An Experimental Natural-Language Processor for Generating Data Type Specifications

by James R. Comer
Texas Christian University
Ft. Worth, Texas 76129

1. Introduction

Computer system hardware architecture has experienced significant progress and growth over the past 40 years. So much so, that today's computer user community is confronted with a very realistic software predicament that can be directly traced to this rapidly changing technology [1]. Unfortunately, during this same time frame, productivity of software designers, and the tools with which they work, has improved rather slowly. Only in recent years, has any significant effort been paid to the development of techniques for improving the systematic design of well-structured software architecture.

A continuing goal of computer science has always been to make the computer more accessible to a user community not specifically trained in its use. To this end, considerable effort has been expended to improve the languages with which programmers must work. However, until recently, little attention has been paid to the languages with which software designers must work. While it is true that any practice or technique which reduces the cost of creating a program is useful (e.g., structured and GO-TO- less programming), it is not necessarily true that it impacts the main problem of reducing overall software development costs. To accomplish this, new software design methodologies and strategies are needed that go far beyond mere programming techniques.

If software systems are to be developed to meet given design goals, the designer must have viable methods of specifying, analyzing, and evaluating the performance and organization of system components prior to any attempt to implement the proposed solution. To achieve this, the designer must be able to develop models of the various system components and, to use these models to evaluate the performance of the completed system.

Regrettably, at present, no technique exists which can guarantee total certification of a sizable software system. Since the design of large systems is inherently a creative process, the designer must frequently be able to translate a

set of often times vague system requirements into an organized and well-defined system. The approach used by the designer has generally had a profound effect upon the overall quality of the final product and its associated cost. In the past, too many systems have been designed and implemented using ad hoc, disorganized techniques. Experience has demonstrated that such methods are especially inadequate when applied to the development of very large software systems. To overcome such problems recent design methodologies such as the life-cycle development approach [2,3,4,5], requirements engineering [6], structural analysis [7], and automated design aids [8,9,10,11,12,13,14,15] have been, and continue to be, investigated with considerable interest. Of equal importance, has been the development of languages designed to aid in the creation of software specifications [16,17,18].

As with programming languages, much of the recent effort in the development of specification languages has been to make them more user-oriented. Notations such as the predicate calculus are losing favor among designers of software. However, while newer attempts at specifications languages are underway, they are, still artificial languages, and as such require special training for their use. With the goal of minimizing the effort required to produce software specifications, a project was undertaken to develop an automated system with which data specifications could be created through natural language dialogue with the computer.

The purpose of this paper is to serve as an introduction to this project. A short example session is given to illustrate how interaction with the system proceeds and what processing takes place.

2. Sample Problem

Figure 1 shows a sample problem done with the system. Statement numbers appearing in this figure were added for ease of reference in the discussion. Sentences preceded by an asterisk are system responses. All others are generated by the system user.

As with other automated systems [19,20], a typical session consists of three phases: problem acquisition (lines 1 through 17), model verification (line 18), and finally automatic generation of data type specifications (line 19). The dialogue shown in the following figure is intended to correspond roughly to these three distinct phases.

