A PROTOCOL FOR EFFICIENT MAN-MACHINE INTERFACES
(also Research Thesis)

by

Jacob Ukelson

Technical Report #556

May 1989
A Protocol for Efficient Man-Machine Interfaces

RESEARCH THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF SCIENCE

Jacob Ukelson

SUBMITTED TO THE SENATE OF THE TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY
Adar, 5749
HAIFA
February, 1989
Contents

1.0 Introduction 1
1.1 Dialogue Definition Languages 2
1.2 Dialogue Management 5
1.3 Aim of the Thesis 6
1.4 Reader's Guide 8

2.0 A User Description of the System 11
2.1 Introduction 11
2.2 Dialogue Definition Language 11
   2.2.1 An Example Dialogue Definition 11
   2.2.2 Formal Definition of Dialogues 14
2.3 The Screen Manager 20
   2.3.1 Error Screens 22
2.4 Using the System 23

3.0 System Design Issues 25
3.1 Introduction 25
3.2 Reader's Guide 27
3.3 Selecting Objects for Display from the Dialogue Structure Graph 29
   3.3.1 The Dialogue Management Problem 33
   3.3.2 The Monotonic Distance Approach 37
   3.3.3 An Algorithm to Solve MDDMPs 38
   3.3.4 Example of the MDDMP Algorithm 43
   3.3.5 Navigation and User Input in the Dialogue Management Graph 43
   3.3.6 Extending the Dialogue Management Graph 45
   3.3.7 Displaying the Display Graph 47
3.4 Rule Execution 48
   3.4.1 The Constant Propagation/Partial Evaluation Problem 48
   3.4.2 Loop Execution and Constant Propagation 51
   3.4.3 Extending the Dialogue Management Algorithm for Rules 61
3.5 Constraints 62
3.6 Handling User Errors 63
   3.6.1 Violation of Selection Criteria 63
   3.6.2 Violation of Semantic Constraints 64
3.7 Extended Example 66
   3.7.1 Actual Screens 72

4.0 Conclusions and Further Research 79

5.0 Bibliography 83
Figures

1. Secheim model of user interface 5
2. Dialogue structure graph defined by objects of the example of Section 2.2.1 18
3. Example screen for dialogue definition of Section 2.2.1 21
4. Example screen with continuation objects 22
5. Example of an error screen for the dialogue definition in Section 2.2.1 23
6. A simplified version of the dialogue structure graph of Section 2.2.2 30
7. An example of a directed acyclic graph with no $S$ for $m = 12$ 35
8. $S$ of size 11 or less 35
9. After addition of vertex 1 36
10. After addition of vertex c 36
11. After addition of vertex m 37
12. Monotonic distance dialogue management algorithm 38
13. A dialogue management graph (an extension of Figure 6) 45
14. An example of a single choice menu for graph of Figure 13 46
15. Constant propagation lattice 52
16. Meet rules 52
17. An example program for the incremental ConditionalConstant algorithm 56
18. Program Flow graph for the example program 57
19. DefUse graph for the example program 58
20. Hierarchical DefUse graph for the example program - top level 58
21. Hierarchical Program-Flow graph of the example program - top level 58
22. Hierarchical DefUse graph detailing loop1 of the example program 59
23. Hierarchical Program-Flow graph detailing loop1 of the example program 59
24. Hierarchical DefUse graph detailing loop2 of the example program 60
25. Hierarchical Program-Flow graph detailing loop2 of the example program 60
26. Initial screen for the dialogue of the example of Section 3.7 72
27. The user moves mouse in order to select field 73
28. User's selection of a field 73
29. Next screen after user selection of a the continuation field 74
30. The user selects a value from the enumerated range 74
31. Rule 1 is used to deduce that Paid Preparer is no longer needed 75
32. After the user requests to see the context (Up button) 75
33. The user selected the field and moves the mouse in order to continue the dialogue 76
34. Fields under the new context 76
35. The user enters his wages and chooses to enter dividends 77
36. The user enters dividends and requests to return to the defining context 77
37. Total dividends has been automatically computed by rule 2 78
38. The user returns to the defining context 78
Abstract

This thesis investigates the division of interactive systems into application and dialogue components. The focal point of our research was the design of a dialogue manager for large dialogues so as to bother the user as little as possible. A framework for defining precisely what is meant by bother is used to propose the formal definition of the term efficient dialogue. An efficient dialogue is defined as a dialogue which displays a large amount of information on the screen while keeping the information clear and in context, and by abstaining from redundant questions. The model describes the division of a man-machine dialogue manager into several subtasks:

1. dialogue definition language
2. application to dialogue manager interface
3. dialogue manager
   a. dialogue logic
   b. screen manager

The application to dialogue manager interface can actually be considered as a compilation of the dialogue definition language which is then executed by the dialogue manager in order to obtain input from the user.

A prototype dialogue management system is constructed to implement these ideas. The dialogue designer uses the dialogue definition language to give a high-level description of the dialogue. From this description the interaction is automatically created. The dialogue manager has been written in C under the UNIX™ operating system. The presentation subsystem was implemented in C on a SUN™ workstation under the UNIX operating system.
1.0 Introduction

Designing and implementing a user interface of a new interactive application may require as much as 50% of the total development effort [SS77]. This enormous use of resources has generated much interest in tools and methods for the design and implementation of man-machine interfaces. Human factors research, hardware research and software research are the three main classes of man-machine interface (or dialogue management) research [SHN87]. This thesis is confined to the software aspects of man-machine interfaces, but since both software and human factors aspects are interrelated, human factors cannot be completely ignored and some assumptions about how a man-machine dialogue should look are implicitly made. These assumptions were based on common-sense and not on human factors research since that is beyond the scope of this thesis.

An example of a simple interactive application is a system which obtains from an employee his name and the name his spouse, and also information about his manager and the project he works on. After all of this information is acquired from the user, it is returned to the application for further processing, e.g. entering the information into the company's database.

This, or any interactive system, may be divided into interrelated application and dialogue components. In the above example, the application component is the database application which requires the information, while the dialogue component interacts with the user in order to obtain it from the user. This separation between the substance of computer applications and their man-machine interface has attracted a lot of software research. Common to many systems is the effort to allow easy definition of the dialogue while enabling the independent development of the application and dialogue components [GRE85], [COU85], [BASS85], [LH87]. To this end the application must communicate with the dialogue via a well-defined protocol.
This protocol is actually a formal dialogue definition language used to describe the interaction with the user. Since this thesis is concerned with the definition of large dialogues in which a large amount of interrelated information is required from the user, the number of possible screens becomes very large and screen selection becomes an important issue. The following survey of dialogue definition languages is based on this assumption.

1.1 Dialogue Definition Languages

The main approaches to dialogue definition described in the literature are graphs, formal languages, non-procedural languages, algebraic languages, and events.

1. Graphs

   a. Parnas [PAR69] used state graphs to describe dialogues. Every in-edge describes output displayed to the user and every out-edge describes input obtained from the user.

   b. Denert [DEN73] also suggested the use of state graphs but with the possibility of defining a hierarchy between graphs. A node of a state graph may represent a complete state graph. These graphs are called interaction diagrams. A box node represents a computation while a circle node represents an interaction point. The edges of the graphs are directed and marked. An in-edge denotes input to be obtained from the user or some change in the systems state.

   c. Ben-Bassat [BW84] suggested a system similar to Denert's. A compiler was developed which creates a prototype of the dialogue based on the dialogue state graph.

   Example:

   ![Graph Example]

This type of formal language explicitly shows the state of the system and the allowed input at any given time. However, this requires the dialogue designer to describe every possible dialogue state. The number of states tends to be very large for any but the most trivial dialogues. These graphs do not take into account the screen layout and that many items may be displayed and obtain values from the user at
once. Therefore, this type of definition is useful for small dialogues but becomes clumsy for large dialogues.

2. Formal Languages

a. Hanau [HL81] suggested using BNF to describe dialogues. They also created a compiler which creates a prototype of the dialogue based on a BNF description.
b. Shneiderman [SHN82] suggested using a multi-party BNF. Each non-terminal has an added token which describes the dialogue agent represented by this non-terminal. This extension offers a natural mechanism which differentiates between the different parties of a dialogue.
c. Guest [GUE82] suggested the use of production systems for dialogue description. This method is very similar to BNF.
d. Jakob [JAK83] compares the use of state graphs and formal languages for dialogue description. He asserts that state graphs are a more natural form of dialogue description because the graphs explicitly show the system state and the input required at any given time.

These types of definitions are just variation on graphs. It also requires the dialogue designer to describe every possible dialogue state but has the advantage of implicitly taking into account that a number of items that may be displayed and receive user input at once. Example:

```
main_screen  ----> <employee_name> <spouse_name>
<employee_name>  ----> <character_string>
<spouse_name>  ----> <character_string>
```

3. Non-Procedural Languages

Lafuente [LAF77] suggested the use of a non-procedural language based on PASCAL. He defines a frame as the basic unit of interaction with the user. A frame contains fields which must be completed by the user and logic relating the fields. The logic between fields in a frame is defined using a non-procedural language. The frame structure itself is rigid and defined by the dialogue designer, so here too every screen must be explicitly designed.

This thesis takes the use of non-procedural languages one step further. Instead of a rigid structure of frames, the dialogue is left as flexible as possible. The use of a non-procedural language has the advantages of allowing the dialogue to evolve based on user input, without the dialogue designer having to define all the possible dialogue states. This degree of freedom between the dialogue's definition and
the actual interaction with the user allows the system to attempt to optimize the system with respect to a specific user. The dialogue definition language presented here is based on rules and objects. The inter-rule control is similar to a non-procedural language, while the rule itself is defined using a procedural PASCAL-like language. This hybrid language allows rules to behave as in a non-procedural language, while classical compiler optimization techniques can be used in order to optimize the rules and customize them to the user's input. Objects are the intermediaries between the user and the system; the rules are used to define the dependencies between these objects.

4. Algebraic Languages
   a. Shaw [SHAW79] suggested the use of flow expressions to describe dialogues. These form an extension of regular expressions such that synchronization and interleaving of commands can be simply described.
   b. Chi [CHI84], [CHI85] also suggested using flow expressions augmented by an algebra to describe interaction. He defines an algebra of interaction objects and operations which describes a single interaction in order to extend flow expressions. This increases the expressive power as compared with simple flow expressions.

