ABSTRACT

Reuse has attracted attention as a promising approach for improving software productivity and quality. This paper summarizes an effort that views reuse as a higher level specification method with its own environment. Combining the knowledge of an application domain in terms of objects, functions, templates, and rules with the reuse process, a prototype for semi-automatic software generation has been developed. The prototyped reuse environment is described here, with emphasis on the Extensible Library Language (ELL). It allows users to input the specification of their systems in terms of different components of the library. In conjunction with the reuse environment, ELL provides for incremental decomposition of the missing library components into lower level ones, and combination of the components of the library that generate the code for the missing module.

A. INTRODUCTION

Automatic programming is one of the long range goals of computer science research. In its actual implementation, automatic programming always is viewed as substitution of a higher level language for specifying a system to a machine for the languages that are presently available [2]. High level languages improved software specifications and introduced basic generic and reusable programming concepts that were not encapsulated by assembly languages. A program in FORTRAN could be interpreted as a design pseudocode for an assembly program.

Similarly Very High Level (VHL) software design methods are being developed and are moving up toward greater abstraction of specifications, by relaxing syntactic rules of high level languages, or including more semantic information. The current VHL approaches are classified in groups such as design tools [4], transformation techniques [5,6], component composition methods [7,8], object-oriented approaches [9], application language [10], and knowledge-based methods [11]. The results of the efforts and research of the above approaches are compared [12, 3] and the following conclusions reached.

1. Efficient VHL software design needs to be domain related. The knowledge of application domain can be used for generation of domain related software.

2. The higher level specification method needs to be user-friendly and applicable by the ones who are familiar with the domain.

3. The higher level specification environment needs to be open for interaction and overlap between different application domains.

4. A set of domain-independent and generic constructs (e.g. objects such as arrays, stacks, and templates such as control structures) can be developed as library components and be used for any domain.

5. Considering the human factors in algorithm design and computer programming, studies have suggested [14] that programmers recall generic plans and functional components for developing a system and later are concerned about details of the interface of components (similar to component composition approaches).

6. The systematic reuse of software structures (in terms of predefined modules, objects, templates (frames and cliches are similar concepts), vocabulary, and rules) can be used for specification and implementation of the greater part of any new system.

B. BACKGROUND

On the basis of the above considerations an environment for software generation has been prototyped for the domain of graph theory [12,13]. The Graph theory Software Development System (GSDS), provides for semi-automatic software generation for an application domain, applying a library of reusable objects, functions and templates. The approach
facilitates the instantiation and composition of library components and allows higher level specifications for domain problems. Software specifications in terms of objects and components are transformed to implementations by instantiation of the component's interface parameters, evaluation of their interface rules and writing code to text files. Standardized objects and their procedures are used for automatic evaluation of interface parameters. Higher level specification and more semantic checking is possible by incorporating knowledge of an application domain in terms of attributes of objects, rules, and programming conventions.

Starting from the main module, software generation is viewed as a top-down decomposition of the modules that are missing in the library into lower level components, and a bottom-up composition of available library components [20]. Missing components in the library are developed by combining available constituent objects, templates, and lower level functions. Figure 1 includes the scenario for implementing this approach. The scenario encapsulates a model for semi-automatic software generation that can be used for any application domain and in our case for graph theory. The knowledge of application domains is hidden from our general approach.

In order to implement the scenario a model for the reuse environment was designed. The suggested model includes User Interface, Library of templates, objects and functions, Library Interface, Pool of Instantiators, User’s Environment Table, Object Transformation Base, and Text Generator. A prototype of such an environment has been developed for the domain of graph theory. Figure 2 illustrates the relationship among the components of the reuse environment.

- InvokeHIERARCHICALLY_DECOMPOSE for the initial module.