1. DECLARE A STACK WITH FOUR ENTRIES.
   * STACK $016 HAS BEEN CREATED.
2. ENTITLE THE STACK LIFO.
   * THE SPECIFIED ENTITY NOW NAMED LIFO.
3. ASSEMBLE A SET WITH 3 MEMBERS WITHIN LIFO NAMED UNIVERSE.
   * SET UNIVERSE HAS BEEN CREATED.
4. FORMULATE A 1-DIMENSIONAL ARRAY WITH TWO ITEMS BESIDE UNIVERSE NAMED VECTOR.
   * BESIDE NOT SPECIFIC: 'AFTER' USED
   * ARRAY VECTOR HAS BEEN CREATED.
5. FABRICATE A QUEUE WITH 4 ELEMENTS NAMED ALPHA WITHIN VECTOR.
   * QUEUE ALPHA HAS BEEN CREATED.
6. BUILD A QUEUE WITH 2 ELEMENTS NAMED BETA BESIDE ALPHA.
   * BESIDE NOT SPECIFIC: 'AFTER' USED
   * QUEUE BETA HAS BEEN CREATED.
7. FASHION 2 STRINGS WITHIN ALPHA.
   * STRING $022 HAS BEEN CREATED.
   * STRING $021 HAS BEEN CREATED.
8. TERM THE FIRST STRING WITHIN ALPHA, NAME_FLD.
   * THE SPECIFIED ENTITY NOW NAMED NAME_FLD.
9. TAG THE SECOND STRING INSIDE ALPHA, ADRES_FLD.
   * THE SPECIFIED ENTITY NOW NAMED ADRES_FLD.
10. CREATE TWO NUMBERS IN BETA.
    * NUMBER $024 HAS BEEN CREATED.
    * NUMBER $023 HAS BEEN CREATED.
11. BUILD A BOOLEAN-VALUE AFTER VECTOR CALLED BOOL.
    * BOOLEAN BOOL HAS BEEN CREATED.
12. CONSTRUCT A CIRCULAR LINKED LIST WITH 3 NODES AFTER BOOL ENTITLED CIR_LIST.
    * LIST CIR_LIST HAS BEEN CREATED.
13. CREATE 3 NUMBERS WITHIN CIR_LIST.
    * NUMBER $029 HAS BEEN CREATED.
    * NUMBER $028 HAS BEEN CREATED.
    * NUMBER $027 HAS BEEN CREATED.
14. CONSTRUCT A LOGICAL AFTER ADRES_FLD NAMED SEX_FLD.
    * BOOLEAN SEX_FLD HAS BEEN CREATED.
15. BUILD A NUMBER CALLED AGE_FLD BESIDE ADRES_FLD.
    * BESIDE NOT SPECIFIC: 'AFTER' USED
    * NUMBER AGE_FLD HAS BEEN CREATED.
16. RENAME THE QUEUE CALLED BETA, GAMMA.
    * THE SPECIFIED ENTITY NOW NAMED GAMMA.
17. BUILD 3 NUMBERS WITHIN UNIVERSE.
    * NUMBER $034 HAS BEEN CREATED.
    * NUMBER $033 HAS BEEN CREATED.
    * NUMBER $032 HAS BEEN CREATED.
18. EXHIBIT THE STACK NAMED LIFO.

```
1  LIFO: STACK.
   2  UNIVERSE: SET.
      3  $032:  NUMBER.
      3  $033:  NUMBER.
      3  $034:  NUMBER.
   2  VECTOR:  1 DIMENSIONAL ARRAY.
      3  ALPHA: QUEUE.
         4  NAME_FLD: CHARACTER STRING.
         4  ADRES_FLD: CHARACTER STRING.
         4  AGE_FLD:  NUMBER.
         4  SEX_FLD: BOOLEAN.
      3  GAMMA: QUEUE.
         4  $023:  NUMBER.
         4  $024:  NUMBER.
   2  BOOL: BOOLEAN.
   2  CIR_LIST: CIRCULAR LINKED LIST.
      3  $027:  NUMBER.
      3  $028:  NUMBER.
      3  $029:  NUMBER.
```

19. GRAPH THE STACK ENTITLED LIFO.

Figure 1

The example shown here is an attempt to illustrate the variety of vocabulary permitted and the freedom of sentence structure enjoyed by the user. The rather bizarre model developed for this example illustrates how easy it is to construct highly complex structures (albeit, it is difficult to conceive of an application calling for such a strange data structure).

Problem acquisition is, therefore, a natural language dialogue consisting of statements entered by the user. The system responds that a required action is completed or with an error message if it is unclear as to what the user has requested. At any time during the session, the user may verify the model, or any portion of it by requesting that the model be displayed (line 18). If the user is satisfied with what has been created, he may elect to continue conversing with the system or to request that design specifications be created (line 19).

Automatic production of design specifications results in the generation of data type specifications for each item in the model. Specifications for the stack named "LIFO" and the set named "UNIVERSE" are shown in Figures 2 and 3 respectively. Others have been omitted for brevity.

3. Notation

In presenting a methodology for specifying data types, it is imperative to establish a convenient notation for describing the abstractions involved. The notation selected should be able to specify unambiguously a proposed data type without unduly influencing the eventual means of implementation. The notation developed and used in the system described herein is based on the axiomatic specification technique proposed by Guttag, Horowitz, and Musser [21,22,23,24]. The format chosen for expressing the specifications is adapted from research due to Belford, Bond, Heuderson, and Sellers [25].

4. Discussion

The natural language processing system described herein Benefits from the limited subset of English that is needed

1 'LIFO' - A STACK CONTAINING A MAXIMUM OF N ENTRY(S).