Flow expressions are simple and easy to understand but have limited expressive power. The addition of an algebra extends the expressive power but causes the definition of dialogues to be lengthy and obtuse. Example of a simplified algebraic definition:

```
get_string(employee_name);
get_string(spouse_name);
```

where `get_string` is some previously defined operation over the set `{employee_name, spouse_name}`.

5. Events
   Green [GRE85], [GRE86] describes an event-driven model of dialogue description. An interaction is described as a collection of events. Each event is handled by a specific event handler which is called to process the event whenever it occurs. This approach, though claimed to be new, is actually equivalent to the non-procedural approach. Below is an example of a dialogue that obtains an employee's name and then the name of his or her spouse:
Event handler get_info is
  Token keyboardstring s;

Var
  int state = 0;
  string employee_name, spouse_name;

Event Init
  print: "Employee Name";

Event s : string
  if (state == 0)
    employee_name = s;
    state = 1;
    print "Enter spouse name";
  else
    spouse_name = s;
    state = 0;
    send_to_application (employee_name,
                         spouse_name);

1.2 Dialogue Management

A dialogue designer uses some formal language to define the dialogue which may then be used to automatically create a dialogue. The dialogue component of an interactive application is handled by a user interface management system or dialogue manager.

The Seeheim model of user interfaces [GRE84] is a typical user interface management scheme. The dialogue manager is divided into a presentation component, dialogue control and an application interface model. This can be considered as a division of dialogues into their visual and logical aspects, where the presentation component defines the visual aspects and the logical aspects are defined by the dialogue control and application interface.

![Seeheim model of user interface](Figure 1. Seeheim model of user interface)
Most commercial user interface management systems stress the visual aspect and many advances have been made recently in this direction (e.g. windows, mice, touch screens) [ISPF], [TOP]. Visual aspects of a dialogue depend on human factors and are beyond the scope of this thesis, but many papers suggest features that should be available in good, user friendly dialogues [FIT79], [GRI79], [WS82], [NIE82], [SHN87].

1.3 Aim of the Thesis

This thesis investigates the division of interactive systems into their application and dialogue components. The focal point of the thesis is how to design a dialogue manager for large dialogues so as to bother the user as little as possible. A framework for defining precisely what is meant by bother is used to propose the formal definition of the term efficient dialogue. An efficient dialogue is defined as a dialogue which displays a large amount of information on the screen while keeping the information clear and in context, and by abstaining from redundant questions. The model describes the division of a man-machine dialogue manager into several subtasks:

1. dialogue definition language
2. application to dialogue manager interface
3. dialogue manager
   a. dialogue logic
   b. screen manager

The application to dialogue manager interface can actually be considered a compilation of the dialogue definition language which is then executed by the dialogue manager in order to obtain input from the user.

Most software research in dialogue management systems is concerned with the dialogue definition language, application to dialogue manager interface and the screen manager [MYE87]. The dialogue logic component is usually of less interest since, as can be seen in the previous section on dialogue definition languages, the dialogue's control is explicitly or implicitly contained in the dialogue definition. This thesis focuses on the dialogue logic subtask of the man-machine dialogue manager. The aim is to allow the dialogue designer
define the dialogue using a high-level dialogue definition language and have the dialogue manager
automatically create an efficient dialogue based on that definition. In order to achieve this aim, the dialogue
definition language proposed is a rule based language which allows the dialogue designer give a high-level
definition of the information required from the user. This definition is concise and clear, but still contains
enough information so that an efficient dialogue can be automatically created.

In order that a dialogue manager be able to minimize the interaction with the user, it must receive from the
application a sufficiently high-level specification describing the information elements required and their
interrelationships. Using a dialogue definition written in the language proposed here, the dialogue control
can decide on the order in which the questions will be presented to the user, as well as on the number of
different questions to be presented at once.

For the dialogue control component of the dialogue logic this thesis considers three aspects of efficient
dialogues:

1. Internal representation and selection of objects for display - only the structure of the dialogue's internal
   representation is used to select the number of items displayed on the screen and their context. The
   internal representation used is a directed acyclic graph as opposed the more usual tree representation.
   This denser representation allows better utilization of the display device.

2. Restructuring the dialogue representation based on the user input - shorten dialogue interaction with the
   user based on the relations between the objects in the dialogue domain. These relations are expressed as
   rules which describe computational dependencies between the objects required from the user. These
   dependencies allow the dialogue to be customized to a specific user and avoid redundant questions.
   Using partial evaluation techniques to execute these rules allows output to be obtained even if not all of
   the rule's input objects have been defined. This is in keeping with the incremental nature of
   man-machine dialogues.

3. Handling user errors - contradictions between answers given by the user at different times are
   indications of user errors. Most systems handle user error by a simple backtracking paradigm. This thesis
   suggests that in many cases people insist on their most recent answers and only rarely prefer old
   responses. Therefore, older values causing the problem should be displayed to the user first since they
   have a greater chance of being incorrect. This method also presents to the user a consistent error
correction mechanism, since as the user decides to keep older values they become more recent and the user steps through the values causing the problem from oldest to most recent.

The presentation component or screen manager is responsible for the actual display and for obtaining input from the user. Using an abstract description of the dialogue an actual physical dialogue is created. This element of the system is briefly touched on in this thesis, since the actual physical dialogue depends on the type of dialogue desired and the physical input devices available. In order to build a prototype of the system, some decisions concerning the screen manager were made. The decisions made were based on common-sense and not on human factors research which is beyond the scope of this thesis. Of course, other solutions are possible.

Recent advances in workstation technology provides the processing power needed to manage dialogues as outlined above. The division between main-frame and workstation seems to naturally support the division between applications and their associated dialogues. As a side benefit, applications become less dependent on the specific terminal type.

A prototype of the system which takes a formal definition of the dialogue and using the strategies outlined in this thesis, creates a dialogue prototype has been implemented. The dialogue manager has been written in C under the UNIX operating system. The screen manager has been also written in C under the UNIX operating system on a SUN workstation using SUNVIEW screen management facilities.

1.4 Reader’s Guide

Chapter 2 of this thesis consists of a users guide which describes the dialogue definition language and the screen display as it appears to the user. This section contains enough information to allow a dialogue designer to create an interactive dialogue and test it using the prototype.

Chapter 3 describes an abstract model of the system and discusses theoretical issues arising from this model. This section discusses the problems of creating an efficient dialogue based on the dialogue definition language. Algorithms and heuristics solving these problems are also presented.
Each chapter also contains its own detailed introduction.
2.0 A User Description of the System

2.1 Introduction

In this chapter, the dialogue definition language is defined. This language consists of declarations which define the objects of information whose values are to be obtained from the user and its context. Relations between the values of these objects are defined by rules in a PASCAL-like language. User input is restricted by constraints on the legal values for an object and on the number of objects available for input.

2.2 Dialogue Definition Language

2.2.1 An Example Dialogue Definition

The following example is annotated in order to give the reader a preliminary feel for how dialogues are defined. Explanations are surrounded by /* */ markings for readability.

GOAL: (Name Spouse_Info Project_Info);
TEXT: "Project Example";
/*This defines the set of objects required from the user. */
/*The text below the GOAL definition defines the text displayed */
/*on the screen. The dialogue is completed if all the objects */
/*contained in the GOAL list are completed. */
OBJECT: Name
/*The object name and the start of an object definition. */
TYPE : string;
/*The object type. This object is used to contain input obtained */
/*from the user. The input to be obtained is a string of characters.*/
TEXT : "Enter name";
/*This is the text associated with the object and displayed to the */
/*user describing the input required for this object. */

OBJECT: MStatus
TYPE : enumerated;
/*This object is enumerated. The input for this object be only one */
/*of the values listed in the range below. */
RANGE : (married, single);
TEXT : "Enter marital status";

OBJECT: Spouse_Info
TYPE : list;
/*The list type defines an object as an internal object which */
/*is used only to to define a hierarchy between other objects. The */
/*object must also have a VALUE field defining the relation between */
/*the objects offspring. */
VALUE : (ALLOC MStatus Name);
/*This field has meaning only for internal objects and defines */
/*a hierarchy between objects. This object has two offspring (Name */
/*and MStatus) and the ALLOC combining operator means that all of */
/*the enabled offspring must be completed in order for this object */
/*to be considered completed. */
TEXT : "Marital Information";

OBJECT: Project_Question
TYPE : enumerated;
RANGE : (yes, no);
TEXT : "Do you work on a project?"

OBJECT: Manager_Question
TYPE : enumerated;
RANGE : (yes, no)
TEXT : "Do you report to a manager?"
OBJECT: Manager_Info
TYPE : list;
VALUE : (ALLOF Manager_Question Name);
TEXT  : "Manager Information";

OBJECT: Project_Info
TYPE : list;
VALUE : (ALLOF Project_Question Manager_Info);
TEXT  : "Project Information";

RULES
/*This is the RULE definition section. It consists of two rules. */

RULE: BEGIN
/*This rule states that if the user has stated that he does not */
/*work on a project there is no use asking him for more information */
/*on his manager (the objects under the Manager_Info object). To */
/*address an object, its fully qualified name is given. If the */
/*objects simple name is unique, then the simple name can be given */
/*instead of the fully qualified name as shorthand. */
  IF Project_Info.Project.value = "no" THEN
    Project_Info.Manager_Info.enabled := DISABLED;
  END
  Project_Info.Manager_Info.enabled := ENABLED;
END.

RULE: BEGIN
    IF Manager_Info.Manager.value = "no" THEN
      Manager_Info.Name.enabled := DISABLED;
    END
    Manager_Info.Name.enabled := ENABLED;
END.

RULE: BEGIN
/*This rule states that if the user has stated that he is single */
/*there is no use asking him for his spouses name. */
    IF Spouse_Info.MStatus.value = "single" THEN
      Spouse_Info.Name.enabled := DISABLED;
    ELSE
      Spouse_Info.Name.enabled := ENABLED;
    END.
CONSTRANTS

/*This is the CONSTRANTS definition section. It consists of one */
/*constraint. This constraint states that the value for the Name of */
/*employee cannot be the same as his managers. If this is true for */
/*the input obtained from the user, his input is erroneous. The user*/
/*is notified that there is an error, and the text of the constraint*/
/*is displayed along with the oldest input causing the contradiction*/
/*so that the user understands the reason for the error and can */
/*correct it. */

CONSTRAINT : Name.value <> Project_Info.Manager_Info.Name.value;
TEXT: "You cannot be your own manager";

/*This example does not contain all possible language constructs, */
/*but is detailed enough to be instructive. A more complete (and */
/*and complex) example can be found in Section 3.7. */

2.2.2 Formal Definition of Dialogues

A dialogue definition consists of four sections:

1. Goal - describes a tuple of information to be obtained from the user.
2. Declaration of objects - describes the attributes of the objects and the hierarchy between them.
3. Rules - define computational relations between objects in the dialogue domain.
4. Constraints - define passive relations between objects in the dialogue domain.

