formality of the language allows for the direct execution of a TUG specification through an interpreter, which supports the operational specification process. Operational specification provides another means of improving the quality of the specifications and software system. But whereas the formality of the language improves the quality of the descriptions of user requirements, rapid prototyping via operational specification improves the understandability of the user requirements. Although the TUG specification language supports the operational specification process, the language still lacks notations for specifying the non-functional properties of software systems, such as the user interface and performance. To alleviate this problem, the TUG method allows a prototype to be automatically built. Then codes for the non-functional properties of the system can be added manually to the prototype for rapid prototyping purposes. The TUG method supports the construction of a prototype via software transformations. The underlying mathematical principles of DCGs provide a vehicle for developing a set of transformation rules used to generate a prototype in Prolog from a TUG specification. The resulting prototype can be executed for user feedback under the Prolog environment.

Reuse is believed to be one key to improving software development productivity and quality [2]. The reuse of software allows developers to spend less time creating new software. To facilitate software reuse, the TUG specification language allows a developer to write strongly typed and weakly typed specifications. In a strongly-typed specification, each terminal node is declared to be at most one type. In contrast to a strongly typed specification, a weakly typed specification allows for a terminal node belonging to more than one type. The application of weak typing, such as overloading, provides a way of constructing reusable specifications in the language. In addition, there often is a specification for reuse that is not in exactly the right form. The TUG method allows for some modification of the specifications. TUG achieves these goals by providing a renaming operator used to replace an old name with a new name. The language also provides an extending operator, used to expand some portions of the base specifications.

A specification in this language can be analyzed for detecting errors due to the formality of the language. Errors such as type inconsistency, redundancy, and conflict can be detected by analyzing the specification via inspection or review. In addition, the TUG specification language provides a way of proving the correctness of a specification against corresponding user claims by first translating the specification into a set of Horn clauses. The set of Horn clauses with a user claim are then proved for correctness via theorem proving. Proofs on a TUG specification demonstrate the consistency of the specification and also show that the specification satisfies the user claims. To support software testing, a set of test cases can be generated from a specification in the language.

The remainder of this paper is organized as follows. Section 2 reviews related works. Section 3 introduces the TUG specification language. An overview of the syntax and semantics of the language is presented. The language is illustrated through examples. The operational specification process that uses TUG is presented in section 4. Section 5

describes how a TUG specification can be transformed into a prototype in Prolog via a set of transformation rules. Section 6 describes how the language can be used to construct reusable specifications. The use of the language to support reuse is illustrated through examples. Section 7 shows how a TUG specification corresponding to the bubble sort problem can be constructed incrementally and addresses the proofs of TUG specifications against user claims by means of theorem proving. In section 8, we present how a TUG specification can be systematically mapped into a structured design and, subsequently, into a structured program. Finally, a summary is provided and future works discussed in section 9.

## 2. RELATED WORK

To facilitate a discussion of formal specification languages, existing formal specification languages may be divided into three categories: algebraic specification languages, model-based specification languages, and logic-based specification languages. Model-based specification languages specify system behavior in terms of tuples, relations, functions, sets, and sequences. Therefore, a model-based specification has two components: a set of states and the operations over the states. One popular model-based specification is the Z specification language, which is a formal specification language based on set theory and first-order logic [3, 4]. A specification in Z is presented as a collection of schemes which can be combined and used in other schemes. The schema is a diagrammatic notation for displaying the predicates that are used in defining operations and invariants. The top line of the schema introduces the schema name. The section above the middle line is called the signature (declaration) part of the schema. The purpose of the signature is to set out the names and types of variables which represent the states introduced in the schema. The section below the middle line is known as the predicate part of the schema. The predicate part of a schema is optional. The predicate sets out the operations over the states in the signature part. Z has been used as a specification language to formally describe and analyze the requirements and the design architectures of a variety of hardware and software systems [5]. However, Z has mainly been successfully used for the specification of sequential systems rather than concurrent systems because it lacks notation and timing to model concurrency. A number of researchers have attempted to overcome these difficulties, usually by combining Z with some other formalism, such as Petri Nets [6-8], temporal logic [9], or TLA [10] to specify dynamic properties. Examples of model-based specification languages include PAISLey [11, 12] and VDM [13-16].

Algebraic specification languages specify system behavior in terms of axioms and algebras. An abstract data type is specified by axioms relating the operations on that abstract data type. An algebraic specification has two parts: the abstract data type name with the names of associated operations on that abstract data type; and the axioms which relate the operations. One of the algebraic specification languages is OBJ [17, 18]. The OBJ specification can be executed by interpreting the equations of the OBJ specification as a left-to-right term rewriting system. Examples of algebraic specification languages include ASF [19], and Larch [20, 21]. The use of algebraic specification languages in software engineering is discussed in [22].

Logic-based specification languages specify system behavior using first-order predicate logic, Horn clauses, or higher order logic. A logic-based specification basically consists of two parts: the names of abstract data types, variables, functions, and predicates; and the predicates or Horn clauses which associate the relations on that abstract data type. Applications of logic-based specification languages include temporal logic [23] and interval temporal logic [24]. In [25], a first order interval logic based specification language, called TILCO, was proposed to specify, validate, and verify real-time systems. The operational validation of TILCO specifications is supported by the TILCO executor. The TUG specification language presented in this paper can also be regarded as a logic-based specification language that is based on definite clause grammars. A definite clause grammar expresses context-free rules as logic statements that allow for the use of arguments with non-terminals. A prototype derived from specifications in the language consists of definite clause grammar rules and normal Prolog goals that can be executed by a Prolog interpreter.

### 3. DESCRIPTION OF TUG

The TUG specification language consists of three parts: a name part, in which the title with input/output parameters are given; an analysis part, in which the input data is defined; and an anatomy part, in which the output data is generated.

The name part contains a module or scheme title with input/output parameters. The input/output parameters are enclosed in parentheses. The analysis part contains the rules for analyzing the input data. To analyze the input data, definite clause grammars (DCGs) [26] are used to represent the rules so that syntax analysis can be performed. Each rule of a DCG expresses a possible form of a non-terminal, as a sequence of terminals with optional constraints on the terminals and non-terminals. Non-terminal nodes in uppercase characters indicate constituents. A terminal node in lowercase characters indicates a token that must occur in the input data. A terminal node can be a literal, which is any string enclosed in a pair of quotes. A constraint wrapped in braces places conditions, such as type checking, on a terminal node. Table 1 shows all the operators used in the conditions. An input is parsed into a tree representation that takes the form of a Prolog list with a node name acting as the relationship symbol of the input data. This tree representation will then be the input to the anatomy analysis part of the TUG specification.

The anatomy part describes the output to be generated. The tree is fed into the anatomy part, and pattern matching is performed. The root node of the matched tree then becomes a name for the entire tree. Each non-terminal node becomes the name for that part of the entire tree. The non-terminal node names that part of the input data. The output data can then be synthesized or the new input data can be generated and sent back to the analysis part for further analysis; otherwise, execution terminates. Table 2 shows all the numeric and list operators used in the statements of the anatomy part to perform arithmetic and list operations.

A specification in TUG is structured with regular expression notations (union, Kleene closure, positive closure, and concatenation). Each non-terminal node structures its tree according to one of the regular expression notations. Adding tree structures to a

**Table 1. Operators used in the conditions.**

Operators	Description
any(t)	t belongs to any type
bool(t)	t is a Boolean
character(t)	t is a character
digit(t)	t is a digit
equal_to(t <sub>1</sub> , t <sub>2</sub> )	t <sub>1</sub> is equal to t <sub>2</sub>
float(t)	t is a float
follow(t <sub>1</sub> , t <sub>2</sub> )	t <sub>1</sub> follows t <sub>2</sub>
greater_than(t <sub>1</sub> , t <sub>2</sub> )	t <sub>1</sub> is greater than t <sub>2</sub>
integer(t)	t is an integer
length(t)	length of a string t
less_than(t <sub>1</sub> , t <sub>2</sub> )	t <sub>1</sub> is less than t <sub>2</sub>
lowercase_character(t)	t is a lowercase character
member(t <sub>1</sub> , t <sub>2</sub> )	t <sub>1</sub> is a member of t <sub>2</sub>
not(t)	negation of t
precede(t <sub>1</sub> , t <sub>2</sub> )	t <sub>1</sub> precedes t <sub>2</sub>
remainder(t <sub>1</sub> , t <sub>2</sub> )	remainder for integer division t <sub>1</sub> //t <sub>2</sub>
string(t)	t is a string
text(t)	t is a text
uppercase_character(t)	t is an uppercase character
word(t)	t is a word

**Table 2. Operators used in the statements.**

Operators	Description
+	addition
=	assignment
-	subtraction
&&	logical and
*	multiplication
	logical or
/	division
::	append
//	integer division
<>	list concatenation
%	remainder
~	string concatenation
#	length
call	function invocation
input	accept data from the keyboard
output	output data to the screen

specification allows a software developer to construct the specification in a structured way to deal with complexity. The union notation is indicated by a vertical bar sign (|) suffixed to the node name; thus,

```
FLAG |
    'on'
    'off'
```

indicates that *FLAG* is one of the alternatives, 'on' and 'off'. The concatenation notation is indicated by an ampersand sign (&) suffixed to the node name; thus,

```
NAME&
    last_name
        {string(last_name)}
    first_name
        {string(first_name)}
```

indicates that *NAME* is a concatenation of *last\_name* and *first\_name*. The *last\_name* and *first\_name* must be of string type. The Kleene closure notation (\*) indicates that there is zero or more elements over the node. Thus,

```
E_MAIL_BOX*
    e_mail
        {text(e_mail)}
```

indicates that *E\_MAIL\_BOX* contains zero or more e-mail messages, each of which is a text. The positive closure notation (+) indicates that there is one or more elements over the node; thus,

```
E_MAIL+
    MESSAGE*
        letters
            {letter(letters)}
```

indicates that *E\_MAIL* contains at least one *MESSAGE* which may contain zero or more letters. If a *MESSAGE* contains no letters, then the *E\_MAIL* is an empty e-mail.

#### 4. OPERATIONAL SPECIFICATION

The operational specification process for software development is illustrated in Fig. 1. From the user requirements, however incomplete or inexact, developers construct a TUG specification, which is a formal, executable specification. This specification is exercised to demonstrate system behavior. The developers and users can then validate whether the user requirements are met. When the validation and revision loop indicates a satisfactory specification, the developers can begin to consider design and implementa-

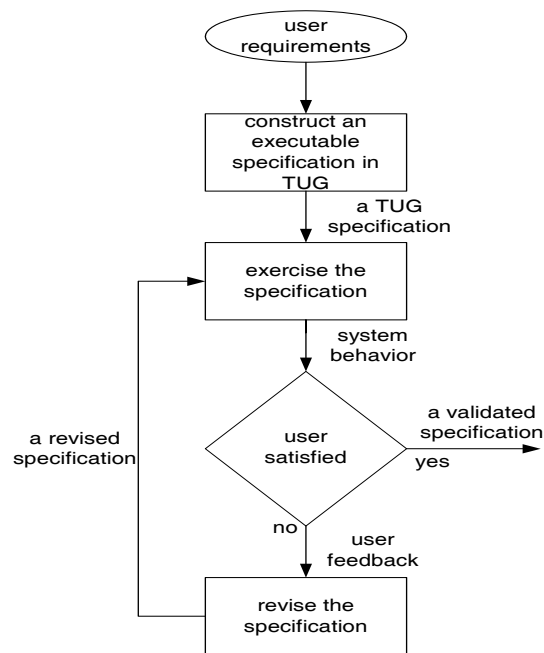


Fig. 1. The operational specification.

tion issues. To realize the language, an interpreter has been written in C. A specification in the language can be executed through the interpreter, which runs under the UNIX environment. Besides the interpreter, a scanner has been developed to detect any illegal tokens in the specification. A parser has been developed to detect any illegal statements that do not satisfy the syntax of the language. A syntax-based editor has been developed to help developers construct a TUG specification with guidance.