DECOMPOSITIONAL ELEMENT

INPUT (FUNCTIONAL STIMULUS)	PROCESS (FUNCTIONAL ARGUMENT)	OUTPUT (FUNCTIONAL RESPONSE)
1) LIFO	ISEMPTYSTACK(LIFO)	'BOOLEAN'
2) (LIFO , ENTRY)	PUSH(LIFO , ENTRY)	LIFO
3) LIFO	POP(LIFO)	LIFO
4) LIFO	TOP(LIFO)	ENTRY

PERFORMANCE REQUIREMENTS

FOR ALL LIFO ∈ STACK, ENTRY ∈ STACK_ENTRY LET

```
*  ISEMPTYSTACK( 'EMPTYSTACK' )  ::=  'TRUE'
*  ISEMPTYSTACK(PUSH( LIFO , ENTRY ))  ::=  'FALSE'
*  TOP( 'EMPTYSTACK' )  ::=  'UNDEFINED'
*  POP( 'EMPTYSTACK' )  ::=  'UNDERFLOW'
*  PUSH( 'FULLSTACK' , ENTRY )  ::=  'OVERFLOW'
*  POP(PUSH( LIFO , ENTRY ))  ::=  LIFO
*  TOP(PUSH( LIFO , ENTRY ))  ::=  ENTRY
```

DATA ACCESS AXIOMS

LET LIFO = {D<1>, ... ,D<N>} SUCH THAT:
D<I> ∈ STACK_ENTRY ∈ {LIFO} = N THEN:

```
** POP( LIFO )  ==>  {D<1>, ... ,D<N-1>}  ∈  {LIFO}  =  N-1
** PUSH( LIFO , X )  ==>  {D<1>, ... ,D<N>,X}  ∈  {LIFO}  =  N+1
```

Figure 2

/ 'UNIVERSE' - A SET CONTAINING 1 MEMBER(S).

DECOMPOSITIONAL ELEMENT

INPUT (FUNCTIONAL STIMULUS)	PROCESS (FUNCTIONAL ARGUMENT)	OUTPUT (FUNCTIONAL RESPONSE)
1) UNIVERSE	ISEMPTYSET(UNIVERSE)	'BOOLEAN'
2) (UNIVERSE , MEMBER)	INSERT(UNIVERSE , MEMBER)	UNIVERSE
3) (UNIVERSE , MEMBER)	DELETE(UNIVERSE , MEMBER)	UNIVERSE
4) (UNIVERSE , MEMBER)	MEMBER(UNIVERSE , MEMBER)	'BOOLEAN'
5) (UNIVERSE , SET1)	UNION(UNIVERSE , SET1)	UNIVERSE

PERFORMANCE REQUIREMENTS

```
FOR ALL    (UNIVERSE,SET1)  ε  SET,  (ITEM1,ITEM2)  ε  SET_MEMBER   LET

*   ISEMPTYSET( 'EMPTYSET' )  ::=  'TRUE'
*   ISEMPTYSET( INSERT( UNIVERSE , ITEM1 ))  ::=  'FALSE'
*   MEMBER( 'EMPTYSET' , ITEM1 )  ::=  'FALSE'
*   MEMBER( INSERT( UNIVERSE , ITEM1 ) , ITEM2 )
                    IF  ITEM1 = ITEM2
                        THEN  'TRUE'
                        ELSE  MEMBER( UNIVERSE , ITEM2 )
*   DELETE( 'EMPTYSET' , ITEM1 )  ::=  'EMPTYSET'
*   DELETE( INSERT( UNIVERSE , ITEM1 ) , ITEM2 )
                    IF  ITEM1 = ITEM2
                        THEN  DELETE( UNIVERSE , ITEM2 )
                        ELSE  INSERT( DELETE( UNIVERSE , ITEM2 ) , ITEM1 )
*   UNION( UNIVERSE , 'EMPTYSET' )  ::=  UNIVERSE
*   UNION( UNIVERSE , SET1 )  ::=  (ALL ELEMENTS X S.T.
                                    X IS IN UNIVERSE OR X IS IN SET1 )
```

DATA_ACCESS_AXIOMS

```
LET   UNIVERSE = (D<1>, ... ,D<N>)   SUCH THAT:
                 D<I>  ε  SET_MEMBER  &  |UNIVERSE|  =  N    THEN:

      ** INSERT( UNIVERSE , X )  ==>  (X,D<1>, ... ,D<N>)  &  |UNIVERSE|  =  N+1
      ** DELETE( UNIVERSE , D<I> )  ==>  (D<1>, ... ,D<I-1>,D<I+1>, ... ,D<N>)  &  |UNIVERSE|  =  N+1
```

Figure 3

for describing data structures and data types. In the case of more complex models (e.g., operating systems and data base systems), the domain of discourse would naturally be much broader, necessitating more elaborate decoding rules -- probably augmented by extensive tables and decoding formulas. It is felt, though, that the results obtained to date are sufficiently rewarding to encourage further research in the area of specifications automation.