2.2.2.1 Goal

The goal object consists of a list of objects required by the application. The dialogue manager interacts with
the user in order to instantiate the objects contained in the goal definition. The formal definition of a goal
object is given in the following section.
2.2.2.2 Declaration of Objects

Objects are the intermediaries between the application and the dialogue manager, and between the dialogue manager and the user. Objects have various properties describing the structure of the information contained in each of them. These properties are initially defined by the application and may be altered by the rules during the interaction with the user.

2.2.2.3 Formal Definition of Objects

<declarations> ::= OBJECTS <goal_declaration> <object_declarations>
<goal_declaration> ::= GOAL : ( <object_list> ) ; <text_def> ;
<object_list> ::= <identifier> | <object_list> <identifier>
<object_declarations> ::= <object_declarations> <object_declaration> |
<object_declaration> ::= OBJECT : <object_name> <property_list>
<object_name> ::= <identifier>

<property_list> ::= <type_def> <range_def> <format_def>
    <default_def> <status_def> <value_def>
    <text_def>

<type_def> ::= TYPE : <type> ;
<type> ::= STRING | INTEGER | LIST | ENUMERATED

<range_def> ::= RANGE : <range> ; | ε
<range> ::= < subrange_type> | <enumerated_type>
<enumerated_type> ::= ( <enumerated_list> )
<enumerated_list> ::= <identifier> | <enumerated_list> <identifier>
<subrange_type> ::= <integer> .. <integer> |
    <real> .. <real>

<format_def> ::= FORMAT : <format> ; | ε
<format> ::= <format> A (<integer>) | <format> D (<integer>) | ε

<default_def> ::= DEFAULT : <value> ; | ε
<status_def> ::= STATUS : <status> ; | ε
<status> ::= <status> | <one_status> ; |
<one_status> ::= SELECTED ::= <selected_value> |
    ENABLED ::= <enabled_value>
    DISPLAY ::= <display_value>
    ASSIST ::= <assist_value>
<selected_value> ::= SELECTED | NOT_SELECTED | DEDUCED
<enabled_value> ::= ENABLED | DISABLED
<display_value> ::= NORMAL | UNDERLINE | HIGHLIGHT | INVISIBLE | SHOW |
<other_possible_displays>
<assist_value> ::= ASSIST | NOASSIST

<value_def> ::= VALUE : (<ellipsis> <operator> <object_list> ) ; | ε
<ellipsis> ::= ... | ε
<operator> ::= ALLOP | NONEOF | EXACTLY ( <integer> ) |
    ATMOST ( <integer> ) | ATLEAST ( <integer> )
<text_def> ::= TEXT : <string> ; | ε
1. `<type>` - STRING and INTEGER were chosen as examples of simple types. These were arbitrary choices for the prototype and of course can easily be augmented by other simple types (i.e. reals, exponents). The other two types have special meanings within the dialogue definition:
   a. LIST - a conglomeration of elsewhere defined objects. The exact description of the objects to be conglomerated and their combining operator is contained in the VALUE property of a list object. This defines the object as an internal object which receives no user input. All the other types define the object as an external object which is used to obtain input from the user.
   b. ENUMERATED - an enumerated type described by the RANGE property.

2. RANGE - describes the set of values for an ENUMERATED object or a range of values for a simple object.

3. FORMAT
   a. A ( `<integer>` ) - specifies the number of characters the input should contain at that point.
   b. D ( `<integer>` ) - specifies the number of digits the input should contain at that point.

4. DEFAULT : `<value>` - contains the default value of the object. This property may be defined only for objects with no explicit value property (external objects).

5. STATUS - the status properties. The default values for each of the alternatives are shown below in bold face. Only variations from the default values need be specified.
   a. SELECTED := SELECTED/NOTSELECTED/DEDUCED - indicates whether the object has been selected (instantiated) by the user for use or if the value of the object has been computed (deduced) from other values using the rules.
   b. ENABLED := ENABLED/DISABLED - indicates whether the object is available for input from the user or to be completely ignored.
   c. DISPLAY := NORMAL/UNDERLINE/HIGHLIGHT/INVISIBLE/SHOW - describes the display type of the object. Here too, the choice of the display types were arbitrary and selected only to demonstrate some of the display types possible: SHOW defines the object as an output only object which cannot receive input from the user.
   d. ASSIST := ASSIST/NO_ASSIST - indicates whether the user should be assisted in finding a correct value for the object. Assistance consists of enumerating for the user the legal values for the object or instantiating the object with the legal value if only one such value exists.
6. **VALUE**: \(<\text{ellipsis}\> \text{<operator>} \text{<object_list}>\) - defines the object as an *internal object*. The \text{<operator>} defines a relation between objects in the \text{<object_list)}. This property applies only to objects of type LIST, which are internal objects. Internal objects do not define a value to be obtained, but a logical grouping and hierarchy between objects. An internal object is considered *completed* when its \text{<operator>} is satisfied by the ENABLED objects in the \text{<object_list}>. Other objects, i.e. external objects, define a value to be obtained for the application. An external object is considered selected and completed when it has received a legal value from the user or its value may be deduced from other objects. An external object may be the descendent of more than one internal object and therefore, when being instantiated, must be identified by its *fully qualified name*. An object's fully qualified name must uniquely specify its ancestors as defined in the objects VALUE property. So, if there is more than one possible ancestor for an object or any of its ancestors, the object's fully qualified name is a linear reconstruction of the object's ancestors needed to uniquely identify the object. The objects in this linear construction are separated by periods. Any object that has more than one possible fully qualified name can actually be considered a set of distinct objects of the same \text{<object_name>} but with different fully qualified names. An ellipsis indicates that the object's descendents may be repeated (i.e. selected more than once - any descendents which is an external object may receive more than one value) as many times as the user requires. Each copy has a fully qualified name which includes the number of the object's copy, which is an integer in the range 1..\text{NUMBER_OF_COPIES(object_name)}, where \text{NUMBER_OF_COPIES} is a predefined function on ellipsis objects. There is another predefined function for ellipsis objects - \text{CURRENT(object_name)} which gives the copy number of the value that the user has altered. An example of a fully qualified name is: \text{A.B.C[3]} where A, B are internal objects, C is an ellipsis object and 3 represents the 3rd copy of the external object C.

The possible combining operators for internal vertices are:

a. **exactly(x)** - satisfied if exactly \(x\) of the enabled objects in the list have been completed.

b. **atmost(x)** - satisfied if at most \(x\) of the enabled objects in the list have been completed.

c. **atleast(x)** - satisfied if at least \(x\) of the enabled objects in the list have been completed.

d. **all_of** - shorthand for **exactly(x)** where \(x\) is the number of enabled objects in the list.

e. **none_of** - shorthand for **atmost(0)**.
It may be useful to view the objects as a hierarchical acyclic structure where internal objects describe their successors in a summarized form. The descendants of a internal object are the objects comprising its list, ordered according to their relative position in the list. An external object has no descendants. One additional vertex of the graph is the goal itself which defines the root of the hierarchy. This graph will be called the dialogue structure graph.

![Dialogue structure graph](image)

**Figure 2.** Dialogue structure graph defined by objects of the example of Section 2.2.1

7. **TEXT**: `<string>` - contains the text of the object's prompt. This is the message displayed to the user in order to obtain a value for the object. A LIST object may have no TEXT property. This situation stems from the fact that the object is not displayed, but is only used to define a logical grouping of objects.

**2.2.2.4 Rules**

Rules define relations between objects. The rules describe how to deduce information about certain objects from information obtained about other objects. These rules are written in a PASCAL-like notation. A rule can define its own local variables and can also access the VALUE and STATUS properties of objects.

**2.2.2.5 Formal Definition of Rules**

The language used to define rules is a simple programming language which incorporates assignments, conditions and loops. This language was chosen for use since it is simple enough so that data-dependencies can be obtained in a simple syntax-directed manner as described in [HPR87] and yet is complex enough that it is of interest for data-flow analysis [KKLPM81]. This type of procedural language allows classical
compiler optimization techniques to be applied in order to customize a rule to partial input obtained from the user. This includes techniques such as constant propagation and dead-code elimination [ASU85].

\[
<\text{rule}> ::= \text{RULE : <program>}
\]
\[
<\text{program}> ::= <\text{variables}> <\text{constants}> <\text{rest\_of\_program}>
\]
\[
<\text{variables}> ::= \text{VAR <var\_list> | \epsilon}
\]
\[
<\text{var\_list}> ::= <\text{var\_list}> <\text{single\_var}> | <\text{single\_var}>
\]
\[
<\text{single\_var}> ::= <\text{identifier}> : <\text{var\_type}> ;
\]
\[
<\text{var\_type}> ::= \text{INTEGER | STRING}
\]
\[
<\text{constants}> ::= \text{CONST <const\_list> | \epsilon}
\]
\[
<\text{const\_list}> ::= <\text{const\_list}> <\text{single\_const}> ; | <\text{single\_const}> ;
\]
\[
<\text{single\_const}> ::= <\text{identifier}> = <\text{string}> | <\text{identifier}> = <\text{integer}>
\]
\[
<\text{rest\_of\_program}> ::= \text{BEGIN <statements> END.}
\]
\[
<\text{statements}> ::= \text{BEGIN <statements> <statement> END; | <statement> ;}
\]
\[
<\text{statement}> ::= \text{IF <condition> THEN <statements> ELSE <statements> | IF <condition> THEN <statements> | WHILE <condition> DO <statements> | <identifier> ::= <expression>
\]
\[
<\text{condition}> ::= <\text{expression}> <\text{comparison\_operator}> <\text{expression}>
\]
\[
<\text{expression}> ::= <\text{expression}> <\text{expression\_operator}> <\text{identifier}> <\text{identifier}>
\]
\[
<\text{expression\_operator}> ::= + | - | * | / | \text{AND} | \text{OR}
\]
\[
<\text{condition\_operator}> ::= = | < | > | <>
\]

The types of variables available in the language are simple variables (i.e. integer), strings (i.e. character arrays of an undetermined length), one dimensional arrays and dialogue objects. For the purpose of the discussion here, it is sufficient to consider a dialogue object property as another variable type which is used to obtain input or to define output for a rule.