A TUG specification for a bubble sort program which sorts a sequence of integers in ascending order is presented as follows. In bubble sort, each pass bubbles the smallest element to its appropriate position. The specification of the bubble sort problem reads in a sequence of integers, sorts them, and prints them out in ascending order. The input is analyzed through case analysis on a sorted input and an unsorted input. With a test, the input is checked for the order of a pair of integers,  $x$  and  $y$ . If the integers,  $x$  and  $y$ , being compared are out of order, they are interchanged. The interchange operation is accomplished by splitting an unsorted input into five parts: *list1*,  $x$ , *list2*,  $y$ , and *rest\_of\_elements*. The two integers,  $x$  and  $y$ , are not in ascending order and need to be switched. The sublist *list1* contains the integers before  $x$ . The sublist *list2* contains the integers between  $x$  and  $y$ . The sublist *rest\_of\_elements* contains the rest of the integers in the input list. The sublists can be empty in the following cases:

1. If  $x$  is the first item in the unsorted input, *list1* is an empty list.
2. If  $x$  and  $y$  are adjacent, *list2* is an empty list.
3. If  $y$  is the last item in the unsorted input, *rest\_of\_elements* is an empty list.

Once  $x$  and  $y$  are located, they are swapped. The process is repeated, so that the next smallest integer is placed in the correct position until the input is sorted. As soon as the input is sorted, each item is printed out.

To construct a TUG specification module for the bubble sort problem, a module name *bubbles\_sort* is entitled with an input parameter, *SEQUENCE*. In a TUG specification module, a module name must be in lowercase characters, and the arguments must be in uppercase characters. The input data sequence, *SEQUENCE*, is analyzed in the analysis part that begins with a keyword, 'ANALYSIS'. The input data sequence can be either unsorted or sorted. Thus, *SEQUENCE* node has two children: *UNSORTED* and *SORTED*. The *SEQUENCE*, *UNSORTED*, and *SORTED* are all non-terminals indicating constituents. To define an unsorted sequence, the terminals, *list1*,  $x$ , *list2*,  $y$ , and *rest\_of\_elements*, are defined as children of the *UNSORTED* node. In addition, an unsorted sequence of integers must contain a pair of integers,  $x$  and  $y$ , where  $y$  is greater than  $x$ . These constraints associated with  $x$  and  $y$  are given in a pair of curly braces, respectively. In TUG, all the terminals defined in the analysis part must be in lowercase characters in order to distinguish them from non-terminals. A sorted sequence, *SORTED*, contains one child, namely, *ASCENDING\_SEQUENCE*, indicating that the sequence is already sorted in ascending order. Since the sequence is defined as having one or more elements in the input sequence, a positive closure notation (+) is suffixed to the non-terminal node, *ASCENDING\_SEQUENCE*.

The anatomy part of the specification begins with a keyword, 'ANATOMY'. In the anatomy part, all the nodes become terminals. Only temporary variables are in uppercase characters. An unsorted sequence, *unsorted*, will be rearranged by swapping  $x$  and  $y$  in order to make the sequence a sorted sequence. To swap  $x$  and  $y$ ,  $y$  is appended (::) to the head of *list2* to make a new temporary variable, *T\_L1*. Following that,  $x$  is appended to the head of *rest\_of\_elements* to make another new temporary list, *T\_L2*. Finally, string concatenations (<>) are performed on *list1*, *T\_L1*, and *T\_L2* to make another new list, *T\_L*. In *T\_L*,  $x$  and  $y$  are already swapped. In TUG, temporary variables in the anatomy part are all in uppercase characters. The swapping process is repeated until the sequence is sorted. Once a sequence is sorted, the sorted sequence, *sorted*, is sent to the display.

```

MODULE bubble_sort(in: SEQUENCE)
  ANALYSIS
    SEQUENCE |
      UNSORTED&
        list1
        x
        {integer(x)}
        list2
        y
        {integer(y),
         greater_than(x, y)}
        rest_of_elements
      SORTED&
        ASCENDING_SEQUENCE+
        element

```

```

                                {integer(element)}
END OF ANALYSIS;
ANATOMY
  Sequence |
    unsorted&
      T_L1 = y :: list2
      T_L2 = x :: rest_of_elements
      T_L = list1 <> T_L1 <> T_L2
      call bubble_sort(T_L)
    sorted&
      ascending_sequence+
        output element
        output ' '
  END OF ANATOMY;
END OF MODULE bubble_sort.

```

Suppose there are seven integers to be sorted. The input data are 10, 4, 18, 23, 34, 4, and 56. At the end of the first pass through the list, the smallest element 4 is sorted into the correct position. The process is repeated, the next smallest element 4 is sorted into the correct position, and so on until the list is sorted in ascending order. The results for each pass are shown below:

```

bubble_sort → [10, 4, 18, 23, 34, 4, 56]
             → [4, 10, 18, 23, 34, 4, 56]
             → [4, 4, 18, 23, 34, 10, 56]
             → [4, 4, 10, 23, 34, 18, 56]
             → [4, 4, 10, 18, 34, 23, 56]
             → [4, 4, 10, 18, 23, 34, 56]

```

In the first pass, a pair of integers 10 in  $x$  and 4 in  $y$  is found to be out of order. *List1* and *list2* contain no integers. The *rest\_of\_elements* sublist contains the integers 18, 23, 34, 4, and 56. The values of  $x$  and  $y$  are switched, so that the integer 4 is stored in the first position. In the second pass, the integers 10 in  $x$  and 4 in  $y$  are found to be out of order and are switched. In the third pass, the integers 18 in  $x$  and 10 in  $y$  are found to be out of order and are switched. In the fourth pass, the integers 23 in  $x$  and 18 in  $y$  are found to be out of order and are switched. In the fifth pass, the integers 34 in  $x$  and 23 in  $y$  are found to be out of order and are switched. Finally, the entire input list is sorted in ascending order, which is 4, 4, 10, 18, 23, 34, and 56.

## 5. RAPID PROTOTYPING VIA SOFTWARE TRANSFORMATIONS

A prototype can be used to explore evolutionary user requirements that should be built cheaply and quickly. A rapid prototyping approach via software transformations is used to achieve this goal. The idea of prototyping via software transformations is not new [27]. In our work, a specification is transformed into an executable program in Prolog by

applying a set of transformation rules. The program is then considered as a prototype to be exercised by the developer and the user. This approach allows developers to ignore low level details during implementation and optimization, such as the selection of data structures and algorithms due to the features of the Prolog programming language. A long waiting time for regenerating a new prototype is avoided following a user request to exercise the prototype after a specification is updated. To avoid complete retransformation, a Change Request Script (CRS) is written and used to update the prototype only in response to the revised specification. A CRS provides a *replace ... with ...* facility to replace a node description with another one in a TUG specification. A CRS is only used when a minor revision needs to be made to the nodes in a specification. Finally, prototype construction and specification acquisition are combined to handle changes in the user requirements. Whenever there is a specification change, the prototype is updated to respond to the change. The TUG specification language may not provide abstractions for prototyping some features of systems, such as the user interface, timing, parallelism, concurrency, and performance. Therefore, our approach is designed to allow the generated prototype to be manually extended or modified in a modular manner. The specification in the language is written in modules in terms of the language patterns that support module independence. The prototype is then derived in a modular way, which supports easy modification of the prototype.

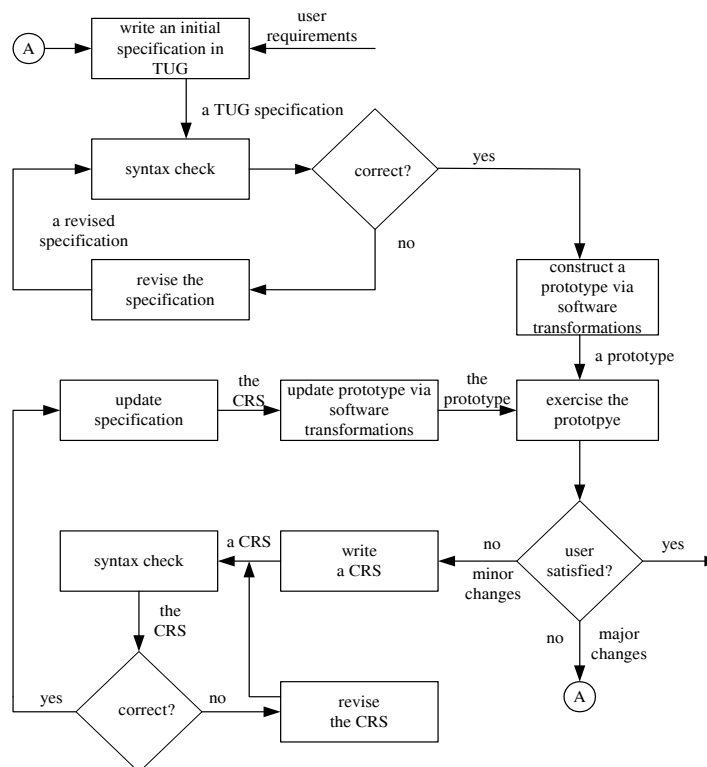


Fig. 2. Rapid prototyping cycle via software transformations.

The procedure for realizing a prototype is shown in Fig. 2. User requirements are first written into a specification in TUG. The specification need not necessarily be complete, precise, and correct, based on the user requirements after the first attempt. However, the specification should confirm the syntax of the language in order to be further processed. Next, a prototype is derived from the specification via correctness preserving software transformations. The prototype is exercised by the developer and the user. Feedback is obtained to decide whether the change is minor or major. If the modifications require that the nodes be modified, extended, relaxed, or refined, a minor change is suggested. A CRS specifying the change is then used to update the specification and the prototype. If a major change is needed, the developer may rewrite the specification and rederive a new prototype. A major change may involve the structure of the specification to be modified. This prototyping process continues until the requirements have been thoroughly exercised and the user is satisfied with the demonstrated behavior of the prototype. The result of prototype evolution is a set of modular TUG specifications for the proposed system. In addition, a set of CRSs record the design decisions made during the transformations.

The following section discusses how a prototype in Prolog can be generated via software transformations in the rapid prototyping process as depicted in Fig. 2. Throughout this paper, we will use the *bubble\_sort* specification specified in section 4 to illustrate how TUG supports the automated construction of a prototype in Prolog and the proofs of specifications in the language. In the first phase of the transformation, the input specification is transformed into an intermediate form using DCGs. In the second phase of the transformation, a prototype in Prolog is derived from the intermediate representation using DCGs via a set of transformation rules and library functions. The prototype is then exercised to demonstrate the system behavior in the Prolog environment. A driver that reads in the input data and then calls the main program with parameters needs to be constructed manually. The set of transformation rules is given below. The conventions are as follows:

- C is a finite set of condition tests and has the form  $\{c_1, c_2, \dots, c_n\}$  with  $n \geq 1$ , where  $c_i$  is a TUG condition test;
- Y is a finite set of dummy non-terminal or terminal nodes and has the form  $\{y_1, y_2, \dots, y_n\}$  with  $n \geq 1$ , where  $y_i$  is a dummy non-terminal or terminal node;
- names of predicates in Prolog are in all in lower case letters;
- names of variables in Prolog are in all in upper case letters;
- Q is a finite set of Prolog procedure calls and has the form  $\{q_1, q_2, \dots, q_n\}$  with  $n \geq 1$ , where  $q_i$  is a Prolog predicate for which a Prolog definition has been given; and
- $\langle \rangle$  encloses optional syntactic items.