HIERARCHICALLY_DECOMPOSE:
- Search the library for the component;
  If it is located then
  - instantiate appropriately:
    - obtain required objects;
    - check for interface conditions;
    - save new objects, attributes and relations for later access;
    - provide required structures, importations, declarations, etc.;
    - provide required invocations;
    - provide for messages;
  - write resulting program into appropriate files.
  else
  - manually decompose the component into subcomponents.
  - for each subcomponent do
    call HIERARCHICALLY_DECOMPOSE.
  end
- Develop final text file of resulting program.

Figure 1. General Software Generation Scenario Used for GSDS.

Figure 2. General Design of GSDS.
B.1 LIBRARY

The library of GSDS is an application oriented software pool designed for use in developing programs for graph theory problems. The design of graph theory algorithms, as with other domains, involves general programming design structures such as stacks, loops, etc., and domain specific concepts. Consequently the GSDS library includes general design structures and graph theory objects and functions. Templates of the library represent the control and data structure of the implementation language (Modula-2) and functions of the domain that cannot be represented by generic library functions due to the limitation of the language or complexity of the functions respectively.

B.2 LIBRARY INTERFACE

Interfacing with the library serves the purpose of locating a library component that exists in the library and correctly determining when a component is missing. Library taxonomy, layered classification [15], knowledge-based library interface [16], and data base query [17] are among the available methods. Additionally, aliasing and using different views are considered for referencing the library components. Attributes such as different views, developer, sub and super components, pre and post conditions, functionality (textual description), component index (unique code used for the recognition of the component) are defined. Views of a component are different types of implementation of the component. For instance a directed graph may be viewed as a network with edges of value one. This can be an important consideration when an object with the same nature is used for different purposes in different application domains or subdomains.

B.3 USER’S ENVIRONMENT TABLE

The primary purpose of the User’s Environment Table (UET) is to serve as the repository of all current information about objects, attributes, relations, and identifications pertaining to software being developed in a session. The UET can be viewed as a dynamic symbol table that maps a user’s given component identities to the identities used for developing the user’s software (e.g. different aliasing names used for actual function calls, different default names of transformed objects developed by instantiators of transformation functions, etc.). The entities of the table are used to recognize a user’s recall of applied objects in terms of their internal representation that in turn is used for interfacing between components and instantiating their interface parameters. For instance, while interfacing two library algorithms a tree object (TREE) that internally is represented by an adjacency list needs to be transformed to a two column matrix. The new presentation is hidden from the user and the matrix representation by default may be called TREE1. The primary functions of TREE1 (e.g. matrix node access function) may be used for evaluating other interface parameters of the function using TREE1. The UET internally is a set of nodes and links for preserving object attribute relation structure among the nodes. The user-identified objects are treated as objects of the UET. Some defined attributes such as the internal structure (e.g. graph g internally is a matrix) are defined for all objects. The default presentations that are hidden from the user are attributes of digital objects. The functions that are requested by a user according to their nature, are either attributes of an object or relations among the input output objects (one to many relations). Hierarchy among objects are also defined (a vertex V1 may belong to a path P1 which in turn may belong to a graph G1).

B.4 OBJECT TRANSFORMATION BASE

An object may be viewed from different perspectives during a software generation session while interfacing different library components. Object transformation does not change the nature of an object (modifying its data), but develops new perspectives and/or internal representations for the given object. The new perspective may be a different view of an object in the GSDS library, or defined as a different object (e.g., a path is also a graph).

B.5 POOL OF INSTANTIATORS

An instantiator of a function evaluates the interface conditions for the given objects, obtains the required attributes of the objects, writes the necessary code for calling the function and updates UET. An instantiator is invoked when its associated library component is called by a user or by another instantiator. An instantiator provides a called one with the necessary attribute or transformation of input objects is needed. For the latter reason a template generator functions are used. The instantiation of the the generators (e.g. GRAPGEN used in the example in the appendix) obtain the information for
developing code for importation or appropriate declaration of the objects.