2.2.2.6 Constraints

The constraint section of the dialogue definition language defines a set of predicates over the object values that must not be contradicted when the user wishes to complete the dialogue and send a packet to the application.
Constraints are not active, they cannot cause the value of an object to be changed. They are used to constrain the possible values of an external object and notify the user of problems if they are violated by user input. The constraints are checked whenever the user decides to enter input. If a constraint is violated, an error condition exists and the user must be notified of the problem. The user may legally terminate a dialogue only when no constraint is contradicted and all the requested information has been obtained. Obviously, a dialogue may be aborted at any time.

2.2.2.7 Formal Definition of Constraints

\[
\text{<constraint> ::= <constraints> <constraint>}
\]
\[
\text{<constraints> ::= <constraint> OR <one_constraint> |}
\]
\[
\text{<one_constraint> ::= CONSTRAINT: <object_exp> <constraint_operator> <identifier> ;}
\]
\[
\text{TEXT: <string> ;}
\]
\[
\text{<object_exp> ::= <object_exp> <exp_operator> <identifier> |}
\]
\[
\text{<identifier>}
\]
\[
\text{<exp_operator> ::= + | - | * | /}
\]
\[
\text{<constraint_operator> ::= < | > | = | <>}
\]

2.3 The Screen Manager

The screen manager corresponds to the presentation component of the Secheim model. The dialogue control selects the set of objects from the dialogue structure graph for display. These objects are then displayed on the screen via the screen manager, which also takes care of the physical interaction with the user.

In order to build a prototype of the system, some decisions concerning the screen manager were made. These decisions were based on common-sense and not on human factors research which is beyond the scope of this thesis.

Once a dialogue has been defined using the dialogue definition language of Section 2.2, the user only has to run the prototype in order to create a working dialogue. The command receives the file name which contains the dialogue definition and number of items allowed on a single screen as input. The actual command has other optional parameters which are described in Section 2.4. The user can then interact with the dialogue and supply the requested information.
The screen manager must display both the dialogue objects, which are abstractly represented as the vertices of the dialogue structure graph, and the interrelationships between the objects, which are abstractly modeled by the graph edges. The context of a vertex is displayed by indenting the vertices, based on the number of ancestors for the object that are displayed on the screen, and drawing arrows. These auxiliary arrows show the context of each object by pointing from an ancestor to its descendants. The user uses these arrows to find an external objects fully qualified name in order to enter a value into the object. In order to allow the user to select the fully qualified name of a displayed vertex, either for data entry or navigation, the screen manager presented here allows the user to select internal vertices by means of a mouse click. Once an internal object is selected, all of its descendents are marked by highlighting the arrow originating from the internal object. This gives the user feedback as to which objects are under the chosen context.

At any time during the dialogue step the user may enter input into a field, abort the dialogue with the *quit* button, attempt to send the information back to the application with the *send* button or continue the dialogue with the *continue* button. The *refresh* button allows the user to display only the object which still require values by removing from the screen all the instantiated objects and displaying only the unanswered objects.

An example screen:

![Diagram](image)

**Figure 3. Example screen for dialogue definition of Section 2.2.1**

The fields on the screen marked by boxes are user selectable so that they can be selected to define a context. The *Enter Marital Status* field is an example of an external vertex which is defined by a range - married and single. The *Enter Name* is another external object used to obtain information from the user.
Since a dialogue may not fit on a single screen, an object displayed on the screen may define a context for objects that are not currently displayed. These continuation objects are designated by a more... field on the screen. The selection of a continuation object causes the successors of the object to become available for user selection. The user can press the "up" button at any time to return to the context defining the current screen, if such a context exists.

Internal objects may also have associated selection criteria as defined in the value field of an object. These criteria are displayed along with the objects text so as to inform the user of how many objects should be selected.

![Diagram](image)

Figure 4. Example screen with continuation objects

### 2.3.1 Error Screens

The user may cause an error by violating a constraint or a combining operator. When this happens an error screen is initiated. The user is notified in the upper left corner of the screen that an error screen has been initiated. The text associated with the violated constraint is also presented. The error screen displays the oldest objects which cause the discrepancy. The user may then decide to change the values of the objects displayed or continue the dialogue and ignore the problem for a while.
Figure 5. Example of an error screen for the dialogue definition in Section 2.2.1

The values obtained from the user may be sent back to the application by using the send button only if the goal is complete and no constraints are violated.

2.4 Using the System

Once a dialogue definition has been created and placed in a file, the system may be run by the dialogue command. The command is defined as follows:

dialogue -i INPUT_FILE -o OUTPUT_FILE -noopt -s SELECT_ALGO SCREEN_SIZE

Parameters:

1. INPUT_FILE - Defines the file containing the dialogue definition. Defaults to ex.
2. OUTPUT_FILE - Defines the output file for syntax errors discovered in the dialogue definition. Defaults to parser.out.
3. noopt - rules should be run without any optimizations. Speeds the dialogue response time, but superfluous questions may be asked.
4. SELECT_ALGO - specifies the selection algorithm used to place objects on the screen. The possible choices are:
   a. DMS - dialogue management algorithm.
   b. DFS - depth first search algorithm.
   c. BFS - breadth first search algorithm.
The default is DMS. These different algorithms are discussed in Section 3.3.

5. SCREEN_SIZE - A bound on the number of objects to be displayed on a single screen. The default is 8.

Example:

dialogue -i ex2 6

The dialogue defined in the file ex2 is to be displayed with at most 6 objects on every screen.
3.0 System Design Issues

3.1 Introduction

This section deals with the issues of the abstract model of the system presented. The emphasis is placed on efficient dialogues. The issues discussed are internal representation and selection of objects for display, restructuring of the dialogue representation based user input and handling user errors. The internal representation of the dialogue objects is the *dialogue structure graph* which was presented in Section 2.2.2.

Selection of Objects for Display

The first aspect of efficiency is based on only the structure of the dialogues internal representation and concerns the number of items displayed on the screen and their context. A dialogue is considered a set of questions to be answered by the user. These questions have a certain context which should be kept in order to make user orientation easier. Context means that the user is made aware of where s/he has been, where s/he is and where s/he can go at anytime [FIT79]. In many cases the amount of information required from the user may be overwhelming, e.g. for tax return forms, and a dialogue manager should select a subset of the objects defined by the dialogue designer for presentation to the user. The optimal size of this subset may vary for different physical devices, dialogue types and user proficiency levels. Imposing a tree-like hierarchical organization on the objects offers ease of orientation and navigation through the set of objects, together with the ability to present summary information, i.e. context, to the user. However, a denser representation can be obtained by using a directed acyclic graph instead of a tree. This denser representation allows more information to be presented to the user at one time. This thesis presents an exact formulation of this object selection problem along with an efficient selection algorithm which is preferable to the usual depth first or breadth first search solutions usual for such problems. This selection mechanism keeps the
user in context while allowing the dialogue designer to describe compact dialogues in order to maximize the amount of information on a single screen.

Restructuring the Dialogue Representation

The second aspect of efficiency is based on the relations between the objects in the dialogue domain. These relations are expressed as rules which describe dependencies between the objects required from the user. The rules can help avoid redundant questions, and obtain new information based on previously obtained user information. For example, if during an interaction with the user, the dialogue manager ascertains that an answer to question $B$ is computable from an answer to question $A$, then as soon as $A$ is obtained it will deduce the value of $B$ from $A$.

The rules are defined by the dialogue designer using the simple procedural language presented in Section 2.2.2. Partial evaluation techniques [BHOS76, ERS82, HAR77] allow the possible evaluation of a rule even if not all of the user input is available. The algorithm for partial evaluation presented in this thesis is an extension of optimization techniques used in compiler design, i.e., constant propagation and dead-code elimination [ASU85]. The algorithm is safe [ASU85], i.e., partial evaluation does not change the output produced by a rule for given input, although the output is obtained incrementally. In many cases during interaction with the user, values are obtained from the user incrementally, therefore partial evaluation of the rules is essential.

Artificial intelligence techniques are not applied since such techniques require a priori knowledge of the application domain, which is not assumed. Therefore it is up to the application designer to set up the data dependencies for a specific dialogue.

Handling User Errors

A third aspect of efficient dialogue management is that of handling incorrect user input. The premise here is that in many cases people insist on their most recent answers and only rarely prefer old responses. Therefore, temporal aspects of the dialogue have to be taken into account. On the technical level, the input from the user consists of several fields obtained from the user. Every field has a time-stamp which is used when the
cause of an inconsistent state is sought. Several fields are allowed to be entered together and, therefore, have the same time-stamp for the following reasons:

1. For many types of dialogues, entering more than one field at once is a natural and efficient mode of interaction. When filling a form or entering a command the user may enter several pieces of information, e.g. all the fields of an address, at once. This also allows the dialogue to conceptually support natural language processing; a sentence contains several pieces of information whose semantics may be determined after the entire statement is entered. Again, all of the fields obtained would be marked with the same time-stamp.

2. Some terminal to mainframe connections do not support key-stroke capture capability. The communication with the host computer is via a collection of fields.

The presentation component or screen manager is responsible for the actual display and for obtaining input from the user. Using an abstract description of the dialogue an actual dialogue is created. This element of the system is only briefly touched upon in this thesis, since the actual dialogue depends on the type of dialogue desired and the physical input devices available. It is shown that the preferred layout for displaying the objects and their context cannot be obtained efficiently. Therefore a similar, computation efficient layout is proposed, along with an appropriate display technique for the proposed layout.

3.2 Reader’s Guide

The layout of following sections is based on dialogue manager’s interaction with the user. However, the order of issues discussed is based on increasing complexity of the model presented. The algorithm receives as input a dialogue definition as described in the users guide. This input consists of the dialogue structure graph as described in Section 2.2.2, rules and constraints. These structures are then used by the dialogue manager to obtain input from the user. The dialogue manager’s flow of control is:
should be notified of the problem. The problems of user navigation and input in a dialogue are discussed in Section 3.3.5. The problem of error handling is discussed in Section 3.6.

4. Simplify and execute the rules using the partial information obtained and program flow analysis techniques. A rule may compute an object’s value or change the topology of the dialogue structure graph by altering the value of the object’s ENABLED property.

The basic algorithms used are extensions of constant propagation and dead code elimination algorithms which customizes rule structure to the partial input obtained from the user. This step and the associated heuristics are described in greater detail in Section 3.4.

5. When the user wishes to terminate the dialogue step and all requested information has been obtained, the list of values which participate in the goal is sent to the application as a packet. Otherwise the procedure is reiterated.