The following four rules translate the analysis part of the specification into DCGs. Each non-terminal node in the analysis tree structures its subtrees according to one of the structuring notations. Each structuring operation can be transformed into a DCG form by applying the following four rules in a straightforward manner. A DCG usually has the following form:

$$\alpha \rightarrow \beta \{ \text{constraints} \},$$

where  $\alpha$  and  $\beta$  are constant strings, terminals, and non-terminals. A constant string indicates a constant value in the input sequence. A terminal indicates a value in the input sequence. A constraint rule associated with a terminal requires that the value satisfy the constraint. Constraints are optional. Non-terminals indicate constituents.

#### Rule 1

$$\begin{array}{l}
 \alpha \mid \\
 \beta_1 \quad \langle \{\Phi_1\} \rangle \\
 \beta_2 \quad \langle \{\Phi_2\} \rangle \\
 \dots \\
 \dots \\
 \beta_n \quad \langle \{\Phi_n\} \rangle
 \end{array}
 \quad \begin{array}{l}
 \text{where } \alpha \text{ is a non-terminal node} \\
 \beta_i \text{ is a non-terminal or terminal node with condition tests} \\
 \Phi_i \subseteq C
 \end{array}$$

---


$$\begin{array}{l}
 \alpha \rightarrow \beta_1 \langle \{\Phi_1\} \rangle \\
 \alpha \rightarrow \beta_2 \langle \{\Phi_2\} \rangle \\
 \dots \\
 \dots \\
 \alpha \rightarrow \beta_n \langle \{\Phi_n\} \rangle
 \end{array}$$

In Rule 1, a union non-terminal node  $\alpha$  in the analysis tree indicates that  $\alpha$  is one of the alternatives,  $\beta_1, \beta_2, \dots, \beta_n$ . If  $\beta_i$  is a constant string enclosed in a pair of single quotes, then there is no  $\Phi_i$  associated with  $\beta_i$ . Each translated DCG represents an alternative.

#### Rule 2

$$\begin{array}{l}
 \alpha \& \\
 \beta_1 \quad \langle \{\Phi_1\} \rangle \\
 \beta_2 \quad \langle \{\Phi_2\} \rangle \\
 \dots \\
 \dots \\
 \beta_n \quad \langle \{\Phi_n\} \rangle
 \end{array}
 \quad \begin{array}{l}
 \text{where } \alpha \text{ is a non-terminal node} \\
 \beta_i \text{ is a non-terminal or terminal node with condition tests} \\
 \Phi_i \subseteq C
 \end{array}$$

---


$$\alpha \rightarrow \beta_1 \langle \{\Phi_1\} \rangle \beta_2 \langle \{\Phi_2\} \rangle \dots \beta_n \langle \{\Phi_n\} \rangle$$

In Rule 2, a concatenation non-terminal node  $\alpha$  in the analysis tree indicates that  $\alpha$  is a concatenation of  $\beta_1, \beta_2, \dots, \beta_n$ . If  $\beta_i$  is a constant string enclosed in a pair of single quotes, then there is no  $\Phi_i$  associated with  $\beta_i$ . Each translated DCG represents a concatenation form.

## Rule 3

$$\begin{array}{l}
 \alpha^* \\
 \beta_1 \quad \text{where } \alpha \text{ is a non-terminal node} \\
 \quad \langle \{\Phi_1\} \rangle \quad \beta_i \text{ is a non-terminal or terminal node with condition tests} \\
 \beta_2 \quad \Phi_i \subseteq C \\
 \quad \langle \{\Phi_2\} \rangle \\
 \dots \\
 \dots \\
 \beta_n \\
 \quad \langle \{\Phi_n\} \rangle
 \end{array}$$


---

$$\begin{array}{l}
 \alpha \rightarrow [ ] \\
 \alpha \rightarrow \beta_1 \langle \{\Phi_1\} \rangle \beta_2 \langle \{\Phi_2\} \rangle \dots \beta_n \langle \{\Phi_n\} \rangle \alpha
 \end{array}$$

In Rule 3, a Kleene closure non-terminal node  $\alpha$  in the analysis tree indicates that  $\alpha$  is a sequence of zero or more occurrences of  $\beta_1, \beta_2, \dots, \beta_n$ . If  $\beta_i$  is a constant string enclosed in a pair of single quotes, then there is no  $\Phi_i$  associated with  $\beta_i$ . Two translated DCGs represent a Kleene closure form.

## Rule 4

$$\begin{array}{l}
 \alpha^+ \\
 \beta_1 \quad \text{where } \alpha \text{ is a non-terminal node} \\
 \quad \langle \{\Phi_1\} \rangle \quad \beta_i \text{ is a non-terminal or terminal node with condition tests} \\
 \beta_2 \quad \Phi_i \subseteq C \\
 \quad \langle \{\Phi_2\} \rangle \\
 \dots \\
 \dots \\
 \beta_n \\
 \quad \langle \{\Phi_n\} \rangle
 \end{array}$$


---

$$\begin{array}{l}
 \alpha \rightarrow \beta_1 \langle \{\Phi_1\} \rangle \beta_2 \langle \{\Phi_2\} \rangle \dots \beta_n \langle \{\Phi_n\} \rangle \\
 \alpha \rightarrow \beta_1 \langle \{\Phi_1\} \rangle \beta_2 \langle \{\Phi_2\} \rangle \dots \beta_n \langle \{\Phi_n\} \rangle \alpha
 \end{array}$$

In Rule 4, a positive closure non-terminal node  $\alpha$  in the analysis tree indicates that  $\alpha$  is a sequence of one or more occurrences of  $\beta_1, \beta_2, \dots, \beta_n$ . If  $\beta_i$  is a constant string enclosed in a pair of single quotes, then there is no  $\Phi_i$  associated with  $\beta_i$ . Two translated DCGs represent a positive closure form.

To demonstrate the use of transformation rules presented in this section, the *bubble\_sort* specification presented in section 4 is used as an example. The same specification will also be used as an illustration in sections 6 and 7. The application of Rules 1-4 to the analysis tree of the *bubble\_sort* specification produces the following results.

(1) SEQUENCE  $\rightarrow$  UNSORTED

- (2) SEQUENCE  $\rightarrow$  SORTED  
 (3) UNSORTED  $\rightarrow$  list1  
     x {integer(x)}  
     list2  
     y {integer(y),  
       greater\_than(x, y)}  
     rest\_of\_elements  
 (4) SORTED  $\rightarrow$  ASCENDING\_SEQUENCE  
 (5) ASCENDING\_SEQUENCE  $\rightarrow$  element {integer(element)}  
 (6) ASCENDING\_SEQUENCE  $\rightarrow$  element {integer(element)}  
     ASCENDING\_SEQUENCE

The following four rules translate the anatomy tree of the specification into DCGs. The rules are similar to the rules for translating the analysis tree. The difference is that we use the “:-” symbol instead of the “ $\rightarrow$ ” symbol. The “ $\rightarrow$ ” symbol denotes a derivation of a tree in Prolog and the “:-” symbol indicates a pattern match operation. Another difference is that dummy nodes appear in the rules. The reason for having dummy nodes is that, often, only the parts of a tree are referenced in the anatomy tree of the specification. The remaining unreferenced parts of the tree still need to be unified in the course of pattern matching. Dummy nodes are obtained by referring back to the analysis tree of the specification.

#### Rule 5

$$\alpha \mid$$

$\beta_1$	where $\alpha$ is a non-terminal node
$\beta_2$	$\beta_1$ is a non-terminal node
...	
...	
$\beta_n$	

---

$\epsilon :- \gamma_1$	where $\epsilon$ is a non-terminal symbol, where $\epsilon = \text{uppercase}(\alpha)$
$\epsilon :- \gamma_2$	$\gamma_i$ is a non-terminal symbol, where $\gamma_i = \text{uppercase}(\beta_i)$
...	
...	
$\epsilon :- \gamma_n$	

In Rule 5, a union non-terminal node  $\alpha$  in the anatomy tree indicates that  $\alpha$  is one of the alternatives,  $\beta_1, \beta_2, \dots, \beta_n$ . Each translated DCG represents an alternative.  $\epsilon$  and  $\gamma_i$  in uppercase characters are non-terminal symbols.

#### Rule 6

$$\alpha \&$$

$\beta_1$	where $\alpha$ is a non-terminal node
$\beta_2$	$\beta_1$ is a non-terminal node, terminal node, or statement

...  
 ...  
 $\beta_n$

---

$\varepsilon :- \gamma_1 \langle \{\psi_1\} \rangle \gamma_2 \langle \{\psi_2\} \rangle \dots \gamma_n \langle \{\psi_n\} \rangle$   
 where  $\psi_1 \subseteq Y$   
 $\varepsilon$  is a non-terminal symbol, where  $\varepsilon = \text{uppercase}(\alpha)$   
 $\gamma_i$  is a non-terminal symbol, where  $\gamma_i = \text{uppercase}(\beta_i)$   
 if  $\beta_i$  is a non-terminal node;  
 otherwise  $\gamma_i$  is a terminal symbol, where  $\gamma_i = \beta_i$

In Rule 6, a concatenation non-terminal node  $\alpha$  in the anatomy tree indicates that  $\alpha$  is a concatenation of  $\beta_1, \beta_2, \dots, \beta_n$ . The translated DCG represents a concatenation form.

#### Rule 7

$\alpha^*$   
 $\beta_1$                       where  $\alpha$  is a non-terminal node  
 $\beta_2$                        $\beta_i$  is a non-terminal node, terminal node, or statement  
 ...  
 ...  
 $\beta_n$

---

$\varepsilon :- [ ]$   
 $\varepsilon :- \gamma_1 \langle \{\psi_1\} \rangle \gamma_2 \langle \{\psi_2\} \rangle \dots \gamma_n \langle \{\psi_n\} \rangle \varepsilon$   
 where  $\psi_1 \subseteq Y$   
 $\varepsilon$  is a non-terminal symbol, where  $\varepsilon = \text{uppercase}(\alpha)$   
 $\gamma_i$  is a non-terminal symbol, where  $\gamma_i = \text{uppercase}(\beta_i)$   
 if  $\beta_i$  is a non-terminal node;  
 otherwise,  $\gamma_i$  is a terminal symbol, where  $\gamma_i = \beta_i$

In Rule 7, a Kleene closure non-terminal node  $\alpha$  in the anatomy tree indicates that  $\alpha$  is a sequence of zero or more occurrences of  $\beta_1, \beta_2, \dots, \beta_n$ . Two translated DCGs represent a Kleene closure form.

#### Rule 8

$\alpha^+$   
 $\beta_1$                       where  $\alpha$  is a non-terminal node  
 $\beta_2$                        $\beta_i$  is a non-terminal node, terminal node, or statement  
 ...  
 ...  
 $\beta_n$

---

$\varepsilon :- \gamma_1 \langle \{\psi_1\} \rangle \gamma_2 \langle \{\psi_2\} \rangle \dots \gamma_n \langle \{\psi_n\} \rangle$

$$\varepsilon :- \gamma_1 \langle \{\psi_1\} \rangle \gamma_2 \langle \{\psi_2\} \rangle \dots \gamma_n \langle \{\psi_n\} \rangle \varepsilon$$

where  $\psi_1 \subseteq Y$

$\varepsilon$  is a non-terminal symbol, where  $\varepsilon = \text{uppercase}(\alpha)$   
 $\gamma_i$  is a non-terminal symbol, where  $\gamma_i = \text{uppercase}(\beta_i)$   
 if  $\beta_i$  is a non-terminal node;  
 otherwise,  $\gamma_i$  is a terminal symbol, where  $\gamma_i = \beta_i$

In Rule 8, a positive closure non-terminal node  $\alpha$  in the anatomy tree indicates that  $\alpha$  is a sequence of one or more occurrences of  $\beta_1, \beta_2, \dots, \beta_n$ . Two translated DCGs represent a positive closure form.

