A.M. de A. PRICE

SUÍTARIO
Este trabalho apresenta uma descrição do Software Design Assistant (SODA), uma ferramenta automática conversacional para auxiliar projetistas de software no desenvolvimento de novas aplicações. SODA aceita a especificação gradativa de software fornecendo procedimentos para a verificação de consistência e integridade das especificações, e permitindo consultas ao software em projeto. SODA fornece, também, análises e documentação do projeto, auxílio para sua própria utilização e meios para integrar as fases de projeto e implementação de software.

ABSTRACT
This work introduces the Software Design Assistant (SODA), an interactive automatic tool for helping software developers in the design of new computer applications. SODA supports incremental design providing procedures for checking the consistency and completeness of specifications and for accepting enquiry on the system being designed. It provides design analysis and documentation, helping support and means for integrating the design process with the implementation phase.

* Engenheira Química (UFRGS, 1972), Mestre em Informática (PUC-RJ, 1976), Doctor in Philosophy (University of Sussex, U.K., 1984); engenharia de Software: projeto, estimativa de custo e medidas de complexidade de software; Pós-Graduação em Ciência da Computação, Av. Osvaldo Aranha, 99 Porto Alegre - RS.
1. Introduction

It is well-known that the cost of software is now the most expensive element of a computer-based system. While hardware costs have continue to drop over the last decade, the cost of software has been rising and now can reach the figure of 85 percent of the total computer system expenditure [Wasserman 82].

These is a need to control the rising costs of software by improving both the process of software production and the quality of the resulting software. A reasonable solution to control software rising costs, over a short term is to make software personnel more effective by providing them with techniques and automatic tools for software development.

Modern software development tools are claimed to increase both the quality of the product and developer productivity, and to enhance job satisfaction for the tool user [Riddle 80].

Although there exist a number of effective software tools, there is a shortage of tools to assist with requirements specification, design and testing [Wasserman 82]. According to Boehm, the cost of system analysis and design account for about 40 percent of the software development costs. Additionally, errors in the design phase which are not detected until testing commences can be very expensive to correct [Sommerville 82].

The motivation for the development of SODA [Price 84b] originated from the above considerations.

2. The Software Design Assistant

The Software Design Assistant (SODA) is an interactive software design tool providing the following capabilities:
- supporting of incremental software design;
- consistency and completeness checking of design specifications;
- a library mechanism for storing software constructs which can be instantiated as components of the software being designed;
- supporting the design of parallel processing systems;
- supporting enquiry on the system being designed;
- providing design analysis and documentation;
- providing means for integrating the design process with the implementation phase;
- providing helping support, explaining SODA behaviour to the user at each stage of its duty.

SODA enforces a design methodology providing mechanisms to
ensure that the design consistency is maintained across levels of detail. It primarily supports the top-down approach to modular decomposition with flexibility to accept the refinement of lower level modules before completing the specification of higher level ones. It is oriented towards functional specification but also promotes design with data abstractions. SODA perceives a software system as a set of hierarchically structured modules, which can be either sequential (programs, subroutines of functions) or parallel processing (processes) segments. Inter-module data communication is accomplished by parameters and global data is restricted to external files for achieving module independence and information hiding.

SODA supports the design specification of sequential and concurrent systems. Both approaches to process cooperation followed by Ada and Concurrent Pascal can be described by DSL, the SODA design language. DSL provides means for specifying communication and synchronization of processes, including statements such as the REQUEST/ACCEPT operation for process communication and SELECT for describing non-determinism. Mutual exclusion and delaying of monitor operations can be described by the statement SELECT provided with guarded conditions.

3. DSL - The Design Specification Language

Design specifications are input in a PDL-like language [Gaine77] which includes means for describing data and module behaviour. PDL's have proved to be worthy for representing software behaviour for many reasons (e.g. easy to learn, convenient because of their similarity to programming languages, straightforward path to implementation).