3.3 Selecting Objects for Display from the Dialogue Structure Graph

This section presents an object selection algorithm based on the dialogue structure graph described in Section 2.2.2.

Since the dialogue structure graph is a directed acyclic graph, every vertex may have more than one parent. For example, an insurance company may inquire about both the primary work address and secondary work address, using identical formats. Since it is not certain that the user will enter both addresses, it may be desirable to present only one copy together with some selection mechanism, so that the user can enter the information by indicating what he/she had in mind. If this approach is followed, repetition of information is avoided and the interrelationships between objects are represented as directed acyclic graphs, rather than trees. To each vertex of such a directed acyclic graph, annotation information for description purposes is attached. Obviously, the information attached to an internal vertex \( v \) would describe the successors of \( v \) in a summarized form, and allow navigation within the dialogue structure graph.
For the model described here we used a equivalent, simplified version of the dialogue structure graph which is less explicit but allows for simpler graphs. In this simpler version, an internal vertex is allowed to represent a binary question describing the summary information. Therefore the vertices Spouse_Info and MStatus of the example in Section 2.2.1 become the single vertex c of Figure 6. It is easy to see that this simplified graph is equivalent to the regular dialogue structure graph since a binary question can be considered the summary information, which may or may not be selected in order to define context. Therefore, Figure 6 is a simplification of the dialogue structure graph of Section 2.2.2. This equivalent, simplified version of the dialogue structure graph is referred to in following sections as the dialogue management graph.

When presenting information to the user, it is desirable to ask the most general questions first. This implies a breadth first search [EVEN79] type of approach. This approach seems well suited to tree-like structures [AMSS88]. However, it is inadequate for directed acyclic graphs. For example, consider the directed acyclic graph of Figure 6 with the corresponding annotation information of Table 1.

![Diagram](image)

**Figure 6. A simplified version of the dialogue structure graph of Section 2.2.2**

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Attached Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Employee information</td>
</tr>
<tr>
<td>b</td>
<td>Enter name</td>
</tr>
<tr>
<td>c</td>
<td>Are you married ?</td>
</tr>
<tr>
<td>d</td>
<td>Do you work on a project ?</td>
</tr>
<tr>
<td>e</td>
<td>Do you report to a manager ?</td>
</tr>
</tbody>
</table>

**Table 1. Annotation information attached to the vertices of the graph of Figure 6**
precisely defining what s/he has in mind when answering the Enter name request. The possible fully qualified names of vertex b are a.b, a.c.b, a.d.e.b. As many paths as possible should be shown, but not at the expense of the more general questions.

The question of what to present to the user is actually a question of how many objects to display at once. Any internal vertex which has some descendant not displayed defines a possible continuation of the dialogue and is displayed here with a more suffix attached to it.

The following examples show possible screen displays of Figure 6, for screens of different sizes. The possible contexts of a vertex are displayed by the arrows. An arrow points from a context to its offspring. Since the arrows may cross, junctions between vertical and horizontal lines are marked by blackened circles.

These examples demonstrate that the vertices chosen for display cannot be chosen by using a simple depth first search algorithm [EVEN79] (case 4) or by using a simple breadth first search algorithm (case 2).

Alternative formal definitions of this problem are described later in this section. In these formalizations the appearance of vertices on the screen is dependent on the other vertices to be displayed.

1. A possible two line screen display:

   a: Employee information (more...)
   b: Enter name

2. A possible three line screen display; this choice of vertices for display is based on a Breadth First Search selection of vertices from the graph:

   a: Employee information (more...)
   c: Are you married?
   b: Enter name
Here there is a problem, because once vertex $c$ is displayed, vertex $b$ should be shown at lower layer, i.e. a less general question, than vertex $d$. So if the screen consists of only 3 lines, then the screen should contain:

$\text{a: Employee information (more...)}$
$\quad \downarrow c: \text{Are you married? (more...)}$
$\quad \quad \downarrow d: \text{Do you work on a project? (more...)}$

3. a possible four line screen display:

$\text{a: Employee information (more...)}$
$\quad \downarrow c: \text{Are you married?}$
$\quad \quad \downarrow b: \text{Enter name} \quad \quad \quad \quad$
$\quad \quad \downarrow d: \text{Do you work on a project? (more...)}$

Notice that as the arrows show, there are two ways to get to the Enter name line based on the structure of the screen. Enter name is a descendent of both vertices $a$ and $c$. If the user decides to instantiate this field, the correct context must be indicated, i.e., $a \cdot b$ or $a \cdot c \cdot b$. The third way, namely, going through vertices $d$ and $e$ is not presented explicitly on the screen, and if the user indicates that s/he wishes to see the additional information attached to the question Do you work on a project, a secondary screen may appear explaining this option.

4. an alternative four line layout, based a different selection of four vertices from the graph.

$\text{a: Employee information (more...)}$
$\quad \downarrow d: \text{Do you work on a project?}$
$\quad \quad \downarrow e: \text{Do you report to a manager?}$
$\quad \quad \quad \downarrow b: \text{Enter name} \quad \quad \quad \quad$
This representation is undesirable since vertex c which is not displayed is closer to the root while the more remote vertex e is displayed.

5. a possible five line screen display:

```
a: Employee information
   c: Are you married ?
   d: Do you work on a project ?
   e: Do you report to a manager ?
   b: Enter name ________
```

3.3.1 The Dialogue Management Problem

This section consists of a formal definition of the problem of selecting objects for display from the dialogue management graph. The examples of the previous section make it apparent that neither breadth first search nor depth first search is an appropriate formalism. Some tradeoff between the two types of searches is required, which selects general questions first, while still keeping the questions in context.

Let $D(V, E)$ be a rooted directed acyclic graph with root $r \in V$, $n = |V|$ and $e = |E|$. Let $S \subseteq V$ be a set of vertices such that $r \in S$ and $D[S]$ (the subgraph of $D$ which is obtained by deleting all vertices not in $S$ and all edges incident to them) is connected. Define the level, $\text{level}(v, S)$, of a vertex $v \in S$ with respect to $S$, as the length of the longest path from $r$ to $v$ in $S$. Also, let $\text{level}(S) = \max_{v \in S} \text{level}(v, S)$.

The dialogue management problem (DMP) is the problem of choosing a set $S$ of size $m$ with smallest level such that for no vertex $v \in V - S$, $\text{level}(v, S \cup \{v\}) < \text{level}(S)$. While this definition captures the intuitive nature of the problem, it is not very useful due to the following result:

THEOREM 1: DMP is NP-Hard.

Proof: Consider the following subproblem:

1-DMP: Is there a solution $S$ to the given DMP such that $|S| = m$ and $\text{level}(S) = 1$?
Claim: 1-DMP is NP-complete.

Proof: The independent set problem [GAR79] can be reduced to the 1-DMP problem. The independent set problem (ISP) defined as follows:

ISP: Given a graph $G(V,E)$ and an integer $m$ is there an independent set of $G$ of size $m$, i.e. a subset $I \subseteq V$ such that $|I| = m$ and no two vertices in $I$ are joined by an edge in $E$.

To this end, let $G(V,E)$ and $m$ be an arbitrary instance of ISP. A directed acyclic graph $D(V, \overline{E})$ is constructed for which there is a set $S$ of size $m + 1$ which satisfies the conditions of the 1-DMP if and only if $S - \{r\}$ is an independent set of $G$ of size $m$. In order to construct $D(V, \overline{E})$ from $G(V,E)$ an arbitrary linear order on the vertices of $G$ is imposed and the edges are directed so that they follow the linear order. Then a vertex $r$ with no in-edges and with an out-edge to every vertex in this directed acyclic graph is added.

Assume that the 1-DMP has been solved, namely a set $S$ of size $m + 1$ with $level(S) = 1$ has been found. There are no edges in $S$ which connect the vertices in $S - \{r\}$; otherwise, $level(S)$ would be greater than 1. Therefore $S - \{r\}$ is an independent set of size $m$.

In the other direction, it remains to be shown that if $G$ has an independent set $M$ of size $m$ then 1-DMP has a solution of size $m + 1$. Choose the set $S$ that $S = M \cup \{r\}$. Clearly all the vertices in S have $level(v, S) = 1$ and so $level(S) = 1$. By the construction, $D$ has only one root and therefore no vertex other than $r$ has $level(v, S \cup \{v\}) < 1$. Therefore $S$ is legal and 1-DMP has a solution of size $m + 1$. $
$

Given the difficulty of solving DMP an alternative formulation was sought, called the weak dialogue management problem (WDMP) which is defined as follows:

WDMP: Given $m < n$ choose a set $S$, $|S| = m$, such that no vertex $v \in V - S$ can be added to $S$ and $level(v, S) < level(S)$.

Here the restriction that $S$ should have a minimum level was eased. Unfortunately yet another difficulty arose:
THEOREM 2: Certain WDMPs do not have a solution.

Proof: Below is demonstrated that the WDMP whose directed acyclic graph is shown in Figure 7 has no solution of size 12. In the graph of Figure 7, it is simple to obtain an S of size 11 or less by selecting for addition to S the appropriate vertices above the dotted line in Figure 8. It is clear that for such an S, \( \text{level}(S) \) must be at least 5.

Since an S of size 12 is required, another vertex must be added to S. This vertex \( v \) must have the minimum \( \text{level}(v, S) \) of any vertex in the graph not already in S. Vertex \( l \) is the only vertex not in S with \( \text{level}(l, S) = 5 \) when added to S, therefore \( l \) must be added to S. The addition of vertex \( l \) causes the vertices from \( b \) to \( f \) to be removed from S since, after the addition of \( l \) to S, all these vertices \( v \) have a \( \text{level}(v, S) \) greater than
level(S) = 5. The graph now appears as in Figure 9 with all the vertices above the dotted line in S. So if level(S) = 5, S may be of size 11 or 7.

Figure 9. After addition of vertex \( l \)

After the addition of \( l \) level(S) = 5 and there are no more vertices \( v \) that would have level(v, S) ≤ 5 if added to S. The candidates for addition to S are b and m that will have level(v, S) = 6 when added to S. If level(S) = 6 then the possible sizes of S are 8 and 9. Now the vertex c with level(c, S) = 7 may be added S has level(S) = 7 and is of size 10.

Figure 10. After addition of vertex \( c \)

The candidates for addition to S are now n, o, p, q, r and s. These vertices are the only vertices \( v \) with level(v, S) ≤ 7 when added to S and so must be added to S before any other vertices. The addition of any of
these vertices to \( S \) causes the vertex \( l \) to be removed from \( S \) since \( \text{level}(l, S) = 8 \) and there are still vertices \( v \) with \( \text{level}(v, S) = 7 \).