The application of Rules 5-8 to the anatomy tree of the *bubble\_sort* specification produces the results shown below.

```
(7) SEQUENCE :- UNSORTED
(8) SEQUENCE :- SORTED
(9) UNSORTED :- T_L1 = y :: list2
                T_L2 = x :: rest_of_elements
                T_L = list1 <> T_L1 <> T_L2
                call bubble_sort(T_L)
(10) SORTED :- ASCENDING_SEQUENCE
(11) ASCENDING_SEQUENCE :- output element
                           output ' '
(12) ASCENDING_SEQUENCE :- output element
                           output ' '
                           ASCENDING_SEQUENCE
```

The following six rules generate DCG rules in Prolog from the DCGs produced through analysis. DCG rules in Prolog can be directly executed in a Prolog environment. Each non-terminal node of the DCG is transformed into a form of the following kind:

```
predicate-name(argument-list).
```

For each DCG rule in Prolog, the left-hand side becomes the predicate-name, and the right-hand side becomes the arguments. The condition tests are transformed into procedure calls in Prolog through a set of transformation rules. The transformation rules for the condition tests are encoded as a collection of built-in functions stored in a library. Whenever a condition test is encountered, it is replaced by an appropriate function in the library. A predicate name in a DCG rule in Prolog must be in lowercase characters. The arguments in the argument list are either constants, variables, or trees in functional notations. A constant with a pair of single quotes, such as a string, is enclosed in square brackets to distinguish it from a variable. Variables are used to represent trees or subtrees whose values are not known until unification is performed. Unification is a process of finding values in the input sequence for the variables. It is also possible that several rules may apply at the same time. There is no preference as to which rule should be chosen; thus, the same results will be obtained no matter which rule is applied first.

## Rule 9

$$\alpha \rightarrow \dots \beta_i \dots \quad \begin{array}{l} \text{where } \alpha \text{ is a non-terminal node} \\ \beta_i \text{ is a string enclosed in a pair of single quote} \end{array}$$


---


$$\rho(\rho(\dots, \beta_i, \dots)) \rightarrow \dots \quad \begin{array}{l} \text{where } \rho \text{ is a predicate name, where } \rho = \text{lowercase}(\alpha) \\ [\beta_i], \\ \dots \end{array}$$

In Rule 9, if  $\beta_i$  is a constant string enclosed in a pair of quotes in the analysis tree, then  $\beta_i$  is enclosed in brackets in a DCG rule in Prolog. A string node is different from a terminal node. A string can be considered as a terminal node with a constant value associated with it. However, a terminal node can take any value as long as the value satisfies the node constraints. A string node in the anatomy part of the specification indicates that there is a constant value in the input sequence.

## Rule 10

$$\alpha \rightarrow \dots \beta_i \beta_{i+1} \{ \Phi_{i+1} \} \dots \quad \begin{array}{l} \text{where } \alpha \text{ is a non-terminal node} \\ \beta_i \text{ is a terminal node} \\ \Phi_{i+1} \subseteq C \\ \beta_{i+1} \text{ is a terminal node with a condition test} \end{array}$$


---


$$\rho(\rho(\dots, \gamma_i, \gamma_{i+1}, \dots)) \rightarrow \dots \quad \begin{array}{l} \text{where } \rho \text{ is a predicate name, where } \rho = \text{lowercase}(\alpha) \\ \text{skip1}(\gamma_i, \gamma_{i+1}), \quad \gamma_i \text{ is a variable, where } \gamma_i = \text{uppercase}(\beta_i) \\ [\gamma_{i+1}] \quad \gamma_{i+1} \text{ is a variable, where } \gamma_{i+1} = \text{uppercase}(\beta_{i+1}) \\ \{ \xi \}, \quad \{ \xi = \text{translate}(\Phi_{i+1}) \} \subseteq Q \\ \dots \end{array}$$

In Rule 10,  $\beta_i$  is a terminal node and is one type of list in the analysis tree.  $\gamma_i$  is used to match a list of tokens in the input data until  $\beta_{i+1}$  matches a satisfied token. Variables  $\gamma_i$  and  $\gamma_{i+1}$  are used to obtain the values from the input sequence during unification. Variables in a DCG rule in Prolog are in uppercase characters to distinguish them from constants.

## Rule 11

$$\alpha \rightarrow \dots \beta_i \quad \begin{array}{l} \text{where } \alpha \text{ is a non-terminal node} \\ \beta_i \text{ is a terminal node} \end{array}$$


---


$$\rho(\rho(\dots, \gamma_i)) \rightarrow \dots \quad \begin{array}{l} \text{where } \rho \text{ is a predicate name, where } \rho = \text{lowercase}(\alpha) \\ \text{rest\_of\_set}(\gamma_i) \quad \gamma_i \text{ is a variable, where } \gamma_i = \text{uppercase}(\beta_i) \end{array}$$

In Rule 11, if  $\beta_i$  is the last terminal node in the analysis tree, then the  $\gamma_i$  variable in Prolog is used to match the rest of the tokens in the input data.  $\gamma_i$  is used to unify the rest of the data in the input sequence.

#### Rule 12

$$\alpha \rightarrow \dots \beta_i \{ \Phi_i \} \dots$$

where  $\alpha$  is a non-terminal node  
 $\Phi_i \subseteq C$   
 $\beta_i$  is a terminal node

$$\rho(\rho(\dots, \gamma_i, \dots)) \rightarrow$$

where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$   
 $\gamma_i$  is a variable, where  $\gamma_i = \text{uppercase}(\beta_i)$   
 $\{ \xi \} = \text{translate}(\Phi_i) \subseteq Q$

In Rule 12, if  $\beta_i$  is a terminal node in the analysis tree, then  $\beta_i$  is enclosed in brackets in a DCG rule in Prolog. The condition test  $\Phi_i$  associated with this terminal node  $\beta_i$  is translated into a set of Prolog procedure calls,  $\xi$ .

#### Rule 13

$$\alpha \rightarrow \dots \beta_i \dots$$

where  $\alpha$  is a non-terminal node  
 $\beta_i$  is a non-terminal node

$$\rho(\rho(\dots, \beta_i, \dots)) \rightarrow$$

where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$   
 $\beta_i$  is a variable  
 $\eta_i$  is a predicate name, where  
 $\eta_i = \text{lowercase}(\beta_i)$

In Rule 13, if  $\beta_i$  is a non-terminal node in the analysis tree, then  $\beta_i$  is translated into a predicate,  $\eta_i(\beta_i)$ .  $\eta_i(\beta_i)$  is a predicate, where  $\eta_i$  is a predicate name that must be in lower-case characters, and  $\beta_i$  is an argument of the predicate for the purpose of data unification.

#### Rule 14

$$\alpha \rightarrow [ ]$$

where  $\alpha$  is a non-terminal node

$$\rho(\rho([ ])) \rightarrow [ ]$$

where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$

In Rule 14,  $[ ]$  represents an empty list in Prolog. The Prolog predicate is used to form an empty list.

The application of Rules 9-14 to the DCGs of the analysis tree of the *bubble\_sort* specification produces the results shown below, which can be directly executed in a Prolog environment.

- (1') `sequence(sequence(UNSORTED)) →  
       unsorted(UNSORTED).`
- (2') `sequence(sequence(SORTED)) →  
       sorted(SORTED).`
- (3') `unsorted(unsorted(LIST1, X, LIST2, Y, REST_OF_ELEMENTS)) →  
       skip1(LIST1, X),  
       [X],  
       {integer(X)},  
       skip1(LIST2, Y),  
       [Y],  
       {integer(Y),  
       X > Y},  
       rest_of_set(REST_OF_ELEMENTS).`
- (4') `sorted(sorted(ASCENDING_SEQUENCE)) →  
       ascending_sequence(ASCENDING_SEQUENCE).`
- (5') `ascending_sequence(ascending_sequence(ELEMENT)) →  
       [ELEMENT],  
       {integer(ELEMENT)}.`
- (6') `ascending_sequence(ascending_sequence(ELEMENT,  
       ASCENDING_SEQUENCE)) →  
       [ELEMENT],  
       {integer(ELEMENT)},  
       ascending_sequence(ASCENDING_SEQUENCE).`

The following eight rules derive a Prolog program from the DCGs of anatomy. We use the “:-” symbol to indicate a pattern match operation. Note that dummy non-terminal or terminal nodes appear in the axioms. The reason for having dummy nodes is that, often, only the parts of a tree are referenced in the anatomy tree of the specification. The remaining unreferenced parts of the tree still need to be unified in the course of pattern matching. Dummy nodes are obtained by referring back to the analysis tree of the specification. A translated Prolog statement has the following predicate header form:

$$\rho(\rho(\text{argument-list})) \text{ :-,}$$

where  $\rho$  is a predicate name that is always in lowercase and  $\rho(\text{argument-list})$  is a tree in functional notation. Arguments in the *argument-list* can be either constants, variables, or trees. Constants indicates that a constant value must appear in the input sequence for data match. Variables won't obtain values until data unification is performed. A tree may, in turn, contain constants, variables and subtrees.

## Rule 15

$$\alpha :- \dots \beta_i \dots$$

where  $\alpha$  is a non-terminal node  
 $\beta_i$  is a terminal node  
 $\beta_i \in Y$

---


$$\rho(\rho(\dots, \beta_i, \dots)) :-$$

... where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$

In Rule 15, the dummy node  $y_1 \in Y$  is used to unify the unreferenced parts of a tree.

## Rule 16

$$\alpha :- \dots \beta_i \dots$$

where  $\alpha$  is a non-terminal node  
 $\beta_i$  is a string enclosed in a pair of single quotes

---


$$\rho(\rho(\dots, \beta_i, \dots)) :-$$

... where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$

In Rule 16,  $\beta_i$  is a constant string indicating that a constant string must appear in the input sequence to be unified with  $\beta_i$ .

## Rule 17

$$\alpha :- \dots \beta_i \dots$$

where  $\alpha$  is a non-terminal node  
 $\beta_i$  is a terminal node

---


$$\rho(\rho(\dots, \gamma_i, \dots)) :-$$

... where  $\rho$  is a predicate name where  $\rho = \text{lowercase}(\alpha)$   
 ...  $\gamma_i$  is a variable where  $\gamma_i = \text{uppercase}(\beta_i)$

In Rule 17,  $\beta_i$  is a terminal node in a DCG.  $\gamma_i$  is used to obtain a sublist of tokens in a list with unification.

## Rule 18

$$\alpha :- \dots \beta_i$$

where  $\alpha$  is a non-terminal node  
 $\beta_i$  is a terminal node

---


$$\rho(\rho(\dots, \gamma_i)) :-$$

... where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$   
 ...  $\gamma_i$  is a variable, where  $\gamma_i = \text{uppercase}(\beta_i)$

In Rule 18,  $\beta_i$  is a terminal node in a DCG.  $\gamma_i$  is used to obtain the rest of tokens in a list through unification.

## Rule 19

$$\alpha :- \dots \beta_i \dots$$

where  $\alpha$  is a non-terminal node  
 $\beta_i \in \{\text{statements}\}$

---


$$\rho(\rho(\dots)) :-$$

$$\dots$$

$$\xi$$

$$\dots$$

where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$   
 $\{\xi = \text{translate}(\beta_i)\} \subseteq Q$

In Rule 19, a statement  $\beta_i$  in a DCG is translated into a set of Prolog statements.