B.6 TEXT GENERATOR

The final output of the system is a program that includes data types, variables, statements to import library modules, invocation (stubs) for required functions, and bodies of templates that are evaluated with some parameters. For each program being generated three different buffers are defined; Importation, declaration, and Program (the body of a module). A patch buffer is also used for writing the code for calling functions that are invoked by others in the middle of some templates such as a loop (e.g. initialization functions). The text generator is a module of GSDDS that includes text-writing functions used by templates and instantiators. Example of such routines are functions to write array and matrix templates in the Declaration buffer.

B.7 USER INTERFACE

Communication between a user and GSDDS is based on limited natural language, including the user's input and system messages. System messages are provided by the Library Interface, instantiators, or templates. During library search for a component, the library interface informs the user about the search results. Instantiators request interface objects for instantiation of library functions and inform the user whether invalid interface objects are provided. Templates provide messages to obtain parameters to fill in the text of the template while writing the template to the program buffers. A user input may be a request for a library function or a template, or a response to a prompt for providing interface objects for functions and templates. A library interface language has been designed to be used for the User Interface of GSDDS. The next sections will describe the language. For more detailed description about the reuse environment, [12] and [13] may be referred.

C. Extendable Library Language (ELL): A generic language

The Extendable Library Language (ELL) is a generic language based on library application of templates, modules and functions. It is extendable since its acceptable vocabulary (all acceptable identities in the library), data types (implemented as library objects), control structures (implemented as library templates), and functions (library functions), are included in the library and can be modified and expanded. ELL is generic in the sense that by its nature it does not depend on any specific language's data or control structure nor any specific application domain.

Domain-specific languages include the concept of the domain (in terms of rules, objects, vocabularies, and functions) in their syntax and consequently are bound to a fixed view of a single application domain. Implementation of ELL is similar to the domain languages, however in ELL domain-specific concepts are in the library. While instantiators and templates of the library develop error-free syntax of the implementation language (Modula-2 here), ELL provides for evaluating their required parameters. The basic resource for checking the semantics of the specification provided by the user is the definition of the objects in the library and the invoked functions of the library that represent relations or attributes of the objects in UET. The precondition rules of the instantiators help for semantic checking of the interface between called functions and evaluating the interface objects passed from one to another. This semantic checking is mostly carried out in terms of known objects of an application domain rather than the type checking approach of compilers. The provided interface objects need to be defined in terms of known objects in the library or their derived objects defined by the user. This approach can be viewed as a conjunction of the knowledge of an application domain in compiling the specifications of a system developed for the domain.

Since library entities can be changed (different domains, templates and modules can be added), ELL can be used generically and even simultaneously for different application domains. A language such as ELL is well suited for implementing the "tower of languages" idea for interfacing different domains and languages. The only libraries that are used for our current prototype system is graph theory library including general parametric objects and templates. The implementation of ELL is totally library-dependent and understandable (parseable) structures by the language are expressions, Make-object construction, and syntax of library component calls. The context of the calls are verified by the instantiators or are given by the templates. Thus any example that illustrates the capability of the language need to be presented in a context of a problem and its domain library. Such an example is provided in Appendix and hopefully will help to verify our claim. Following segments describe the construct of the language and figure 3 provides the syntactic rules for ELL in terms of BNF.
ELL is designed to interface with the library; consequently the initial statement in ELL is a "command". A command can be a reference or "end" statement indicating the end of a GSDS session. A reference is a sequence of one or more of 'module-form', 'template-form', 'text_form', or 'makeobj-form'.

The 'module-form' provides syntax for calling a library importable function or module (including objects and their primary functions). A library function name that possibly is accompanied by specific or default interface parameters is used for calling the instantiator of the library function. If the interface parameter list is not complete then interface parameters must be provided, having one of the forms 'param', 'default' (\&), or end of param list (\)). A 'param' may be an identity (of a new object or an old object in the User's Environment Table), a value, or a module-form (call to a function). A call to a module may be a call to a function that performs a specific graph theory function, or a function for defining attributes of some objects. Such attributes may be defined as 'AnyMember', of sets, 'Reverse_edge' of edges, or 'First', 'Second',... members of ordered lists and arrays. The frequently used attributes makes a vocabulary set that releases the users from defining some expressions or constructs. A value can be a 'constant' (including character strings, numbers, TRUE, FALSE, and NIL) or an expression (similar to Modula-2 expressions).