The framework for module specification is illustrated below.

```
MODULE <name> : <module type> ;

PARAMETER
  INPUT <parameter declaration list> ;
  OUTPUT
  EXTERNAL <file identifier list>; INOUT

CALL <list of called operations/subprograms> ;
TYPE <data type declarations> ;
DATA <data structure declarations> ;
BEHAVIOUR <statement> ;
OPERATION <list of offered operations> ;
SUBCOMPONENT <identification of subcomponents> ;

END
```
The design specifications for a module do not need to be input in the same order as the one presented by the framework above, neither has the entire framework to be input in one sole session with SODA. The designer can input some of the declarations at one time and complete the specifications in later sessions.

SODA includes an interactive SDL processor that provides procedures for analysing the syntax and checking the consistency and completeness of design specifications against the Design Specs Base (DSB). The major checking procedures involve searching for missing definitions and ensuring appropriate use of the defined entities. Inconsistent specifications are discarded but incomplete ones are stored with a special sign to agree with the top-down design approach which accepts references to entities that will be defined later. At each interaction with the system, besides entering new information the designer can also update the information already stored in the DSB. Updating includes commands for appending and deleting parameters, data definitions, behaviour statements, and so on.

DSL is a strongly typed language. It provides a few standard types, but enable its users to define their own data and module types. DSL also accepts parameterized abstract data types which enables different classes of data abstractions to be defined by a single data type definition. It provides a type EITHER which permits the union of any data types and can be used for specifying recursive types. A type EMPTY is provided for specifying empty objects. The combination of types EITHER and EMPTY permits the definition of dynamic data structures without making use of pointer mechanisms [Price 84a]. An example of a type definition for a stack data structure follows.

```plaintext
TYPE Stack: EITHER EMPTY
  OR RECORD
    Cell: Celltype;
    Bottom: Stack;
  END;
```

The DSL user is also permitted to define modular types. This feature is particularly useful when a modular type is defined as an ALGORITHM in the SODA Library so it can be instantiated as a component of any software project.

DSL includes the procedural specification of abstract data types. There are some advantages of using a procedural language for specifying abstract data types (e.g. semantic descriptions are easy to write, understand, and implement because of their similarity to conventional programming). Abstract data types are declared in DSL as being of type ADT
and are basically composed of two parts: (1) DATATYPE which introduces the type that characterizes the class of data objects defined by the ADT, and (2) a set of operations (subroutines/functions) that can be performed on the objects. A type definition in DSL for the binary search tree abstraction is shown below.

TYPE BinaryTree: ADT;

PARAMETER
  INPUT RootType : TYPE ;

DATATYPE =
  EITHER EMPTY
OR RECORD
    Root: RootType ;
    Left: BinaryTree (RootType) ;
    Right: BinaryTree (RootType) ;
END;

OPERATION

Search : FUNCTION LOGICAL ;
PARAMETER
  INPUT Tree : BinaryTree (RootType) ;
    Label: RootType ;

BEHAVIOUR

  if EXIST (Tree)
  then if Label = Tree.Root
    then return True
  else if Label < Tree.Root
    then Search (Tree.left,Label)
  else Search (Tree.Right,Label)
  else return False;

Insert : SUBROUTINE ;

PARAMETER
  INPUT Label : RootType ;
  INOUT Tree : BinaryTree (RootType) ;

BEHAVIOUR

  if EXIST (Tree)
  then if Label < Tree.Root
    then Insert (Label, Tree.Left)
  else Insert (Label, Tree.Right)
else begin
    call RISE (Tree);
    set Tree. Root = Label
end;
END ;

DSL includes the following categories of statements: conditional branch (IF-THEN-ELSE and CASE), iteration (WHILE-DO, REPEAT-UNTIL, FOR, LOOP), compound statement (BEGIN-END), assignment, input/output and subroutine call/return. The semantics of the sequential statements is similar to that defined for the block structured programming languages. The process synchronization statements are REQUEST, ACCEPT, SELECT and ACTIVATE. Besides the above categories, any English sentence may be used as a valid DSL statement.

As SODA follows the top-down design approach, calls to subprograms that have not been defined yet are accepted but a warning message is sent to the user. Also, the declaration of entities whose types have not been defined are accepted. Whenever an entity definition is input to the design base the undefined entries in the base are analysed to see if they match with the new definition.