## Rule 20

$$\alpha :- \dots \beta_i \dots$$

where  $\alpha$  is a non-terminal node  
 $\beta_i$  is a non-terminal node

---


$$\rho(\rho(\dots, \beta_i, \dots)) :-$$

$$\dots$$

$$\eta_i(\beta_i),$$

$$\dots$$

where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$   
 $\beta_i$  is a variable  
 $\eta_i$  is a predicate name, where  
 $\eta_i = \text{lowercase}(\beta_i)$

In Rule 20,  $\beta_i$  is a non-terminal node, which represents a Prolog list with a node  $\eta_i$ . The Prolog predicate  $\eta_i(\beta_i)$  is used to perform pattern matching.

## Rule 21

$$\alpha :- \dots \beta_i\# \dots$$

where  $\alpha$  is a non-terminal node  
 $\beta_i$  is a terminal node

---


$$\rho(\rho(\dots, \gamma_i, \dots)) :-$$

$$\dots$$

$$\xi$$

$$\dots$$

where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$   
 $\{\xi = \text{translate}(\beta_i\#)\} \subseteq Q$   
 $\gamma_i$  is a variable, where  $\gamma_i = \text{uppercase}(\beta_i)$

In Rule 21, the symbol # is used to indicate the number of occurrences of  $\beta_i$ . If the parent of  $\beta_i$  is a non-terminal node followed by a Kleene closure notation or a positive closure notation, then  $\beta_i\#$  computes the occurrences of  $\beta_i$ . Note that the # symbol defined in Table 2 is an operator used in the statements in the anatomy part of a TUG specification. The # symbol is not a regular expression notation; thus, it is not defined in Rules 5 to 8.

## Rule 22

$$\alpha :- [ ]$$

where  $\alpha$  is a non-terminal node

---


$$\rho(\rho([ ])).$$

where  $\rho$  is a predicate name, where  $\rho = \text{lowercase}(\alpha)$

An empty list is discarded and ignored in the Prolog program.

The application of Rules 15-22 to the DCGs of the anatomy tree of the *bubble\_sort* specification produces the results shown below.

```
(7') sequence(sequence(UNSORTED)) :-
    unsorted(UNSORTED).
(8') sequence(sequence(SORTED)) :-
    sorted(SORTED).
(9') unsorted(unsorted(LIST1, X, LIST2, Y, REST_OF_ELEMENTS)) :-
    T_L1 = [Y | LIST2],
    T_L2 = [X | REST_OF_ELEMENTS],
    concatenate(LIST1, T_L1, TT),
    concatenate(TT, T_L2, T_L),
    bubble_sort(T_L).
(10') sorted(sorted(ASCENDING_SEQUENCE)) :-
    ascending_sequence(ASCENDING_SEQUENCE).
(11') ascending_sequence(ascending_sequence(ELEMENT)) :-
    write(ELEMENT),
    write(' ').
(12') ascending_sequence(ascending_sequence(ELEMENT, ASCENDING_SEQUENCE)) :-
    write(ELEMENT),
    write(' '),
    ascending_sequence(ASCENDING_SEQUENCE).
```

A prototype derived from a specification via the above set of transformation rules is then exercised under the Prolog environment to demonstrate the system behavior. Feedback is obtained to guide the reformulation of the specification in cases where the specification misrepresents the user requirements. How the approach supports the evolutionary user requirements is illustrated with an example in section 7.

## 6. SOFTWARE REUSE

Reuse is believed to be one key to improving software development productivity and quality [2]. TUG provides an overloading facility to achieve a compromise between strong and weak typing. The language allows for specifying a value of a terminal node limited to only one type, and also lets the type of the value of a terminal node vary in a disciplined manner. For example, it should be reasonable to write a specification to sort a sequence of elements of any type as long as that type has an ordering < on it. TUG supports not only overloading on terminal nodes but also overloading on operators. A polymorphic operator is provided to denote an operation whose meaning is determined by the corresponding operand types. In addition to the built-in primitive operators, such as *greater\_than* and *equal\_to*, the features of overloading are also applied to user-defined operators. Overloading in a function causes a developer to abstract away from the particular type of argument to the function. The developer does not have to specify the

specifications more than necessary. Without overloading, the developer must give one specification for each different type. Overloading is a purely syntactic way of using the same operator for different semantic operands. Overloading relieves a developer of having to invent various names for different operations. Instead, this burden is placed on the language processor.

TUG permits the creation of a template to represent similar modules. For example, different sorting modules are required to sort a sequence of integers, a sequence of characters, or a sequence of floating point numbers. The specification used for sorting could be the same for each module; the only differences would be in the types of values involved. Rather than requiring the specifications of virtually identical modules, TUG permits the creation of a template such that only the essential properties are captured. The template leaves some parts, such as the types of values, unspecified. Since choices of types or other properties are defined, a template called a scheme in the language can be reused in those applications that call for the same modules, but with different types of values. To use a scheme, a process called instantiation must be applied by giving actual parameters corresponding to scheme formal parameters. A scheme maximizes specification reuse by storing specifications in as general a form as possible.

In some cases, such as when a sequence of values of any type is reversed, instantiation is cumbersome and inflexible for writing a specification. The *any* root type, which is a supertype of all types, makes it possible to write a generic module taking a variety of data types as arguments, rather than instantiating a module with different types of values. A developer does not need to construct a group of similar specifications that only differ in terms of types.

Often there is a specification for reuse that is not in exactly the right form. In this case, a promising approach is to apply some modifications. TUG does so by modifying schemes before instantiation or extending modules before they are used, so that they can fit a wider range of applications. The language provides a renaming operator to replace an old name with a new name. In TUG, extensions to a module are used to expand the non-terminal nodes. The capability of extending non-terminal nodes makes it easier to write a specification for some applications, such as natural language processing, because of the recursive definitions of non-terminal nodes.

As an example demonstrating overloading on operators, let us consider the *greater\_than* operator. The integer *greater\_than*, float *greater\_than*, and character *greater\_than* are defined below:

```
greater_than: Character × Character → Boolean
greater_than: Float × Float → Boolean
greater_than: Integer × Integer → Boolean.
```

The above *greater\_than* operator is polymorphic and is viewed as a single function with different types of values. The function in TUG is specified as follows:

```
MODULE greater_than(in: INPUT_DATA) return bool
  ANALYSIS
    INPUT_DATA |
```

```

CHARACTER_TYPE&
  element1
    {character(element1)}
  element2
    {character(element2),
     follow(element1, element2)}
FLOAT_TYPE&
  element1
    {float(element1)}
  element2
    {float(element2),
     less_than(element1, element2)}
INTEGER_TYPE&
  element1
    {integer(element1)}
  element2
    {integer(element2),
     less_than(element1, element2)}
ERROR_TYPE&
  error
END OF ANALYSIS;
ANATOMY
  input_data |
  character_type&
    return true
  float_type&
    return true
  integer_type&
    return true
  error_type&
    return false
END OF ANATOMY;
END OF MODULE greater_than.

```

The *greater\_than* function overloads the built-in *greater\_than* operator. The function takes three different types of values and returns a *Boolean* value. The operator *less\_than* is a built-in operator for ordering. The polymorphic operator *greater\_than* is resolvable since the operator takes only an *integer* type, or a *character* type, or a *float* type without any ambiguity.

With the polymorphic operator *greater\_than*, the following example illustrates overloading on terminals. Note that the type of a terminal in TUG can vary only if it is used in disciplined manner. A disciplined manner means that a polymorphic terminal node should be properly used with its polymorphic operators. With the *greater\_than* operator defined correctly, for instance, a bubble sort can be easily specified with polymorphic terminal nodes.

The specification of bubble sort in TUG given below shows the use of the polymorphic operator *greater\_than*. The terminal nodes *x* and *y* are polymorphic since they can take any type of value only if that type of value can be accepted by the polymorphic operator *greater\_than*. Since the polymorphic operator *greater\_than* accepts only integers, characters, and floats, this bubble sort, of course, only takes the same types of values that the operator *greater\_than* does:

```

MODULE bubble_sort(in: SEQUENCE)
  ANALYSIS
    SEQUENCE |
      UNSORTED&
        list1
        x
        list2
        y
          {greater_than(x, y)}
        rest_of_elements
      SORTED&
        ASCENDING_SEQUENCE+
          element
  END OF ANALYSIS;
  ANATOMY
    sequence |
      unsorted&
        T_L1 = y :: list2
        T_L2 = x :: rest_of_elements
        T_L = list1 <> T_L1 <> T_L2
        call bubble_sort(T_L)
      sorted&
        ascending_sequence+
          output element
          output ‘ ‘
  END OF ANATOMY;
END OF MODULE bubble_sort.

```

The specification of bubble sort in TUG can also be specified in a scheme. By deferring choices of types, the scheme for bubble sort can be instantiated with the types *integer*, *float*, and *character* to produce a concrete specification for sorting a sequence of integers, characters, or floats. A scheme for bubble sort is defined as follows:

```

SCHEME bubble_sort_s(TYPE)
  ANALYSIS
    SEQUENCE |
      UNSORTED&
        list1

```

```

        x
          {TYPE(x)}
        list2
        y
          {TYPE(y),
           greater_than(x, y)}
        rest_of_elements
    SORTED&
        ASCENDING_SEQUENCE+
        element
          {TYPE(element)}
    END OF ANALYSIS;
    ANATOMY
        sequence |
            unsorted&
                T_L1 = y :: list2
                T_L2 = x :: rest_of_elements
                T_L = list1 <> T_L1 <> T_L2
                call bubble_sort_s(T_L)
            sorted&
                ascending_sequence+
                output element
                output ‘ ‘
    END OF ANATOMY;
END OF SCHEME bubble_sort_s.

```

With the scheme for bubble sort, we can sort a sequence of integers, a sequence of floats, or a sequence of characters by instantiating the specification template. The following specification illustrates how the scheme `bubble_sort` is instantiated to sort a sequence of integers:

```

MODULE bubble_sort(in: SEQUENCE)
    INSTANTIATION
        Instantiate bubble_sort with bubble_sort_s(integer)
    END OF INSTANTIATION;
END OF MODULE bubble_sort.

```

The application of TUG to specify reusable systems can be found in [28, 29].

## 7. ANALYSIS OF TUG SPECIFICATIONS

Errors introduced early in the software development process and discovered later are difficult and expensive to correct. Errors occurring in the requirements definition and specifications phase and found in the completed system may require extensive reworking of the entire system. Any software development process that relies on specifications must

provide a means for detecting and correcting errors in specifications. In addition, writing and analyzing a complete specification at the first attempt is often impossible. The TUG specification language supports the development and analysis of specifications in an incremental manner. The language allows developers to write a specification bit by bit and to test/analyze the specification bit by bit until the entire specification meets the user's requirements. This section explains how user requirements and specifications in TUG can be elicited and formalized incrementally in a top-down manner. We will use the same bubble sort problem, followed by analysis of the specification corresponding to the problem.

The same bubble sort problem is used here to demonstrate how a specification corresponding to the problem can be written incrementally using CRS. The first attempt tries to capture the sequence of integers and display the sequence to the screen. A non-terminal *SEQUENCE* node contains a sequence of integers. A first attempt at writing the specification is as follows:

```

MODULE bubble_sort(in: SEQUENCE)
  ANALYSIS
    SEQUENCE+
      element
        {integer(element)}
  END OF ANALYSIS;
  ANATOMY
    sequence+
      output element
      output ' '
  END OF ANATOMY;
END OF MODULE bubble_sort.