Parameter list in its simplest form can be a value or an object. Function calls as well as mathematical expressions may be provided in parameter list. When a required parameter is in module_form, nested calls to the library functions may be needed. The function calls, starting from the innermost function, are carried out in the reverse order that they are nested. The output objects of the inner calls may be used for evaluating the

command ::= \{ reference >> "end" (* "end" = end of the session *) reference ::= \{ (\{someform\}) | \{template_form\} | \{makeobj_form\} | \{text_form\} \} =\}
module_form ::= \{lib_name\} [paramlist]
lib_name ::= \{name of a component in the library \}
paramlist ::= \{ \{\{param\} \{param\} \{param\} \{param\} \} \{\{param\} \{param\} \{param\} \{param\} \} \}
param ::= \{ \{constant\} | \{expression\}\} \{ \{constant\} \{expression\}\} \{ \{constant\} \{expression\}\}
constant ::= \{ \{number\} \{digit\}\} \{ \{number\} \{digit\}\} \{ \{number\} \{digit\}\}
digit ::= \{ \{0\} \{0\} \{0\} \{0\} \}
char ::= \{ \{a\} \{a\} \{a\} \{a\} \}
temp_name ::= \{name of an available template \}
makeobj_form ::= \{MAKEOBJECT\{entity_name\}\{structure\} \{MADE_OF\}\{object\}\{module_form\}\}
structure ::= \{name of a generic library object \}
object ::= \{name of an instantiated library object \}
cond_form ::= \{expression\} \{relationalop\} \{expression\}
relationalop ::= \{'=\}' \{'\&\&\}' \{'\|\|\}' \{'\lt\\lt\\}' \{'\gt\gt\\}' \{'\lt\\lt\\}' \{'\gt\gt\\}' expresion ::= \{'=\}' \{'\&\&\}' \{'\|\|\}' \{'\lt\\lt\\}' \{'\gt\gt\\}' \{'\lt\\lt\\}' \{'\gt\gt\\}' terms ::= \{'factor\'} \{'\*\}' \{'\+\}' \{'\\-\\}' \{'DIV\}' \{'\\*\}' \{'\\+\}' \{'\\-\\}' \{AND\} \{factor\}
factor ::= \{'\\(\\'} \{expression\} \{\\)\}\{'\\(\\'} \{expression\} \{\\)\}\{'\\(\\'} \{expression\} \{\\)\}
text_form ::= \{TEXT\{entity_name\}\{any_symbol\}\{ENDTEXT\}\{entity_name\}\{any_graph\}\{char\} \{digit\} \{special_chars\}\}
symbol explanation:
* * : terminals;
< > : nonterminals;
\{ \} : select exactly one of the choices;
\{ \} : select zero or one of the choices;
\{ \} : beginning of comment;
\} : end of comment.

Figure 3. BNF of ELL.
parameters of the higher level functions. The parameter list is evaluated and instantiators for the functions are called.

The 'template_form' provides syntax for calling a library template. The template_form includes the identity of a template and the required parameters that can include a condition (e.g., selection template or conditional loop). The template parameter consists of a reference including calls to templates, modules and text_form. The parameters, if not provided, will be requested by the instructions associated with the template, that are saved in the template stack after its invocation. A 'condition' is defined very similarly to the condition statements defined in most procedural languages such as Modula-2. Expression and condition statements are the lowest level user input. These statements are the only common statements of Modula-2 and ELL. The statements are very similar in most programming languages, and thus would allow the implementation of ELL for different languages with very little change (e.g., elimination of functions such as MOD). The functions, templates and instantiators of library components in other language environments than Modula-2, would generate text of the relevant language.