Figure 11. After addition of vertex \( n \)

Once the vertex \( l \) is removed from \( S \), vertices \( b, c, d, e \) and \( f \) must be returned to \( S \) since their \( \text{level}(v, S) < \text{level}(S) \) = 8. Now \( S \) is of size 13 without having obtained an \( S \) of size 12. An \( S \) of size 12 can be obtained only if by adding vertices to \( S \), will shrink \( S \) (as with the addition of vertex \( l \)) to size 12. The only possible action is to add the vertices \( o, p, q, r \) and \( s \). These additions give an \( S \) with \( \text{level}(S) = 8 \) and possible sizes of 10, 13, 14, 15, 16, 17, 18. The above has shown that for any \( \text{level}(S) \leq 8 \) there is no \( S \) of size 12. The vertices \( a, g, h, i, j, k, m, n, o, p, q, r, s \) in \( S \) have no in-edges from any vertex outside of \( S \) and therefore will never have a level greater than 8 and will never be removed from \( S \). So the minimum number of vertices in any \( S \) with \( \text{level}(S) > 8 \) will be of size 13 or more. Therefore there is no level in which size \( S \) is exactly 12.  

Since for certain values of \( m \), WDMP does not have a solution, a different formulation of the dialogue management was sought. The formulation required should always have a solution which can be computed efficiently.

### 3.3.2 The Monotonic Distance Approach

Let \( \text{distance}(v, S) \) be the length of the longest path \( p \) from the root to \( v \) in \( D \) that is at most of length \( \text{level}(S) + 1 \) or \( \infty \) if there is no such path. The monotonic distance dialogue management problem (MDDMP) is defined as follows:
**MDDMP:** Given \( m \leq n \), choose a connected set \( S, r \in S, |S| = m \), such that there is no \( v \in V - S \) that can be added to \( S \) with \( \text{distance}(v, S) < \text{level}(S) \).

### 3.3.3 An Algorithm to Solve MDDMPs

The algorithm shown in Figure 12 uses the following notation:

- \( A|B \) - Subgraph of the graph \( A \) which is defined by the vertices of \( B \).
- \( D^* \) - Transitive closure of \( D \).
- \( \text{connect}(v, A) \) - The vertices of \( A \) which may be reached from \( v \).

1. \( S := \{r\} \);
2. do until \(|S| = m\);
   a. \( P := \) the set of neighbors of \( S \);
   b. do until \(|P| = 0 \) or \(|S| = m\);
      1) \( v := \) a source in \((D|S \cup P)^*|P|\);
      2) remove \( v \) from \( P \); add \( v \) to \( S \);
      3) delete the children of \( v \) from \( S \) and \( P \);
      4) \( S := \text{connect}(r, D|S) \);
      5) end;
   c. end;

**Figure 12.** Monotonic distance dialogue management algorithm

**THEOREM 3:** The algorithm of Figure 12 solves MDDMP.

In order to prove this theorem we have to show be proven that the following two claims hold when the MDDMP algorithm halts:

1. \(|S| = m\).

   Intuitively, it will be shown that the vertices are added to \( S \) one at a time and that the size of \( S \) is checked after the addition of every vertex. It is also shown that for every set of vertices added and removed during loop 2.b, there is at least one vertex added to \( S \) which is never removed.

2. There is no vertex \( v \in V - S \), that can be added to \( S \) with \( \text{distance}(v, S) < \text{level}(S) \). Intuitively, it will be shown that at the start of every iteration of loop 2, \( S \) contains all the vertices \( v \) with \( \text{distance}(v, S) < \text{level}(S) \) or that \( v \) is not connected to \( S \). It will also be shown that during loop 2 all vertices \( v \) that are connected to \( S \) with \( \text{distance}(v, S) = \text{level}(S) + 1 \) are added to \( S \) unless \(|S| = m\).

To this end the following lemmas are proven:
LEMMA 1: For every $i$ such that $1 \leq i \leq level(V)$, there is at least one vertex $v$ such that $distance(v,V) = i$.

Proof: Let $v \in V$ be the end vertex of the longest path in $D$ from $r$. By definition $v$ has maximum distance in $V$. Since $D$ is acyclic, one of the predecessors of $v$ in $D$ must be closer to $r$ than $v$ by exactly one. Repeating this argument proves the lemma. ■

LEMMA 2: After loop 2 is executed for the $n$-th time and $|S| < m$:

(i) All the vertices $v \in V$ with $distance(v,V) \leq n + 1$ are in $S$ and are never removed.

(ii) $level(S) = n + 1$.

Proof: By induction on the number of times loop 2 is executed.

Basis: Consider $S$ at $n = 0$, before loop 2 is executed for the first time. By then, $S = \{r\}$. Since $r$ is the only vertex $v$ in $V$ with $distance(v,V) \leq 1$ and $level(S) = level(r,S) = 1$, the lemma holds.

Induction hypothesis: Assume that the lemma holds for the $n - 1$-st iteration.

Induction step: Consider the $n$-th iteration. It starts with $S = S_n$ and $P = P_n$. By the definitions of $distance$ and $level$, immediately after statement 2.a, all the vertices $v \in P_n$ have $distance(v,S_n) \leq level(S_n) + 1$. By the induction hypothesis, $S_n$ contains all the vertices $v \in V$ such that $distance(v,V) \leq n$. Therefore, $P_n$ includes all the vertices with $distance(v,V) = n + 1$ and by Lemma 1, there is at least one such vertex. None of these vertices have an ancestor with $distance$ larger than $n$ in $D$. Therefore, these vertices are sources in $(D(S \cup P))^c \cap P$, they will be added to $S$ in statement 2.b.2 and since all their ancestors are in $S$, they will never be removed from $S$. After these vertices are added to $S$, $level(S) = n + 1$ and the lemma holds. ■

COROLLARY 1: During the execution of the algorithm, after loop 2, $distance(v,S)$ does not decrease for any $v \in V$.

Proof: By lemma 2, $level(S)$ is non-decreasing and since, by definition, $distance(v,S)$ cannot decrease when $level(S)$ increases, $distance(v,S)$ is also non-decreasing. ■

LEMMA 3: (i) Immediately after statement 2.a is executed for the $n$-th time, $P$ contains all the vertices $v \in V - S$ connected to $S$ with $distance(v,S) = n + 1$ and only those vertices.

(ii) At the end of the $n$-th iteration of loop 2, if $|S| < m$ there is no vertex $v$ connected to $S$, $v \in V - S$ with $distance(v,S) \leq n + 1$.

Proof: By induction on the number of times loop 2 is executed.
Basis: Consider the first time statement 2.a is executed. By then, \( S = \{r\} \) and \( \text{level}(S) = \text{level}(r, S) = 1 \). Immediately after statement 2.a, \( P \) contains all of the neighbors of \( r \) and these are all the vertices whose distance from \( S = \{r\} \) is 2. Thus (i) is proven. After loop 2.b has completed, all the sources have been added to \( S \). The other vertices \( v \) have a larger value of distance \( (v, S) \) which proves (ii).

Induction hypothesis: Assume that the lemma holds for all iterations 1, 2, ..., \( n - 1 \).

Induction step: Consider the \( n \)-th iteration. It starts with \( S = S_n \) and \( P = P_n \). By the induction hypothesis, after the \( n - 1 \)-st iteration of loop 2, \( S_{n-1} \) contains all the vertices \( v \in V \) connected to \( S_{n-1} \) with distance \( (v, S_{n-1}) \leq n \). Since \( S_{n-1} \) is not altered until after statement 2.a is executed for iteration \( n \), this is true also for \( S_n \) immediately before statement 2.a in iteration \( n \). By Lemma 2, when the \( n - 1 \)-st iteration starts, \( \text{level}(S_n) = n \). By the definition of distance and level, \( P_n \) contains all the vertices \( v \in V - S_n \) connected to \( S_n \) with distance \( (v, S_n) \leq \text{level}(S_n) + 1 \) and only those vertices. Thus, for all the vertices \( v \in P_n \), distance \( (v, S_n) \leq n + 1 \). Since, by the induction hypothesis, all the vertices with distance \( (v, S_n) \leq n \) are already in \( S_n \), \( P_n \) contains all the vertices with distance \( (v, S_n) = \text{level}(S_n) + 1 = n + 1 \) and no others. Thus (i) is proven.

To show (ii) we claim that during loop 2, the vertices removed from \( P_n \) and \( S_n \) have distance \( (v, S_n) > n + 1 \) or are not connected to \( S_n \). We also claim that when the \( n \)-th iteration is over all the vertices with distance \( (v, S_n) = n + 1 \) which are connected to \( S_n \) are included in \( S_n \) unless \( |S| = m \).

Let us assume that after loop 2 completes with \( |S| < m \), there is some vertex \( v \in V - S \) connected to \( S \) with distance \( (v, S_i) = i \) for some \( i \leq n + 1 \). Vertex \( v \) is not a descendent of some \( w \) that was added from \( P_n \) to \( S_n \), since by (i), if \( v \) is a direct descendent of such a \( w \), distance \( (v, S_n) = n + 2 \), which is a contradiction to the assumption that \( i < n + 1 \). Therefore, by the induction hypothesis and (i), \( v \) must have been included in \( S_{n-1} \) if \( i \leq n \) or \( P_n \) if \( i = n + 1 \). Since \( v \) is not in \( S \) it must have been removed from \( S_n \) or \( P_n \) during iteration \( n \). This could happen only if either \( v \) were a direct descendent of \( w \in P_n \) which was added to \( S_n \) in statement 2.b.2 (shown above to be impossible) or some ancestor \( w \in P_n \) of \( v \) was added to \( S_n \), and consequently \( v \) was removed in statement 2.b.4. In the second case, since \( v \) was removed in statement 2.b.4, it is not connected to any predecessor in \( S_n \), which contradicts the assumption.

By Corollary 1, any vertex \( v \) with distance \( (v, S_n) = n + 1 \) when loop 2 completes, must have had distance \( (v, S_i) \leq n + 1 \) before loop 2 started. If vertex \( v \) had distance \( (v, S_i) < n + 1 \) before loop 2 started, by
the induction hypothesis would have been in $S_n$ and as we have shown above would have been removed from $S_n$ only if its distance had increased to $n + 2$ or it is no longer connected to $S_n$. Otherwise, if vertex $v$ had distance $(v, S_n) = n + 1$ before loop 2 started, then by (i), $v \in P_n$ and if $v$ is not removed from $P_n$ in statements 2.b.3 or 2.b.4 because its distance has increased to $n + 2$ or it has become disconnected, it will eventually be added to $S_n$ in statement 2.b.2 or $|S| = m$ and the algorithm terminates.