```

The following refinement in CRS involves the *SEQUENCE* node. The input sequence is refined into an unsorted sequence, *UNSORTED*, and a sorted sequence, *SORTED*. The input sequence is checked with regard to the order of a pair of integers,  $x$  and  $y$ . If the integers,  $x$  and  $y$ , being compared are out of order, the sequence is unsorted:

```

replace SEQUENCE+ with
  SEQUENCE |
    UNSORTED&
      list1
      x
      {integer(x)}
      list2
      y
      {integer(y),
        greater_than(x, y)}
      rest_of_elements
    SORTED&

```

```

ASCENDING_SEQUENCE+
  element
  {integer(element)}

```

The following refinement involves the *sequence* node in the anatomy part of the specification. An unsorted sequence, *unsorted*, is rearranged by swapping  $x$  and  $y$  in order to make the sequence a sorted sequence. To swap  $x$  and  $y$ ,  $y$  is appended (::) to the head of *list2* to make a new temporary variable,  $T\_L1$ . Following that,  $x$  is appended to the head of *rest\_of\_elements* to make another new temporary list,  $T\_L2$ . Finally, string concatenations (<>) are performed on *list1*,  $T\_L1$ , and  $T\_L2$  to make another new list,  $T\_L$ . In  $T\_L$ ,  $x$  and  $y$  have already been swapped. If the sequence has already been sorted, individual element followed by a blank is displayed to the screen:

```

replace sequence+ with
  sequence |
  unsorted&
    T_L1 = y :: list2
    T_L2 = x :: rest_of_elements
    T_L = list1 <> T_L1 <> T_L2
    call bubble_sort(T_L)
  sorted&
    ascending_sequence+
      output element
      output ' '

```

After successive refinements to the specification in the first attempt, the final specification is completed, which is exactly same as the specification shown in section 4. Since the TUG specification language is executable, a developer can test or analyze a TUG specification at any moment during the refinement. Therefore, a TUG specification can be written bit by bit and tested/analyzed bit by bit until the specification is complete. In the following section, we will demonstrate the proofs using the same bubble sort specification.

The TUG specification language facilitates analysis of a specification to detect errors. Static analysis allows a TUG specification to be analyzed to uncover inconsistencies, redundancies, and ambiguities without executing the specification due to the formality of the language. Dynamic analysis allows a specification to be analyzed by executing the specification. The method with TUG provides two ways of executing a specification to detect errors: the operational-specification approach and the rapid-prototyping-via-software-transformations approach. However, the execution of a specification can only reveal the existence of errors, not show the absence of errors [30]. Due to the limitations of dynamic analysis, we provide a way of analyzing a TUG specification against the user requirements through theorem proving. Failure to prove a specification that satisfies the user requirements can indicate errors in the specification. In this section, we focus on checking a specification against the user requirements.

A proof technique is presented here to analyze a TUG specification against the user claims. A user claim shows a user's concern about the specification. Therefore, there is

no formality in a user claim. Specifiers must turn a user claim into a Horn clause before resolution refutation can proceed. When the resolution process ends up with an empty clause, the proof succeeds. The steps in proving a TUG specification against the user claims are as follows:

- Prove the analysis tree
  - Transform the analysis tree into DCGs. The rules for transforming an analysis tree of a specification into DCGs have already been given in section 5.
  - Transform DCGs into Prolog-like forms. The rules for transforming a DCG into a Prolog-like form can be found in [26].
  - Transform Prolog-like forms into Horn clauses. Each Prolog form can be interpreted as a Horn clause. A Horn clause is a clause with at most one positive literal. A clause is a set of literals representing their disjunctions. A literal is an atomic sentence or the negation of an atomic sentence. An extension of the procedure for translating axioms into clause form is explained below [31].
    - o Eliminate all occurrences of the  $\leftarrow$ ,  $\rightarrow$ , and  $\leftrightarrow$  operators by substituting equivalent sentences involving only the  $\neg$ ,  $\wedge$ , and  $\vee$  operators:
      - $\phi \rightarrow \psi$  is replaced by  $\neg\phi \vee \psi$ .
      - $\phi \leftarrow \psi$  is replaced by  $\phi \vee \neg\psi$ .
      - $\phi \leftrightarrow \psi$  is replaced by  $(\neg\phi \vee \psi) \wedge (\phi \vee \neg\psi)$ .
    - o Distribute negations over other logical operators until each such operator applies to a single atomic sentence. The following replacement rules do the job:
      - $\neg\neg\phi$  is replaced by  $\phi$ .
      - $\neg(\phi \wedge \psi)$  is replaced by  $\neg\phi \vee \neg\psi$ .
      - $\neg(\phi \vee \psi)$  is replaced by  $\neg\phi \wedge \neg\psi$ .
      - $\neg\forall\gamma\phi$  is replaced by  $\exists\gamma\neg\phi$ .
      - $\neg\exists\gamma\phi$  is replaced by  $\forall\gamma\neg\phi$ .
    - o Eliminate existential quantifiers. If an existential quantifier does not occur within the scope of a universal quantifier, we simply drop the quantifier and replace all occurrences of the quantified variable with a new constant, i.e., one that does not occur anywhere else in our database. If an existential quantifier is within the scope of any universal quantifier, we drop the existential quantifier and replace the associated variable with a term formed from a new function symbol applied to the variables associated with the enclosing universal quantifiers. Any function defined in this way is called a Skolem function.
    - o Eliminate universal quantifiers.
    - o Move the disjunctions down to the literals. This task can be accomplished through repeated use of the following rule:
      - $\phi \vee (\psi \wedge \chi)$  is replaced by  $(\phi \vee \psi) \wedge (\phi \vee \chi)$ .
    - o Eliminate the conjunctions.
    - o Rename all the variables, when necessary, so that no two variables are the same. This process is called standardizing the variables.

The following example transforms the Prolog-like forms of *bubble\_sort* presented in section 5 (1'-6') into a set of Horn clauses using the above rules:

- (a)  $\text{sequence}(\text{sequence}(\text{UNSORTED}), \_1, \_2) \vee$   
 $\neg \text{unsorted}(\text{UNSORTED}, \_1, \_2)$
- (b)  $\text{sequence}(\text{sequence}(\text{SORTED}), \_3, \_4) \vee$   
 $\neg \text{sorted}(\text{SORTED}, \_3, \_4)$
- (c)  $\text{unsorted}(\text{unsorted}(\text{LIST1}, X, \text{LIST2}, Y, \text{REST\_OF\_ELEMENTS}), \_5, \_6)$   
 $\{ \text{skip1}(\text{LIST1}, X, \_5, \_7),$   
 $\text{c}(\_7, X, \_8),$   
 $\text{integer}(X),$   
 $\text{skip1}(\text{LIST2}, Y, \_8, \_9),$   
 $\text{c}(\_9, Y, \_10),$   
 $\text{integer}(Y),$   
 $X > Y,$   
 $\text{rest\_of\_set}(\text{REST\_OF\_ELEMENTS}, \_10, \_6) \}$
- (d)  $\text{sorted}(\text{sorted}(\text{ASCENDING\_SEQUENCE}), \_11, \_12) \vee$   
 $\neg \text{ascending\_sequence}(\text{ASCENDING\_SEQUENCE}, \_11, \_12)$
- (e)  $\text{ascending\_sequence}(\text{ascending\_sequence}(\text{ELEMENT}), \_13, \_14)$   
 $\{ \text{c}(\_13, \text{ELEMENT}, \_14),$   
 $\text{integer}(\text{ELEMENT}) \}$
- (f)  $\text{ascending\_sequence}(\text{ascending\_sequence}(\text{ELEMENT},$   
 $\text{ASCENDING\_SEQUENCE}), \_15, \_16)$   
 $\{ \text{c}(\_15, \text{ELEMENT}, \_17),$   
 $\text{integer}(\text{ELEMENT}) \} \vee$   
 $\neg \text{ascending\_sequence}(\text{ASCENDING\_SEQUENCE}, \_17, \_16)$

- Prove that a valid statement is a logical consequence of Horn clauses by means of resolution refutation proofs (e.g., an empty clause is generated).
- An empty clause is generated.

Suppose a user claims that there are seven integers to be sorted. The input data is 1, 2, 3, 5, 6, 60, and 45. The following shows the user claim is a logical consequence of the Horn clauses generated in the previous proof step:

- (g')  $\neg \text{sequence}(\text{TREE}, [1, 2, 3, 5, 6, 60, 45], [])$  (user claim)
- (a)  $\text{sequence}(\text{sequence}(\text{UNSORTED}), \_1, \_2) \vee$   
 $\neg \text{unsorted}(\text{UNSORTED}, \_1, \_2)$
- $\Rightarrow \neg \text{unsorted}(\text{UNSORTED}, [1, 2, 3, 5, 6, 60, 45], [])$   
 $\{ \_1 / [1, 2, 3, 5, 6, 60, 45],$   
 $\_2 / [],$   
 $\text{TREE} / (\text{sequence}(\text{UNSORTED}))$  (h')
- (h')  $\neg \text{unsorted}(\text{UNSORTED}, [1, 2, 3, 5, 6, 60, 45], [])$
- (c)  $\text{unsorted}(\text{unsorted}(\text{LIST1}, X, \text{LIST2}, Y, \text{REST\_OF\_ELEMENTS}), \_5, \_6)$   
 $\{ \text{skip1}(\text{LIST1}, X, \_5, \_7),$   
 $\text{c}(\_7, X, \_8),$   
 $\text{integer}(X),$   
 $\text{skip1}(\text{LIST2}, Y, \_8, \_9),$   
 $\text{c}(\_9, Y, \_10),$

```

integer(Y),
X > Y,
rest_of_set(REST_OF_ELEMENTS, _10, _6)
⇒
{ _5 / [1, 2, 3, 5, 6, 60, 45],
  _6 / [],
  _7 / [60, 45],
  _8 / [45],
  _9 / [45],
  _10 / [],
  UNSORTED / unsorted(LIST1, X, LIST2, Y, REST_OF_ELEMENTS),
  LIST1 / [1, 2, 3, 5, 6],
  X / 60,
  LIST2 / [],
  Y / 45,
  REST_OF_ELEMENTS / []}

```

...

[The input data are not yet sorted; therefore, the input data must go through another pass to be transformed into the form shown below.]

...

```

⇒ ¬sequence(sequence(sorted(ascending_sequence(1,
ascending_sequence(2, (ascending_sequence(3,
ascending_sequence(5, (ascending_sequence(6,
ascending_sequence(45, (ascending_sequence(60))))))))))))) (m')

```

- Prove the anatomy tree

- Transform the anatomy tree into DCGs. The rules for transforming an anatomy tree of a specification into DCGs have been given in section 5.
- Transform DCGs into Prolog-like forms. The rules for transforming a DCG into a Prolog-like form can be found in [26].
- Transform Prolog-like forms into Horn clauses.

The following example transforms the Prolog-like forms of *bubble\_sort* presented in section 5 (7'-12') into a set of Horn clauses using the above rules:

- (g)  $\text{sequence}(\text{sequence}(\text{UNSORTED})) \vee$   
 $\neg\text{unsorted}(\text{UNSORTED})$
- (h)  $\text{sequence}(\text{sequence}(\text{SORTED})) \vee$   
 $\neg\text{sorted}(\text{SORTED})$
- (i)  $\text{unsorted}(\text{unsorted}(\text{LIST1}, X, \text{LIST2}, Y, \text{REST\_OF\_ELEMENTS}))$   
 $\{T\_L1 = [Y \mid \text{LIST2}],$   
 $T\_L2 = [X \mid \text{REST\_OF\_ELEMENTS}],$   
 $\text{concatenate}(\text{LIST1}, T\_L1, TT),$   
 $\text{concatenate}(TT, T\_L2, T\_L),$   
 $\text{integer\_bubble\_sort}(T\_L)\}$
- (j)  $\text{sorted}(\text{sorted}(\text{ASCENDING\_SEQUENCE})) \vee$

```

    ¬ascending_sequence(ASCENDING_SEQUENCE)
(k) ascending_sequence(ascending_sequence(ELEMENT))
    { write(ELEMENT),
      write(' ') }
(l) ascending_sequence(ascending_sequence(ELEMENT,
    ASCENDING_SEQUENCE))
    { write(ELEMENT),
      write(' ') } ∨
¬ascending_sequence(ASCENDING_SEQUENCE)

```

- Prove that the output is a logical consequence of the Horn clauses by means of resolution refutation proofs (e.g., an empty clause is generated).