4. Integration with the Implementation Phase

Since SODA intends to provide an integration with the further phases of the software development process, the DSL processor makes an attempt to analyse fully the behaviour statement of software components. If the parsing succeeds the analysed expression is stored in a structured form which is more easily translated to a high level programming language than the design language. If either there are operands that have not been declared, or any incompatibility exists then the processor stores the whole expression as a string of characters.

The integration of SODA designs with the implementation phase can be realized by translating the design specifications stored in the Design Specs Base to a programming language. Specifications are stored in the DSB as structured information rather than text. The internal representation is defined in terms of software elements and control structures and so it is independent of the design language syntax. The idea was to represent internally the control structures proposed by the structures programming methodology so that the design could be translated easily to the popular block structured programming languages. The statements that do not follow DSL's syntax are output as comments and left to be refined by the programmer.
5. SODA Library

SODA provides a library mechanism for storing abstract data types and algorithms that can be instantiated as components into the software being designed. The types being introduced in the Library may make reference to any of the types already defined. Thus, it is possible to define more complex types by extension of the ones already defined. Whenever a designer specifies an instantiation, the construct is copied from the Library and incorporated into the software being designed. The Library can be searched either automatically (whenever instantiations of the Library types are specified) or manually (the designer can ask SODA either to show the Library directory, a specific abstraction or to get a full copy of the Library itself).

6. SODA outputs

SODA provides the capability for asking the data base for information such as which modules call a certain module, or which modules use a given table of data. Besides producing a formatted listing of the entire software design (or parts of it), SODA outputs a number of analyses that intend to warn the designer of possible design problem areas. All documents can be obtained either on the screen or on hard copy.

The flow matrices, whose purpose is to show the control and data flows of a software design, constitute one of the SODA analyses. The Control Flow Matrix depicts the calling relations between modules of a software system. The purpose of the Data Flow Matrix is to show the relationship between modules and files/data bases. Therefore, looking at this matrix the designer can see which modules have access to a particular file or data base.

SODA can produce a complete cross-reference listing containing all symbols either defined or referenced in the design project. The Data Dictionary of a software project constitutes a listing of all data definitions specified at the highest level of the software design. The software symbol table, a listing of all symbols defined/referenced in the design project can also be obtained. An example of a formatted listing of a design specification, followed by its cross-reference, symbol table and control flow matrices is presented in Appendix A.

SODA includes an automatic design quality metric which can provide immediate feedback to the designer. The design evaluation method has been proposed by Yin and Winchester [Yin 79] and is based on McCabe's program complexity measure [McCabe 76]. Examining the measures of
design complexity the designer can identify levels at the design 
structure that are error-phone and with help from other SODA analyses he/ 
she can identify the modules that are causing the high complexity and 
restructure the software accordingly. An example of the design quality 
metric is shown in Appendix B.

6. Conclusion

A Pascal implementation of a SODA prototype on a VAX-11 computer 
system has been developed. Most of the SODA features described previously 
have been implemented by the prototype.

SODA has most requisite characteristics to be a successful 
software tool [Nassi 80];
- easy to learn and use; SODA's command language is small and fairly 
simple. Its design language requires no training for people familiar with 
conventional programming languages;
- good performance; the SODA prototype has shown very good response time;
- easily modifiable; the prototype has been developed according to the 
principles of modularity, information hiding and data abstraction, which 
are recognized as successful approaches for the construction of software 
that is easy to understand and maintain;
- meets a clear need; the design process is known to be the most critical 
phase of the software development process, incurring heavy costs on the 
final product. SODA is addressed to the software designer and his pro-
blem, providing functions for validating design specifications, producing 
design documentation and giving design analysis feedback.

The SODA prototype can also be used for educational purposes as 
a software design laboratory. Software engineering students can make use 
of it when developing their case study applications as well as for 
implementing new algorithms of software cost estimation, design analysis 
and design quality evaluation.