Now the following claims can be proven:

Claim 1: $|S| = m$

Proof of Claim 1: By lemmas 1 and 2, there is for every iteration of loop 2, at least one vertex added to $S$ that is never removed.

Since $|S|$ is initially 1, and $|S|$ is checked every time $|S|$ is incremented by 1 and since finally $|S| = |V| \geq m$ there must be some step where $|S| = m$.

Claim 2: There is no vertex $v \in V - S$, that can be added to $S$ with distance $(v, S) < level(S)$.

Proof of Claim 2: Immediate from lemma 3.

3.3.3.1 Efficient Implementation and Complexity

Lemma 4: Loop 2 is executed at most $m$ times.

Proof: By Lemma 2(i) during the $n$-th iteration all vertices $v$ with distance $(v, V) \leq n + 1$ are added to $S$ and are never removed. By Lemma 1, for every $1 \leq n \leq level(V)$ there is at least one such vertex.

Aiming at an $O(me)$ upper bound on the complexity of the algorithm (where $e$ is the number of edges in the input graph), and in view of Lemma 4, it must be shown that the complexity of statements 2.a and 2.b is $O(e)$

Statement 2.a is easy to analyze: all the neighbors of $S$ may be found by considering each edge of $D$ at most once. Therefore the complexity of statement 2.a is indeed $O(e)$.

Lemma 5: All iterations of loop 2.b performed during one iteration of loop 2 require $O(e)$ time.

Proof: When a new set $P$ is obtained as a result of statement 2.a, the vertices and edges in $D(S \cup P)$ are preprocessed. First color all the edges in $D(S \cup P)$ blue. Then, for every $v \in P$, each out-edge
\( v \rightarrow u, u \in S \cup P \) is colored red. For every \( u \in S \) with at least one red in-edge all its out-edges are colored red. The procedure is repeated until there are no more edges to be colored. Since each edge is colored at most once, the entire process takes \( O(e) \) time.

An additional structure used is \( in_v \), a counter associated with each vertex \( v \in S \) which indicates the number of red in-edges of \( v \). Finally, to make finding sources in \( (D(S \cup P)^*) \)\( P \) efficient, notice that each such source must be an element \( v \in P \) for which \( in_v = 0 \). To this end the set of subsets of elements of \( P \) with the same value of \( in \) is maintained such that each vertex may be accessed in constant time. In addition, the ability to find a vertex \( v \) with \( in_v = 0 \) and move a vertex from one subset to another in constant time is required. Such a data structure is easy to design.

The preprocessing described above takes \( O(e) \) time. It makes finding sources (statement 2.b.1) easy provided that the data structures can be maintained efficiently. Statement 2.b.2 is also simple. It takes \( O(1) \) time for every iteration of loop 2.b.

Now consider statements 2.b.3 and 2.b.4. The children of \( v \) are removed along with their out-edges. This takes \( O(e) \) altogether since each edge is processed once. The removal of orphans \( u \in S \cup P \) takes \( O(|V|) \) time. For every red out-edge \( u \rightarrow w \) of a vertex \( u \) removed from \( S \), \( in_u \) is decremented by one and the data structures are updated as mentioned above. If for some \( w \in S \), \( in_u \) becomes zero, then the color of every out-edge \( w \rightarrow x \) is changed to blue and \( in_x \) is decremented by one. The recoloring routine is repeated until there is nothing left to recolor.

The above process either removes a vertex, removes an edge, or changes the color of an edge at most once during loop 2.b. During the process, all the orphans in \( D[S \) have been removed and therefore what remains is \( \text{connect}(r, D[S) \). All the data structures have been updated so as to be ready for the next iteration of loop 2.b. Therefore, loop 2.b takes a total of \( O(e) \) time for one run of loop 2.
3.3.4 Example of the MDDMP Algorithm

Using the MDDMP algorithm on the graph $D$ of Figure 7 and given a screen size of 10, the following subgraph $D|S$ is obtained:

For a screen size of 13 we obtain:

It is clear that the subgraph chosen is different than the BFS or DFS subset of the same size.

3.3.5 Navigation and User Input in the Dialogue Management Graph

The graph $D|S$ derived by the MDDMP algorithm when it is applied to a dialogue management graph defines the display graph. The display graph is a window into the dialogue management graph that defines a set of objects that can be displayed to the user at one time, for instance on one screen. The user interacts with the display graph for data entry and navigation in $D$. Since a vertex in $D|S$ may define more than one object, an object obj is selected by specifying the path from the vertex $v$ representing obj to the root of $D|S$, which defines the fully qualified name of obj.
Once the user has selected \textit{obj}, there is a need to distinguish between the role of \textit{v} as a representative of \textit{obj} and its role as a representative of other objects. This is done as follows: Let \( u \rightarrow w \) be an edge on the path from the root to \( v \). Let \( P(w) \) and \( C(w) \) be the sets of parents and children of \( w \) respectively. If \(|P(w)| > 1\) then we introduce a new vertex \( w' \) into \( D \). The edge \( u \rightarrow w \) is deleted and the edges \( u \rightarrow w' \) and \( \{w' \rightarrow x \mid x \in C(w)\} \) are added.

We see that \( D \) may change as a result of the interaction with the user: it becomes less compact and more explicit. If \( v \) is an \textit{external vertex (leaf)} then it is used for data entry; otherwise, it is used to navigate through the dialogue management graph.

\textit{Internal vertices} can also be classified into two categories: \textit{continuation} vertices and \textit{context} vertices. Context vertices are used only to group vertices together and define the context of their descendents. Therefore, context vertices are informative and are displayed in order to keep the user in context and to allow selection of fully qualified paths from the root to the leaves. All the direct descendents of a context vertex are included in the current display graph.

A continuation vertex, like a context vertex, defines the context of its descendents but has direct descendents in \( D \) that are not in \( D|S \) and, as such, are not displayed to the user. In the example screens, continuation vertices are designated by the suffix (more...). The user selects such vertices in order to see the continuations possible from this vertex towards the leaves of \( D \) so as to see different sections of \( D \) not contained in \( D|S \). The user may select more than one continuation for display. In this case, a \textit{virtual root} is generated, to be treated by the MDDMP algorithm as the root of the new display graph to be built. Fully qualified names are obtained by concatenating fully qualified names of continuation vertices with the fully qualified name of the vertex of interest in the current display graph. Navigation back to the root of \( D \) when it is not shown on the screen is done by a direct command to the dialogue manager, rather than by selection of vertices in \( D|S \).

In the example graph of Figure 6, since vertex \( b \) is a leaf vertex, each of the objects represented by the three possible complete names \( a.b, a.c.b \) and \( a.d.e.b \) may receive input from the user. After each of the objects represented by the fully qualified name is selected by the user, the dialogue graph has the following form:
This representation has 2 more vertices than the graph of Figure 6 and would therefore require a larger screen for complete display.

### 3.3.6 Extending the Dialogue Management Graph

The dialogue management graphs of the above model are useful in describing the hierarchical information pertaining to the dialogue objects. This simplified model supports only flat dialogues - dialogues that consist of sets of questions to be answered along with their context. The model may be extended by imposing additional structure among the dialogue objects.

This additional structure is obtained by attaching selection criteria to the internal vertices. These criteria restrict the number of direct descendants of an internal vertex that may or must be selected by the user. This extension supports dialogue specifications which contain single or multiple choice menus. For example, the relation *exactly* \((v, i)\) states that the internal vertex \(v\) must have exactly \(i\) direct descendants selected by the user. Obviously the criterion of *exactly* \((v, 0)\) is of no interest. Other possible criteria are *atleast* \((v, i)\) and *atmost* \((v, i)\). This is introduced into a dialogue definition by the value field of an internal object definition.

![Diagram](image)

**Figure 13.** A dialogue management graph (an extension of Figure 6)
<table>
<thead>
<tr>
<th>Vertex</th>
<th>Attached Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Employee information</td>
</tr>
<tr>
<td>b</td>
<td>Enter name</td>
</tr>
<tr>
<td>c</td>
<td>Are you married?</td>
</tr>
<tr>
<td>d</td>
<td>Do you work on a project?</td>
</tr>
<tr>
<td>e</td>
<td>Do you report to a manager?</td>
</tr>
<tr>
<td>f</td>
<td>Enter project start date</td>
</tr>
<tr>
<td>g</td>
<td>Enter projected completion date</td>
</tr>
<tr>
<td>h</td>
<td>Project type</td>
</tr>
<tr>
<td>i</td>
<td>Internal</td>
</tr>
<tr>
<td>j</td>
<td>Civilian</td>
</tr>
<tr>
<td>k</td>
<td>Military</td>
</tr>
</tbody>
</table>

Table 2. Annotation information attached to the vertices of the graph of Figure 13.

![Graph Image](image)

Figure 14. An example of a single choice menu for graph of Figure 13

An internal vertex is *satisfied* when its selection criteria is satisfied. By definition the root (goal) is satisfied when all of its direct descendents are satisfied. The packet of information requested by the dialogue management graph from the user is *ready* when the criteria of the root vertex of the dialogue management graph is satisfied.

The selection criteria should be made known to the user, so that he knows how many items may be selected. It seems natural to display the selection criteria along with the text of the associated internal vertex. On the screen shown in Figure 3, the selection criteria *[All of the following]* is displayed along with the text *Marital Status.*
The dialogue management graph is now a complete model of the objects obtained from the application. The addition of this additional structure does not effect the MDDMP algorithm.

3.3.7 Displaying the Display Graph

The actual display of the display graph and the interaction with the user are carried out by the screen manager, a software module that is independent of MDDMP. The screen manager must display both the dialogue objects, which are abstractly represented as the vertices of the display graph, and the hierarchical interrelationships between the objects, which are abstractly modelled by the display graph edges.