An extension of resolution refutation proofs [31] is used to prove the analysis tree and the anatomy tree through the following steps:

- Transform the specification in clause form into a set of clauses.
- Assume that the negation of the user claim is true.
- Add the negation of the user claim, in clause form, to the set of clauses.
- Repeat
  - Resolve two resolvable clauses together, producing the resolvent that logically follows from them.
  - If the resolvent is the empty clause, then a contradiction has been found with the negated user claim.
  - If the resolvent is not empty, add it to the set of clauses available to the procedure until either a contradiction is found or no progress can be made.
- Conclude that the assumed negation of the user claim cannot be true if the procedure leads to a contradiction. Otherwise, conclude that the assumed negation of the user claim cannot be false if no progress can be made.
- Conclude that the user claim must be true if the assumed negation of the user claim cannot be true. Otherwise, conclude that the user claim must be false if the assumed negation of the user claim cannot be false.

The following shows that the specification satisfies the user claim:

```

(m') ¬sequence(sequence(sorted
    (ascending_sequence(1, ascending_sequence(2,
    ascending_sequence(3, ascending_sequence(5,
    ascending_sequence(6, ascending_sequence(45,
    ascending_sequence(60)))))))))) ∨
(h) sequence(sequence(SORTED)) ∨
    ¬sorted(SORTED)
⇒ ¬sorted(sorted
    (ascending_sequence(1, ascending_sequence(2,
    ascending_sequence(3, ascending_sequence(5,
    ascending_sequence(6, ascending_sequence(45,

```

(ascending\_sequence(60))))))))) (n')  
 {SORTED / sorted(ascending\_sequence(1,  
 ascending\_sequence(2, (ascending\_sequence(3,  
 ascending\_sequence(5, (ascending\_sequence(6,  
 ascending\_sequence(45, (ascending\_sequence(60)))))))))})  
 (n') ¬sorted(sorted  
 (ascending\_sequence(1, ascending\_sequence(2,  
 (ascending\_sequence(3, ascending\_sequence(5,  
 (ascending\_sequence(6, ascending\_sequence(45,  
 (ascending\_sequence(60))))))))) ∨  
 (j) sorted(sorted(ASCENDING\_SEQUENCE)) ∨  
 ¬ascending\_sequence(ASCENDING\_SEQUENCE)  
 ⇒ ¬ascending\_sequence  
 (ascending\_sequence(1, ascending\_sequence(2,  
 (ascending\_sequence(3, ascending\_sequence(5,  
 (ascending\_sequence(6, ascending\_sequence(45,  
 (ascending\_sequence(60))))))))) (o')  
 {ASCENDING\_SEQUENCE / ascending\_sequence(1,  
 ascending\_sequence(2, (ascending\_sequence(3,  
 ascending\_sequence(5, (ascending\_sequence(6,  
 ascending\_sequence(45, (ascending\_sequence(60)))))))))}  
 (o') ¬ascending\_sequence(ascending\_sequence(1, ascending\_sequence(2,  
 (ascending\_sequence(3, ascending\_sequence(5,  
 (ascending\_sequence(6, ascending\_sequence(45,  
 (ascending\_sequence(60))))))))) ∨  
 (l) ascending\_sequence(ascending\_sequence(ELEMENT,  
 ASCENDING\_SEQUENCE))  
 {write(ELEMENT),  
 write(' ')} ∨  
 ¬ascending\_sequence(ASCENDING\_SEQUENCE)  
 ⇒ 1  
 ¬ascending\_sequence(ascending\_sequence(2,  
 (ascending\_sequence(3, ascending\_sequence(5,  
 (ascending\_sequence(6, ascending\_sequence(45,  
 (ascending\_sequence(60))))))))) (p')  
 {ELEMENT / 1,  
 ASCENDING\_SEQUENCE / ascending\_sequence(2,  
 (ascending\_sequence(3, ascending\_sequence(5,  
 (ascending\_sequence(6, ascending\_sequence(45,  
 (ascending\_sequence(60)))))))))}  
 ...  
 ...  
 (u') ¬ascending\_sequence(ascending\_sequence(60)) ∨  
 ascending\_sequence(ascending\_sequence(ELEMENT))  
 {write(ELEMENT),  
 write(' ')}

⇒ 60  
 {ELEMENT / 60}

## 8. STRUCTURED DESIGN AND PROGRAMMING

The early works of Mills and others [33, 34] showed that any program could be formulated using three programming language constructs: sequence, selection, and iteration. Structured design methods using these constructs are sufficient to formulate application designs of any level of complexity. In Fig. 3, a set of design blocks corresponding to these three language constructs is presented to help designers build a structure diagram. Three notations are used to denote the sequence, selection, and iteration constructs. A box with an ampersand symbol (&) in the upper right corner denotes a sequence. A box with a vertical bar (|) in the upper right corner denotes a selection made from options. A box with a pound sign (#) in the upper right corner denotes an iteration.

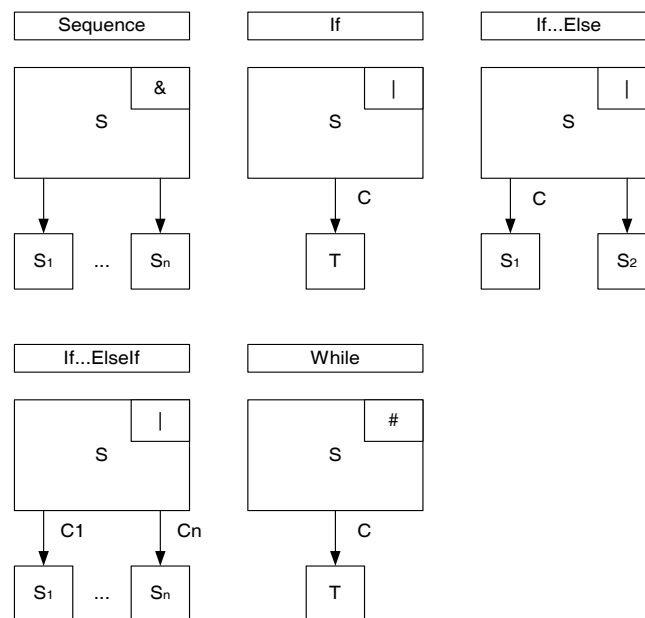


Fig. 3. Basic design blocks.

A TUG specification can be mapped into a set of structure diagrams in a straightforward manner. The mapping process operates in two phases, each of which transforms the input of one representation into another. First, the TUG specification is mapped into a structured design, in which each structure diagram corresponds to a specification module. Second, the structured design is then mapped into a set of structured programs, in which each program module corresponds to a structured diagram. The mapping is performed from a specification, design to coding, according to the structuring notation in the TUG specification.

A structure diagram or structured program is composed of a set of blocks. A block may denote a structuring notation, a single statement, a set of statements, or a complete program. The following mapping guidelines are organized according to the regular notations that guide developers in developing structure diagrams and programs.

- Guideline 1:** A sequence pattern in which a node is concatenated with a set of terminals and non-terminals without comparison operators in the conditional test is mapped to a Sequence design block. The Sequence design block is then mapped to a sequence of language statements.
- Guideline 2:** A sequence pattern in which a node is concatenated with a set of terminals and non-terminals with comparison operators in the conditional test is mapped to an If design block. The If design block is then mapped to an If language statement.
- Guideline 3:** A sequence pattern contains a node concatenated with a set of terminals and non-terminals with comparison operators in the conditional test. In addition, recursion is involved in the operations on the node. The pattern may be mapped to a While design block. The While design block is then mapped to a While language statement.
- Guideline 4:** A union pattern in which a node that has only one alternative is mapped to an If design block. The If design block is then mapped to an If language statement.
- Guideline 5:** A union pattern in which a node that has two alternatives is mapped to an If ... Else or If ... ElseIf design block. This design block is then mapped to an If ... Else language statement.
- Guideline 6:** A union pattern in which a node that has more than two alternatives is mapped to an If ... ElseIf design block and then mapped to an If ... ElseIf language statement.
- Guideline 7:** A Kleene closure pattern in which a node that has zero or more occurrences of terminals or terminals with conditional tests is mapped to a While design block and then a While language statement.
- Guideline 8:** A positive closure pattern in which a node that has one or more occurrences of terminals or terminals with conditional tests is mapped to a Sequence design block followed by a While design block. The Sequence design block is mapped to a set of language statements followed by a While language statement.

A structure diagram and a structure program according to the bubble sort specification presented in section 4 are constructed according to the above guidelines.

The program was implemented in Visual C++ 6.0 under Windows 2000. Guideline 5 is applied to the *SEQUENCE* union pattern. The pattern having two alternatives is mapped to the *If ... Else* block. Guideline 2 suggests that *UNSORTED* and *SORTED* are mapped to an *If* block. However, Guideline 2 is subsumed by the properties of arrays and loops in the Visual C++ language constructs; thus, the sequence pattern is mapped to the while block.

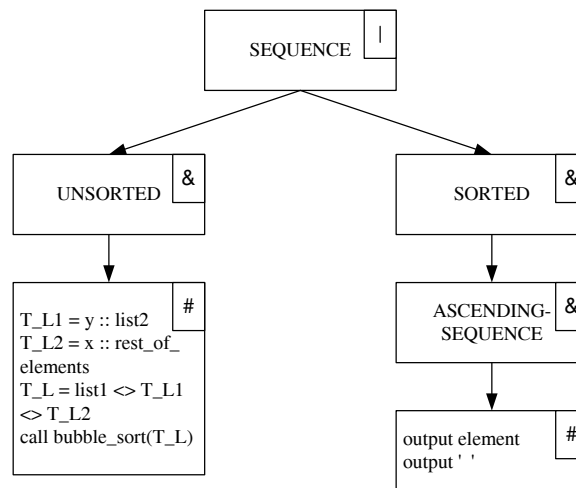


Fig. 4. The bubble sort structure diagram.

```

#include <iostream>
using namespace std;

void bubble_sort(int size, int sequence[]);

void bubble_sort(int size, int sequence[]) {
    bool sorted = true;
    int index;
    for (index = 0; index < size-1; index++) {
        if (sequence[index] > sequence[index + 1]) {
            sorted = false; break;
        }; //if
    }; //for

    if (!sorted) {
        int temp = sequence[index];
        sequence[index] = sequence[index+1];
        sequence[index + 1] = temp;
        bubble_sort(size, sequence);
    } else {
        for (int i = 0; i < size; i++)
            cout << sequence[i] << ' ';
    }; //if ... else
}

int main() {
    const int arraySize = 7;
    int seq[arraySize] = {10, 4, 18, 23, 34, 4, 56};
    bubble_sort(arraySize, seq);
    return 0;
}

```

Fig. 5. A structure program for the bubble sort problem.

## 9. SUMMARY, CONTRIBUTIONS, APPLICABILITY, AND FUTURE RESEARCH

The development of a flexible software development process with a formal specification language, called TUG, has been described in this paper. The formal method supports a mix of software development activities, such as conventional software development, operational specification, rapid prototyping via software transformations, software reuse, and analysis of specifications, such as testing and proofs. The TUG specification language serves as a common media for communications in this process.