The 'makeobj_form' is a generic instruction for developing new objects or instances of library objects. When defining a system, objects as well as functions and templates are needed. Most objects are expected to be developed by direct call to their developer function (e.g., GRAPHGEN develops graphs) or as a result of output of an invoked function. In spite of this, in designing some algorithms, direct declaration of some specific objects such as lists, sets, etc., of specific objects are needed. In order to provide this facility, 'makeobj_form' is used to obtain the name, type of library object(s), and if desired, the initial value of the object, and the identity of the object it belongs to. Functions that interpret common programming vocabulary (referred to in module_form) can be applied suitably for defining and evaluating new objects.

The 'text_form' is provided for insertion of a piece of existing code inside the software being generated by GSXS. The text provided between "TEXT" and "ENDTEXT" is not considered by the GSXS and bypassed (similar to the comments in the programs). The text can include objects defined by the software developed by GSXS, but not vice versa. This feature may be very helpful when a missing component needs to be decomposed into very deep levels, or when the code is available in some existing software of other resources and is not available in the library.

The example in Appendix is a detailed implementation of ELL and our approach for library-based software generation for the graph theory. More detailed information about tracing between the user input algorithm and Modula-2 code can be obtained from [12].

D. SUMMARY AND CONCLUSION

The paper has summarized the effort for designing an environment for reuse and generation of software via combination of library components from high level specifications. The prototype GSXS system provides for software generation for the application domain of graph theory; though it is a generic approach and can be used for different application domains. The inclusion of different types of library components and the generic design of GSXS and ELL can be accounted as the main contributions of the research. ELL allows composition of low level constructs (expressions) with high level design components. The idea of transformation of objects is suggested by this research and allows interaction between different application domains.

Our experiment of developing software by composition of components of the GSXS library has led us to believe that though there is no "Silver Bullet" [21], library-based programming improves productivity of development and maintenance considerably. Faults of the developed system can be detected easily due to the pure modular design of the system, and application of library components that experimentally have been proven to be correct. The efficiency of the software development process relies on the knowledge of the designer about the availability and functionality of library components. This is exactly similar to the case of users of large systems (e.g. UNIX), that their benefit from the system increases as they use the system more and get more familiar with what is available. Users can thus tailor their to-down decomposition in order to develop more efficient software. This leads us to believe that the essential beneficiaries of software library systems will be software development firms or application domain programmers (e.g. programmers for business applications).

A standard approach for representing independent and application-domain objects provides an effective mechanism for interfacing with generic modules. Library generation is an evolutionary task that
requires periodic enhancement. The new
components need to be proven for their
applicability to a class of problems as
well as their correctness. Determining
generic definition of objects and their
primary functions (especially domain
object) needs to be taken a step further. Generic
objects may be viewed from different
points of view, and need to cover the
defined class of objects necessarily and
completely. Developing instantiators for
each function was a tedious job. Design of
a generic instantiator that implements
some standard information about
instantiating functions has been
considered.

APPENDIX

This appendix includes the three
steps, original algorithm, input to the
GSDS system, and the Modula-2 code for
determining an Eulerian circuit of a
diagram, a popular and historical graph
theory problem.

1. Original Algorithm

The Eulerian algorithm suggested by
Fleury [18] is formally described by
Gibbons [19]. This algorithm does not
include any test to determine if a graph
is Eulerian. The algorithm is modified by
adding the test and is included in figure
4. The required objects are described
prior to the algorithm.