7. References

[Boehm 80] B.W. Boehm, Software Engineering - As it is, in Software 

in Tutorial on Software Design Techniques, P. Freeman and 


MODULE SODA : PROJECT ;

CALL
Show <!TO BE DEFINED!>, Parser, Update <!TO BE DEFINE!> ;

TYPE
inicuffer : ADT ;

DATATYPE = RECORD
    line : STRING ;
    index : NUMBER ;
END ;

OPERATION
Getchar : SUBROUTINE ;

PARAMETER
OUTPUT
    symbol : CHAR ;
INPUT
    buffer : inBuffer ;

DATA
    Terminal : FILE OF STRING ;

BEHAVIOUR
BEGIN
    IF end_of_buffer <!TO BE DEFINED!> THEN
    BEGIN
            READ inoutline <!TO BE DEFINED!> FROM Terminal ;
            Initiate index <!TO BE DEFINED!> ;
    END ;
            Get symbol pointed by index <!TO BE DEFINED!> ;
    END ;

DATA
    Terminal : FILE OF STRING ;
    command : STRING ;
    inBuffer : inBuffer ;

BEHAVIOUR
REPEAT
    READ command FROM Terminal ;
    IF command = "input" THEN
        CALL Parser ELSE
        IF command = "update" THEN
            CALL Update ELSE
            IF command = "display" THEN
                CALL Show ;
            END ;
            UNTIL command = "exit" ;

SUBCOMPONENT
    Parser : SUBROUTINE ;

MODULE Parser : SUBROUTINE ;

CALL
    Scanner ;
MODULE Scanner : SUBROUTINE;

PARAMETER
  OUTPUT
       token : STRING;
       code : NUMBER;

CALL
  inBuffer . Getchar;

DATA
  blank : CHAR;
  symbol : CHAR;
  letter : SET OF CHAR;
  digit : SET OF CHAR;
  delimiter : SET OF CHAR;

BEHAVIOUR
10 BEGIN
20  CALL inBuffer . Getchar;
30  WHILE symbol = blank DO
40    CALL inBuffer . Getchar;
50  I= symbol in digit THEN
60    BEGIN
70      SET code = 3;
80      WHILE symbol in digit DO
90        BEGIN
100        append symbol TO token << TO BE DEFINED >>;
110        CALL inBuffer . Getchar;
120      END;
130  END;
140  I= symbol in letter THEN
150    BEGIN
160      SET code = 1;
170      WHILE symbol in ( letter + digit ) DO
180        BEGIN
190          append symbol TO token << TO BE DEFINED >>;
200        CALL inBuffer . Getchar;
210      END;
220    END;
230  END;
240 END;
### Cross Reference List

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>CLASS</th>
<th>DEFINED AT</th>
<th>REFERENCED AT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>UNDEFINED</td>
<td>SODA</td>
<td>SODA</td>
</tr>
<tr>
<td>Show</td>
<td>UNDEFINED</td>
<td>SODA</td>
<td>SODA</td>
</tr>
<tr>
<td>Terminal</td>
<td>DATA</td>
<td>SODA</td>
<td>SODA</td>
</tr>
<tr>
<td>command</td>
<td>DATA</td>
<td>SODA</td>
<td>SODA</td>
</tr>
<tr>
<td>inBuffer</td>
<td>TYPE</td>
<td>SODA</td>
<td>Scanner SODA</td>
</tr>
<tr>
<td>Getchar</td>
<td>OPERATION inBuffer</td>
<td>inBuffer</td>
<td>Scanner inBuffer</td>
</tr>
<tr>
<td>line</td>
<td>RecordField inBuffer</td>
<td>inBuffer</td>
<td>NO REFERENCES</td>
</tr>
<tr>
<td>index</td>
<td>RecordField inBuffer</td>
<td>inBuffer</td>
<td>NO REFERENCES</td>
</tr>
<tr>
<td>inoutBuffer</td>
<td>DATA</td>
<td>SODA</td>
<td>NO REFERENCES</td>
</tr>
<tr>
<td>Update</td>
<td>UNDEFINED</td>
<td>SODA</td>
<td>SODA</td>
</tr>
<tr>
<td>Show</td>
<td>UNDEFINED</td>
<td>SODA</td>
<td>SODA</td>
</tr>
<tr>
<td>Parser</td>
<td>MODULE</td>
<td>SODA</td>
<td>SODA</td>
</tr>
<tr>
<td>Scanner</td>
<td>MODULE</td>
<td>Parser</td>
<td>Parser</td>
</tr>
<tr>
<td>token</td>
<td>PARAMETER</td>
<td>Scanner</td>
<td>NO REFERENCES</td>
</tr>
<tr>
<td>code</td>
<td>PARAMETER</td>
<td>Scanner</td>
<td>Scanner</td>
</tr>
<tr>
<td>symbol</td>
<td>DATA</td>
<td>Scanner</td>
<td>Scanner</td>
</tr>
<tr>
<td>blank</td>
<td>DATA</td>
<td>Scanner</td>
<td>Scanner</td>
</tr>
<tr>
<td>letter</td>
<td>DATA</td>
<td>Scanner</td>
<td>Scanner</td>
</tr>
<tr>
<td>digit</td>
<td>DATA</td>
<td>Scanner</td>
<td>Scanner</td>
</tr>
<tr>
<td>delimiter</td>
<td>DATA</td>
<td>Scanner</td>
<td>NO REFERENCES</td>
</tr>
</tbody>
</table>