The TUG specification language is a formal specification language based on the underlying theory of definite clause grammar. A specification expressed in the language allows it to be exercised to generate the functional behavior of the system. The executable specification can be considered as a prototype for exploring user requirements. The operational specification process via direct execution of a specification iterates until the specification meets the user requirements. Also, developing an executable specification has benefits regarding personnel. Developers generally prefer to work on implementation rather than specification since an implementation can be executed immediately. This language gives them an opportunity to execute a specification, with which developers can detect specification errors using test data.

Rapid prototyping via software transformations helps developers build prototypes automatically. A prototype is automatically derived from specifications. The formal method with TUG supports the rapid prototyping via software transformations process in which a prototype is built quickly and cheaply. Automation of the application of software transformations reduces the amount of labor involved in developing prototypes manually. Rapid prototyping via software transformations also provides support for program modifications. The process bypasses the difficulty of having to modify code that has been poorly structured since changes are made to the specifications. The TUG specification supports prototype evolution by avoiding complete retransformation of the prototype whenever a change is made in the specification. To avoid complete retransformation, a CRS is written and used to update the prototype only in response to minor changes to the specification involving nodes to be modified, extended, relaxed or refined. If a major change is needed, the specification may need to be rewritten and a new prototype may be derived from the beginning. A major change may require the structure of the specification to be modified.

Developers are encouraged to locate and specify potential modules or systems for reuse using TUG in the front-end of the software development process. Sharing reduces the number of errors that occur when one specification is copied to other contexts. Since the language is executable, specifications can be written, tested, and stored specifically for the purpose of reuse. In addition, the specifications in the language are formal, automated tools that can be used for selection, modification, and integration of reusable specifications.

Our approach with TUG supports the detection of errors either by directly executing a specification in the language with test data or by proving a specification against user claims. Executing a specification with test data produces results only for a given test of input data satisfying the specification's input. However, proofs are of little help in some areas, such as user interface suitability [35, 36]. The formal method with TUG empha-

sizes testing and proofs. A combination of testing and proofs enhances software quality and increases software reliability.

The formal method with TUG does not work with object-oriented design very well. It was mainly designed to help its users develop systems using structured design [37]. A formal specification in TUG is written and systematically mapped into a structured design and then into a structured program. The mapping process is performed based on the patterns of union, concatenation, Kleene closure, and positive closure notations in the specification using a set of mapping rules. The structuring notations build a linkage between the specification, design, code, and proofs. Whenever there is a change of user requirements, the impact and changes are located and traced in the specification, design and code through structuring notations. The method for structured programming not only provides a means of detecting errors, but also provides guidance in constructing and proving programs based on the specification's structuring notation. Developers should feel more comfortable developing design and software if they are provided with guidance. In the design and coding phases, the design and software may be proved to be correct by applying a set of proof rules. The structuring notation also guides the application of these rules. Providing guidance for the developer in constructing software makes this method with TUG singularly different from existing formal methods.

The TUG specification has weaknesses. The language based on DCGs provides a logic-based approach to formally specifying and verifying the behavior of sequential software systems. Unlike other existing logic-based approaches, such as those in [23-25], TUG has no notations for concurrency, distribution, parallelism, timing, or performance. This domain-specific approach sacrifices suitability to general purposes by focusing on data processing. In addition, Prolog has efficiency limitations, mainly due to backtracking. Thus, prototypes in Prolog are unsuitable for some specific domains. For instance, TUG should not be used for rapid prototyping in Prolog when system performance is a concern.

DCGs have been used in many applications, such as natural language processing [38], language analysis & compilers [39, 40]. We developed the TUG specification language based on the theory of DCGs. However, a specification in DCGs seems difficult to read and understand. Since the main purpose of a specification is to aid the understanding of user requirements, it is better if a specification can be read and understood easily. Therefore, another form of the representation must be acquired. We adopted regular expression notations to syntactically structure TUG specifications in order to improve readability. The theory of the specification language is also hidden from users to improve understandability. We applied the TUG approach to an industrial reverse engineering project at Viasoft in Phoenix. The project aimed to provide reverse engineering tools for analyzing legacy systems in Assembly, PL/1, and COBOL. Program analysis plays a key role in reverse engineering. Since DCGs are more expressive than context-free grammars for describing languages, we used TUG to prototype a subset of the grammars of COBOL to build tree structures in a form of array representations. The purpose of this rapid prototyping was to help us understand and create common data structures needed for analyzing COBOL, Assembly, and PL/1 legacy programs.

Automated tools need to be developed to aid the development of software using TUG. The language also needs to be improved to support the development of real-time systems. In addition, a simple user claim which is not so different from a user input re-

veals little meaning with respect to quality assurance. How to conduct theorem proving with a complex user claim in an efficient manner needs be researched as well. Currently, the language only supports the development of sequential systems. Visualization is another area of future research. With the availability of powerful workstations, visual software development represents a revolutionary departure from the traditional software development process.

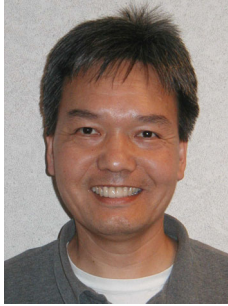
## REFERENCES

1. W. W. Agresti, "Framework for a flexible development process," in *New Paradigms for Software Development*, W. W. Agresti, ed., IEEE Computer Society Press, 1986, pp. 11-14.
2. J. Biggerstaff and C. Richter, "Reusability framework, assessment, and directions," *IEEE Software*, Vol. 4, 1987, pp. 41-49.
3. A. Diller, *Z: An Introduction to Formal Methods*, John Wiley and Sons, 1990.
4. J. M. Spivey, *The Z Notation – A Reference Manual*, 2nd ed., Prentice-Hall International, 1992.
5. I. Hayes, *Specification Case Studies*, 2nd ed., Prentice-Hall International, 1993.
6. K. M. van Hee, L. J. Somers, and M. Voorhoeve, "Z and high level petri nets," in *Proceedings of VDM 91 Formal Software Development Methods*, S. Prehn and W. Toetenel, eds., LNCS, Springer Verlag, Vol. 551, 1991, pp. 204-219.
7. X. He, "PZ nets: a formal method integrating petri nets with Z," in *Proceedings of 7th International Conference on Software Engineering and Knowledge Engineering*, 1995, pp. 173-180.
8. A. Evans, "Visualizing concurrent Z specifications," in *Proceedings of 8th Z Users Workshop (ZUM '94)*, J. Bowen and J. Hall, eds., Workshops in Computing, Springer Verlag, 1994, pp. 269-281.
9. D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith, "Object-Z: an object-oriented extension to Z," in *Proceedings of 2nd International Conference on Formal Description Techniques (FORTE '89)*, North-Holland, 1989, pp. 281-296.
10. P. Baumann and K. Lermer, "A framework for the specification of reactive and concurrent systems in Z," in *Proceedings of 15th Conference on Foundation of Software Technology and Theoretical Computer Science*, Bangalore, India, P. Thiagarajan, ed., LNCS, Springer Verlag, Vol. 1026, 1995, pp. 62-79.
11. P. Zave, "An operational approach to requirements specification for embedded systems," *IEEE Transactions on Software Engineering*, Vol. SE-8, 1982, pp. 250-269.
12. P. Zave and W. Schell, "Salient features of an executable specification language and its environment," *IEEE Transactions on Software Engineering*, Vol. SE-12, 1986, pp. 312-325.
13. D. Andrews, "Specification aspects of VDM," *Information and Software Technology*, Vol. 30, 1988, pp. 164-176.
14. D. Bjorner and C. B. Jones, *VDM '87: VDM – A Formal Method at Work*, Springer Verlag, 1987.
15. M. I. Jackson, "Developing ada programs using the Vienna development method (VDM)," *Software – Practice and Experience*, Vol. 15, 1985, pp. 305-318.

16. C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore, *Mural – A Formal Development Support System*, Springer Verlag, 1991.
17. J. Goguen and J. Meseguer, “Rapid prototyping in the OBJ specification language,” *ACM SIGSOFT*, Software Engineering Notes, Vol. 7, 1982, pp. 75-84.
18. J. Goguen, “Parameterized programming,” *IEEE Transactions on Software Engineering*, Vol. SE-10, 1984, pp. 528-543.
19. J. A. Bergstra, J. Heering, and P. Klint, *Algebraic Specification*, Addison Wesley, 1989.
20. J. V. Guttag, J. J. Horning, and J. M. Wing, “The larch family of specification languages,” *IEEE Software*, Vol. 2, 1985, pp. 24-36.
21. J. V. Guttag and J. J. Horning, “A larch shared language handbook,” *Science of Computer Programming*, Vol. 6, 1986, pp. 135-157.
22. I. V. Horebeek and J. Lewi, *Algebraic Specifications in Software Engineering*, Springer Verlag, 1989.
23. R. J. A. Buhr and G. M. Karam, “Temporal logic-based deadlock analysis for ada,” *IEEE Transactions on Software Engineering*, Vol. 17, 1991, pp. 1109-1125.
24. B. C. Moszkowski, “A complete axiomatization of interval temporal logic with infinite time,” in *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS '00)*, 2000, pp. 241.
25. R. Mattolini and P. Nesi, “An interval logic for real-time system specification,” *IEEE Transactions on Software Engineering*, Vol. 27, 2001, pp. 208-227.
26. F. Pereira and D. Warren, “Definite clause grammars for language analysis,” *Artificial Intelligence*, Vol. 13, 1980, pp. 231-278.
27. V. Berzins, Luqi, and A. Yehudai, “Using transformations in specification-based prototyping,” *IEEE Transactions on Software Engineering*, Vol. 19, 1993, pp. 436-452.
28. C. C. Chiang and J. E. Urban, “Scalable templates for specification reuse,” in *Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC '97)*, 1997, pp. 396-401.
29. C. C. Chiang and D. Neubart, “Constructing reusable specifications through analogy,” in *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC '99)*, 1999, pp. 586-592.
30. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.
31. G. Luger and W. Stubblefield, *Artificial Intelligence and the Design of Expert Systems*, Benjamin/Cummings, 1989.
32. P. Henderson and R. A. Snowdon, “An experiment in structured programming,” *BIT*, Vol. 12, 1972, pp. 38-53.
33. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.
34. H. D. Mills, “The new math of computer programming,” *Communications of the ACM*, Vol. 18, 1975, pp. 43-48.
35. B. W. Boehm, “Verifying and validating software requirements and design specifications,” *IEEE Software*, Vol. 1, 1984, pp. 75-88.
36. A. Hall, “Seven myths of formal methods,” *IEEE Software*, Vol. 7, 1990, pp. 11-19.
37. C. C. Chiang, “A formal method for proving programs correct,” in *Proceedings of 1st IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 2, 2001,

pp. 718-723.

38. H. Abramson and V. Dahl, *Logic Grammars*, Springer-Verlag, 1989.
39. J. Bowen, "From programs to object code and back again using logic programming: compilation and decompilation," *Software Maintenance: Research and Practice*, Vol. 5, 1993, pp. 205-234.
40. D. Warren, "Logic programming and compiler writing," *Software – Practice and Experience*, Vol. 10, 1980, pp. 97-125.



**Chia-Chu Chiang (蔣家駒)** is an Assistant Professor in the Department of Computer Science at University of Arkansas at Little Rock (UALR). Before joining UALR, he worked at Allen Systems Group, Inc. (formerly Viasoft) in Phoenix, Arizona where he was responsible for developing and maintaining commercial products for reengineering Assembly and PL/I legacy systems. He also worked on legacy renewal projects that transform legacy systems into distributed software component-based systems in C++ using CORBA. Dr. Chiang earned his Ph.D. degree in Computer Science from Arizona State University (ASU) in 1995. His research areas include formal methods, reverse engineering, reengineering, program analysis, component-based software development, middleware, and heterogeneous distributed parallel programming. Dr. Chiang is a member of ACM and IEEE.