Acknowledgments

I wish to thank my former colleagues, Professors Dan D. Drew and Anthony P. Lucido for their guidance during the development of this project.

REFERENCES

1. Myers, Ware, The Need for Software Engineering, COMPUTER Vol.11, no. 2 (February 1978), pp. 12-26.

2. Boehm, B.W., Software Engineering, TRW SOFTWARE SERIES, TRW-SS-76-08 (October 1976).

3. Dreyfus, J.M., and Karacsony, P.J., The Preliminary Design as A Key to Successful Software Development, TRW SOFTWARE SERIES, TRW-SS-76-09 (October 1976).

4. Zelkowith, M.V., Perspectives on Software Engineering, COMPUTING SURVEYS vol.10, no.2 (June 1978), pp. 197-216.

5. Williams, R.D., Managing the Development of Reliable Software, TRW SOFTWARE SERIES TRW-SS-75-06 (April 1975).

6. Bell, Thomas E., Bixler, David., and Dyer, Margret E., An Extendable Approach to Computer-aided Software Requirements Engineering, TRW SOFTWARE SERIES, TRW-SS-76-05 (July 1976).

7. Ross, Douglas T., and Schoman, Kenneth E., Structured Analysis for Requirements Definition, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING ,vol. SE-3, no. 1 January 1977), pp. 6-15.

8. Brown, J.R., and Hoffman, R.H., Automating Software Development: A Survey of Techniques and Automated Tools,

TRW SOFTWARE SERIES, TRW-SS-72-03 (May 1972).

9. Lanzano, Bernadine C., Program Automated Documentation Methods, TRW SOFTWARE SERIES, TRW-SS-70-04 (November 1970).

10. Krause, K.W., Smith, R.W., and Goodwin, M.A., Optimal Software Test Planning Through Automated Network Analysis, TRW SOFTWARE SERIES, TRW-SS-73-01(April 1973).

11. Howden, William E., DISSECT-a Symbolic Evaluation and Program Testing System, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, vol. SE-4, no. 1 (January 1978), pp. 70-73.

12. Ramamoorthy, C.V., and Siu-Bun, F. Ho, Testing Large Software With Automated Software Evaluation Systems, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, vol. SE-1, no. 1 (March 1975), pp. 46-58.

13. Boehm, B.W., Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design, TRW SOFTWARE SERIES, TRW-SS-74-02 (August 1974).

14. Hammer, M.M., Howe, W.G., and Wladawsky, I., An Interactive Business Definition System, SIGPLAN NOTICES, vol. 9, no. 4 (April 1974), pp. 25-33.

15. Heidorn, G.E., Automatic Programming Through Natural Language Dialogue: A Survey, IBM JOURNAL OF RESEARCH AND DEVELOPMENT, vol. 20, no. 4(July 1976), pp. 302-313.

16. Rigo, Joseph T., How to Prepare Functional Specifications. DATAMATION (May 1974), pp. 78-80.

17. Ziegler, E.W., An Introduction to the UMTA Specification Language, SIGPLAN NOTICES, vol. 9, no. 4 (April 1974), pp. 127-132.

18. Robinson, Lawrence, and Roubine, Olivier, SPECIAL--A Specification and Assertion Language, Technical Report No.CSL-46, Stanford Research Institute (September 1976).

19. Heidorn, George, English As A Very High Level Language For Simulation Programming, IBM Research Report RC-4536, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. (September 1973).

20. Heidorn, George, Simulation Programming through Natural Language Dialogue, IBM Research Report RC-4535, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. (September 1973).

21. Guttag, John V., Horowitz, Ellis, and Musser, David R., Abstract Data Types and Software Validation, USC/Information Sciences Institute, RR-76-48 (November 1976).

22. Guttag, John V., Horowitz, Ellis, and Musser, David R., The Design of Data Type Specifications, USC/Information Sciences Institute, RR-76-49 (November 1976).

23. Guttag, John V., Horowitz, Ellis, and Musser, David R., Some Extensions to Algebraic Specifications, SIGPLAN Notices, vol. 12, no. 3 (March 1977), pp. 63-67.

24. Guttag, John V., The Specification and Application to Programming of Abstract Data Types, Ph.D. Thesis, University of Toronto, Department of Computer Science (1975), available as Computer Systems Research Group Report CSRG-59.

25. Belford, P.C., Bond, A.F., Henderson, D.G., and Sellers, L.S., Specifications: A Key to Effective Software Development, Proceedings of the 2nd International Conference on Software Engineering (October 1976), pp. 71-79.