2. User Input Algorithm and
Generated Software

The pseudo-code developed for the
Euler problem must be expressed in terms
of the library components of GSDS. The
general algorithm in figure 4 was designed
without any knowledge of the existence and
identity of the library functions. This
verifies that any application domain
library that includes the appropriate and
frequently-used objects and functions can
very effectively be used for designing the
software for the domain. The algorithm
must be transformed to calls of library
components in the forms of ELL commands
(e.g., 'make-object' or expressions and
available library names). The Eulerian
test function for a graph does not exist
in the library and needs to be decomposed
into lower levels. A graph is Eulerian if:
1) it is connected and 2) does not have
any odd vertices. The latter does not
exist in the library, and is decomposed
into the graph theory template
ALLVERTICES, the selection templates (IF),
and the function DEGREE. The generic
parameter 'anyvertex' is defined for
ALLVERTICES template as: vertex(i), i :=
(1 ... P); where P is the size of the
graph. ALLVERTICES is the template that
facilitates access to all of the vertices of a graph. The DEGREE function and
SELECTION template are used as the body of
the ALLVERTICES template. In order to
decide if a degree of a vertex is odd, a
mathematical expression is used. At this
point any low level programming
constructs, namely mathematical
expressions, are included in the
composition process for software
generation. NPLUS function provides a list
of the vertices (EC) that are adjacent
from a vertex (CV) for the graph that by
default is G. This example illustrates
how different levels of abstraction are
combined and used for software
development. Figure 5 provides the
software specifications presented by the
user.

(* ALGORITHM TO FIND EULER CIRCUIT OF A
GRAPH, MODIFIED FROM ALGORITHM 2B (FLEURY)
OF CHAPTER TWO OF THE REFERENCE

FOLLOWING OBJECTS ARE NEEDED IN THE
ALGORITHM.

W = FIRST VERTEX THAT IS VISITED.
EC = LIST OF VERTICES IN THE ORDER THAT
ARE VISITED.
CV = CURRENT VERTEX.
ACV = ADJACENCY LIST OF THE CV.

FOLLOWING IS THE BASIC ALGORITHM.
*)
1. MAKE THE GRAPH
2. TEST IF GRAPH IS EULERIAN
   (CONNECTED AND EVERY VERTEX IS
   OF EVEN DEGREE).
3. IF NOT EULERIAN THEN
4. OUTPUT NO EULER CIRCUIT.
5. HALT
6. END
7. START WITH ORIGINAL VERTEX W = 1 :
8. ADD W TO EC;
9. SET CURRENT VERTEX TO W (CV = W);
10. SET ACV TO ADJACENCY LIST OF
    CURRENT VERTEX;
11. WHILE Adjacency List of Current Vertex
    IS NOT EMPTY DO
12.   CHOOSE A VERTEX V SUCH THAT :
13.   - V IS ADJACENT TO CV
14.   - (CV, V ) IS THE FIRST INCIDENT
    EDGE TO CV THAT IS NOT A BRIDGE.
15.   IF NO SUCH VERTEX, SET V TO THE
    FIRST ADJACENT VERTEX OF
    CURRENT VERTEX;
16.   REMOVE THE EDGE (CV, V) FROM
    THE GRAPH;
17.   ADD V TO THE END OF EC;
18.   CV := V;
19.   SET ACV TO ADJACENCY LIST OF
    CURRENT VERTEX.
20. END
21. OUTPUT EC.

Figure 4. Algorithm for testing and finding
Eulerian circuit of graphs.
In this example the requests for components include the necessary objects, with the exception of the requests for GRAPHGEN and WRITEACIRCUIT. These two function calls exemplify the situations in which the GSDo user does not have information about the nature of the interface objects of the function. The messages provided by the GRAPHGEN and WRITEACIRCUIT instantiators help the user to provide the proper information. The comments in (*) and **) are the messages provided for the user by the instantiators or by the templates.

The developed program for the user input is provided in figure 6. The program imports the functions and declarations of objects that are needed for the function interfaces and declarations.