### Symbol Table

<table>
<thead>
<tr>
<th>SCOPE</th>
<th>CLASS</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>UNDEFINED</td>
<td></td>
</tr>
<tr>
<td>Show</td>
<td>UNDEFINED</td>
<td></td>
</tr>
<tr>
<td>Terminal</td>
<td>DATA</td>
<td>FILE</td>
</tr>
<tr>
<td>command</td>
<td>DATA</td>
<td>STRING</td>
</tr>
<tr>
<td>inBuffer</td>
<td>TYPE</td>
<td>ADT</td>
</tr>
<tr>
<td>Getchar</td>
<td>OPERATION</td>
<td>SUPROUTINE</td>
</tr>
<tr>
<td>line</td>
<td>RecordField STRING</td>
<td></td>
</tr>
<tr>
<td>index</td>
<td>RecordField</td>
<td>NUMBER</td>
</tr>
<tr>
<td>inoutBuffer</td>
<td>DATA</td>
<td>inBuffer</td>
</tr>
<tr>
<td>Update</td>
<td>UNDEFINED</td>
<td></td>
</tr>
<tr>
<td>Show</td>
<td>UNDEFINED</td>
<td></td>
</tr>
<tr>
<td>Parser</td>
<td>MODULE</td>
<td>SUPROUTINE</td>
</tr>
</tbody>
</table>

### Scanner Table

<table>
<thead>
<tr>
<th>SCOPE</th>
<th>CLASS</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>token</td>
<td>PARAMETER</td>
<td>STRING</td>
</tr>
<tr>
<td>code</td>
<td>PARAMETER</td>
<td>NUMBER</td>
</tr>
<tr>
<td>symbol</td>
<td>DATA</td>
<td>CHAR</td>
</tr>
<tr>
<td>blank</td>
<td>DATA</td>
<td>CHAR</td>
</tr>
<tr>
<td>letter</td>
<td>DATA</td>
<td>CHAR</td>
</tr>
<tr>
<td>digit</td>
<td>DATA</td>
<td>SET</td>
</tr>
<tr>
<td>delimiter</td>
<td>DATA</td>
<td>SET</td>
</tr>
</tbody>
</table>
Appendix B

The design complexity evaluation method works on design structure charts as the one shown below. The rectangular and convex boxes represent processing modules and data base tables respectively, while the arrows represent control and data transfer. The method includes three functions which are computed for each level of a design tree structure:

- C, the cyclomatic number, which measures the network complexity. A sharp increase of C from one level to the next indicates an extreme complexity increase in the software system;

- R and D, which measures the tree-impurity of software system; i.e., the deviation of a design tree structure from an ordinary graph. R measures the tree-impurity of level i against level Ø, and D measures that of level i against level i-1.

The graph below shows the complexity measures computed for a large project (involving approximately 15 levels of design structure hierarchy).