4. GSDo:
   (* Welcome to Graph theory Software Generation System.
   What is the name of your program? (1 to 8 letters or numbers starting with a letter). *) EULER

1. GRAPHGEN:
   (*) What is the name of your graph? /* A name starting with a letter. */ G
   (*) What is the number of the vertices of your graph? */ 9
   (*) What is the number of the edges of your graph? */ 36
   (*) Considering following graphs and their corresponding code, give the number representing type of your graph.
   0 - UNWEIGHTED GRAPH
   1 - NUMERIC WEIGHTS ON THE EDGES
   2 - NUMERIC WEIGHTS ON THE VERTICES
   3 - NUMERIC WEIGHTS ON THE EDGES AND VERTICES
   4 - SYMBOLIC WEIGHTS ON THE EDGES
   5 - SYMBOLIC WEIGHTS ON THE EDGES AND VERTICES
   6 - NUMERIC WEIGHTS ON THE EDGES AND SYMBOLIC HEIGHTS ON THE VERTICES
   7 - SYMBOLIC HEIGHTS ON THE EDGES AND NUMERIC HEIGHTS ON THE VERTICES
   8 - NUMERIC HEIGHTS ON THE EDGES AND SYMBOLIC WEIGHTS ON THE VERTICES.

2. EULERTEST G:
   (* EULERTEST DOES NOT EXIST; DECOMPOSE OR DEEP-SEARCH *)
   2.1 SELECTION (CONNECTED G) = FALSE ;
   2.2 OUTPUT 'GRAPH IS NOT EULERIAN.' ;
   2.3 HALT ;
   2.4 \ \ (** END OF TRUE ACTION *)
   (** NO FALSE ACTION *)
   (** END SELECTION *)

2.5 ANYGODOVEX G :
   (* ANYGODOVEX DOES NOT EXIST, DECOMPOSE OR DEEP-SEARCH *)

2.5.1. ALLVERTICES G :
   2.5.2. DEGREE G ANYVEX DEG ; (* BEGINNING OF THE BODY OF THE TEMPLATE *)
   2.5.3. SELECTION ( (DEG DIV 2) = 2 ) ;
   2.5.4. OUTPUT ' CIRCUIT IS NOT EVEN ' ;
   2.5.5. HALT :
   2.5.6. \ \ (** END OF TRUE ACTION *)
   (** NO FALSE ACTION *)
   (** END SELECTION *)

2.5.7. \ \ (** END OF ALLVERTICES TEMPLATE *)

3. MAKEGODOVEX CV : VERTEX = 1 OF G ;
   4. ADDLISTENDNODE EC CV ;
   5. HPLUS / CV ACV ;
   6. \ \ (** BODY OF THE COND_LOOP TEMPLATE *)
   7. MAKEGODOVEX I : COUNTER = 1 ;
   8. GTRVPIOR ACV I V ;
   9. \ \ (** BODY OF THE COND_LOOP TEMPLATE *)
   10. \ \ (** BODY OF THE COND_LOOP TEMPLATE *)
   11. \ \ (** BODY OF THE COND_LOOP TEMPLATE *)
   12. \ \ (** BODY OF THE COND_LOOP TEMPLATE *)
   13. SELECTION ( ( CV ) ) ;
   14. GTRVPIOR ACV I V ;
   15. \ \ (** END OF TRUE ACTION *)
   (** NO FALSE ACTION *)
   (** END SELECTION *)

14. DELETE EDGE G CV V ;
15. ASSIGN CV V ;
16. ADDLISTENDNODE EC CV ;
17. HPLUS / CV ACV ;
18. \ \ (** END COND_LOOP *)
19. WRITEACIRCUIT EC ;
20. \ \ (** GIVE A TITLE FOR OUTPUT *)
21. \ \ (** AN EULERIAN CIRCUIT OF G *)
22. END (** END SESSION WITH GSDo *)

Figure 5. The input of the user for the algorithm of figure 4.
```haskell
MODULE EULER;
FROM SYSTEM IMPORT BYTE, ADR, ADDRESS, SIZE, TSIZE;
IMPORT INTCNODE;
FROM MATRICES IMPORT MATRIXGEN, READINDEX, MATRIXPRINT;
FROM GRAPHATR IMPORT CONNECTED;
FROM INOUT IMPORT WRT Writestring;
FROM VERTICES IMPORT DEGREE;
FROM LINKLIST IMPORT ADDLISTENDNODE, INITIALIZELIST;
FROM VERTICES IMPORT NPLUS;
FROM LINKLIST IMPORT LISTPOINTER;
FROM ORDLISTS IMPORT OLISTHEAD, MAKEOLIST, GETOLISTOBJ;
FROM VERTICES IMPORT VERTEXADR;
FROM EDGES IMPORT ISABRIDGE;
FROM LINKLIST IMPORT LISTNODECOUNT;
FROM GRAPHMOD IMPORT DELETEEDGE;
FROM GWAN IMPORT WRITEAWALK;

TYPE MATRIX1
VAR G : MATRIX1;
= ARRAY [1..g, 1..9] OF INTCNODE.NODE;
VAR P1, Q1, GRAPHTYPE1 : CARDINAL;
VAR I2 : INTEGER;
VAR DEG : CARDINAL;
VAR DEG3 : POINTER TO CARDINAL;
VAR CV : VERTEXNODE;
VAR EC : LISTPOINTER;
VAR ACV : LISTPOINTER;
VAR I : CARDINAL;
VAR ACV4 : OLISTHEAD;
VAR V: VERTEXNODE;
VAR V5 : VERTEXADR;
BEGIN
P1 := 9:
Q1 := 36:
GRAPHTYPE1 := 110:
INTGNODE.INPUTDESCRIPTION (GRAPHTYPE1):
MATRIXGEN (ADR(G), INTCNODE.NULLVALUE, P1, Q1, GRAPHTYPE1,
MATRIXPRINT (ADR(G), P1, SIZE( INTCNODE.NULLVALUE), 'G', INTCNODE.WRITE):
IF CONNECTED (ADR(G), P1, GRAPHTYPE1, INTCNODE.NULLVALUE) = FALSE THEN
Writestring ("GRAPH IS NOT EULERIAN ");
HALT:
END (* IF *);
FOR I2 := 1 TO P1 DO
DEG3 := DEGREE (ADR(G), I2, P1, INTCNODE.NULLVALUE):
DEG := DEG3:
IF ((DEG DIV 2) * 2) = DEG THEN
Writestring ("GRAPH IS NOT EULERIAN ");
HALT:
END (* IF *);
CV := 1:
INITIALIZELIST (EC):
ADDLISTENDNODE (EC, CV, NEWVERTEX):
ACV := NPLUS (ADR(G), CV, P1, INTCNODE.NULLVALUE):
WHILE ACV # NIL DO
I := 1:
MAKELIST (ACV, ACV4):
V5 := (GETOLISTOBJ (ACV4, I)):
V := V5:
WHILE ISABRIDGE (G, P1, CV, V, GRAPHTYPE1, INTCNODE.NULLVALUE) AND (LISTNODECOUNT (ACV) >= I) DO
INC(I):
V5 := GETOLISTOBJ (ACV4, I):
V := V5:
END (* WHILE *):
IF (I > LISTNODECOUNT (ACV)) THEN
V5 := GETOLISTOBJ (ACV4, I):
V := V5:
END;
DELETEEDGE (ADR(G), P1, CV, V, GRAPHTYPE1, Q1, INTCNODE.NULLVALUE):
CV := V:
ADDLISTENDNODE (EC, CV, NEWVERTEX):
ACV := NPLUS (ADR(G), CV, P1, INTCNODE.NULLVALUE):
END (* WHILE *):
WRITEAWALK (EC, "AN EULERIAN CIRCLE OF G");
END EULER.
```

Figure 6. Modula-2 program for algorithm in figure 4.
REFERENCES