Since the display graph may not be a tree, there is the problem of ordering the vertices for display. Since the predecessors of a vertex define its context, it should always be displayed as soon as possible after its predecessors. The screens shown in Section 3.3 are examples of such an arrangement. In the general case, obtaining an optimal arrangement of the vertices of the display graph so that each vertex is as close as possible to its possible contexts is equivalent to the directed optimal linear arrangement (DOLA) problem presented in [GAR79]:

\( \text{DOLA: Given a directed graph } G = (V, A) \text{ and a positive integer } k, \text{ is there a one-to-one function } f: V \to \{1, 2, \ldots, |V|\} \text{ such that } f(u) < f(v) \text{ whenever } (u,v) \in A \text{ and such that } \sum_{(u,v) \in A} (f(v) - f(u)) \leq k? \)

Unfortunately, DOLA is known to be NP-Complete. Therefore we have chosen to determine the order of vertex presentation on the screen according to the following rules:

- A vertex will appear only after all its parents have already appeared (in particular, the root will appear first);
- The next vertex to appear will have a parent which appeared most recently.

These rules define the context ordering problem: (COP)

\( \text{COP: Given a display graph } DG(V, E), \text{ build an ordered list } L \text{ of the vertices } V \text{ so that:} \)

\[ L = L, v \text{ if } v \in V - L \text{ and among the vertices with all their parents in } L, v \text{ has the parent most recently added to } L. \]
The above formulation of the problem also implicitly defines the algorithm used to solve it. The display technique for the ordered objects obtained was given in Section 2.3.

3.4 Rule Execution

In order to extend the model of the dialogue definition language and make it more useful, the dialogue management graph must be augmented by semantic knowledge of the dialogue domain. This knowledge is contained in the RULES and CONSTRAINTS part of the dialogue definition language of Section 2.2. Below, the execution of rules and their integration into the DMP algorithm is discussed.

3.4.1 The Constant Propagation/Partial Evaluation Problem

The constant propagation/partial evaluation problem consists of automatically rewriting the rules to reflect those values which are already known. This rewriting is done by program flow analysis [ASU85, WZ84] and partial evaluation [BHOS76, ERS82, HAR77] of the rule. This is done in order to deduce as much as possible about the uninstantiated objects of the rule. The partial evaluation is based on value propagation and dead-code elimination. A possible path can be eliminated by evaluating a condition or a loop. A condition can be evaluated if all the participant variables are instantiated, while a loop can be evaluated if all of the loop input variables are instantiated. A loop input variable is an variable used on some possible path within the loop, but not previously defined within the loop.

The evaluation scheme employed here is safe [ASU85], i.e., it does not change the values computed by the rules, since both constant propagation and dead code elimination are known to be safe [MJ84]. The addition of loop execution preserves safeness, since a loop is executed by interpretation of its code after all of its required input is defined.
3.4.1.1 Rule Minimization Problem

Let \( R(I, O) \) be a rule where:

\( I = \{i_1, \ldots, i_n\} \) are the input variables of \( R \), these are the variables that are used before they are defined in the body of \( R \). \( O = \{o_1, \ldots, o_m\} \) are the output variables of \( R \), these are the variables that are defined in the body of \( R \).

Let \( C^{AS} \) be an assignment of the input variables \( I^{AS} \subseteq I \) with the constants \( \{c_1, \ldots, c_p\} \). Two rules can now be defined equivalent under the assignment \( C^{AS} \), \( R(I, O) \equiv_{C^{AS}} R'(I', O') \), if:

1. \( I = I' \cup I^{AS} \cup I^{ER} \) where \( I' \) are the variables not assigned, \( I^{AS} \) are the variables assigned, and \( I^{ER} \) are the variables no longer required by the rule on account of the assignment.
2. \( O = O' \cup O^{AS} \cup O^{ER} \) where \( O' \) are the variables that have not yet been computed by the rule, \( O^{AS} \) are the variables computed by the rule and \( O^{ER} \) are the variables no longer computed by the rule.
3. \( z_{C^{AS}} = \{z \mid z : I \rightarrow V \cup \{\Psi\} \text{ such that } \forall i \in I^{AS}, z(i) = c_i\} \) where \( V \) is the set of possible values for variables and \( \Psi \) is the value undefined. Here all the variables in \( I^{AS} \) are assigned values.
4. \( f^*_r : O \rightarrow V \cup \{\bot\} \cup \{\Psi\} \) where \( V \) and \( \Psi \) are the same as above and \( \bot \) is the value of a variable no longer evaluated by the rule. Intuitively, \( f^*_r \) is the evaluation of the rule \( R(I, O) \) for a given assignment \( z \).
5. \( \forall z, z_1 \in z_{C^{AS}}, \forall o_1 \in O^{AS}, f^*_r(o_1) = f^*_r(o_1) = c'_n, c'_i \) are constant values. Intuitively this means that for any two given assignments, the values assigned to the variables of \( O^{AS} \) are the same.
6. \( \forall z, z_1 \in z_{C^{AS}}, z(x) = z_1(x), \forall o_1 \in O', f^*_r(o_1) = f^*_r(o_1) = f^*_r(o_1) = f^*_r(o_1) \). Intuitively, for any two given assignments, if the assignments to \( I' \) are the same, then the evaluation of both \( R(I, O) \) and \( R'(I', O') \) will give the same values to \( O' \).
7. \( \forall z, z_1 \in z_{C^{AS}}, \forall o_1 \in O^{ER}, f^*_r(o_1) = f^*_r(o_1) = \{\} \). Intuitively, no assignment affects the values of the variables in \( O^{ER} \), i.e. these variables are no longer evaluated by the rule.

The rule minimization problem can now be defined as:

Given a rule \( R(I, O) \) and a partial assignment \( C^{AS} \) find an equivalent rule \( R'(I', O') \) such that \( |I'| \) and \( |O'| \) are minimum.

Intuitively, the above notation states that a rule should deduce as much as possible based on the partial input obtained from the user. In the system presented here an approximation of the minimum representation
for a rule \( R(I, O) \) under an assignment \( C^{as} \) of input obtained from the user is achieved by partially evaluating
the rule using input values already obtained.

3.4.1.2 Internal Representation of Rules

Every rule is represented by two hierarchical graphs:

1. The Program Flow graph,
2. The DefUse graph.

In this representation all the language constructs except for loops are represented as in regular DefUse and
Program Flow graphs [ASU85, WZ84]. Loops are represented by special vertices which define a hierarchy
between graphs. In the hierarchical Program Flow graph, a loop vertex represents an hierarchical Program
Flow graph describing the loop, and the in-edges and out-edges of the loop vertex are the same as the
in-edges and out-edges of the loop in the Program Flow graph. Similarly, an hierarchical DefUse loop
vertex represents an hierarchical DefUse graph which describes the data-flow within the loop. In an
hierarchical DefUse graph a loop vertex would have in-edges leading to all the variables of the loop that are
used before they are defined in the loop. If all of these variables obtain values during the dialogue the loop
can be executed or skipped, according to the value of the loop condition variable. The out-edges represent
all the variables defined by the loop that are consequently used in the rule.

From the computational point of view, a loop vertex can be considered in one of two modes:

1. Loop execution
2. Constant propagation + dead-code elimination

If all the data required by the loop is defined, then the loop can be executed and the values obtained are
placed on the out-edges of the DefUse loop vertex. If there are some undefined loop input variables the loop
may be analyzed by constant propagation techniques and the values obtained propagate through the out
edges of the DefUse loop vertex.
2. Output objects must not contain intermediate values used by the algorithm, since these values are displayed to the user as soon as they are obtained. The algorithm in [WZ84] may place erroneous values in output objects if they are not used by some later program statement.

3. Loops should be executed whenever possible since doing so increases the information obtained from partial user input.

These extensions change the ConditionalConstant algorithm from a constant propagation algorithm to a partial evaluation algorithm. Since the algorithm presented here is a variant of the algorithm given in [WZ84], the same notation is used here.

**Lattice for Constant Propagation:** To each variable defined or used within a given assignment or conditional vertex of the Program Flow graph is attached a set of attributes designated as ValueCell and a single attribute designated as LevelCell. Each ValueCell and LevelCell attribute contains either the elements top, bottom or the constant value of the variable. These elements comprise a lattice where bottom means that a constant value cannot be guaranteed, top means that the variable may be some as yet undetermined constant. As more information is obtained about the constants, a variable's attribute may move to a lower lattice level. Each variable used in an expression has a ValueCell for each definition that reaches this use of the variable and a LevelCell which contains a single attribute which is the result of using the meet rules on the attributes of the ValueCells.

```
                   top
                   |
                 constant
                   |
              bottom
```

*Figure 15. Constant propagation lattice*

**Meet Rules:** The rules described here are the same as the rules in [WZ84].

- \( a \cap top \rightarrow a \)
- \( a \cap bottom \rightarrow bottom \)
- \( constant \cap constant \rightarrow constant \) (if the values are equal)
- \( constant \cap constant \rightarrow bottom \) (if the values are not equal)

*Figure 16. Meet rules*
Figure 15. Constant propagation lattice

Meet Rules: The rules described here are the same as the rules in [WZ84].

\[ a \cap top \rightarrow a \]

\[ a \cap bottom \rightarrow bottom \]

\[ constant \cap constant \rightarrow constant \text{ (if the values are equal)} \]

\[ constant \cap constant \rightarrow bottom \text{ (if the values are not equal)} \]

Figure 16. Meet rules

Expression Rules: Variables have values at the entrance of vertices and at the exits of vertices. If a variable is not assigned a value then its value is unchanged by the vertex. If the vertex is an assignment, and any of the variables used in the expression portion of it has a value of bottom, the value exiting the statement for that variable is bottom. If all the values used in the expression portion are constant, the value of the assigned variable is the value exiting the expression, when evaluated with those constant values. Otherwise the value is assigned top.

If the vertex is a loop statement, then if all the variables in the condition and the loop's body are defined, the loop can be executed or skipped, based on the value of the condition.

Rule Preprocessing: In order to force the algorithm to allow output objects to have only their exact values and not some intermediate value that may have been computed by the constant propagation algorithm, some preprocessing of the rule definition is done. First an identity assignment \( x := x \) is added for every input object at the end of the rule. The preprocessing described in [WZ84] is also done. This consists of adding an identity assignment to every vertex in the Program Flow graph that is a predecessor to a birthpoint as defined in [RT81] and is not also a definition site. The definition of birthpoint for a variable \( v \) is as follows:

- Each definition site for \( v \) is a birthpoint.
This flag is initially false for all vertices. Also, all the ValueCells of the input objects are initially set to bottom. The algorithm is run whenever some input object for the rule (program) is changed. For every input object changed by the user, set the ValueCell of that object to the constant value obtained and add all the DefJoin edges emanating from these input variables to the DefWorkList. The FlowWorkList is initialized to contain the program's start edge.

1. Execution is halted when all the worklists become empty. Execution may proceed by processing items from any worklist, although items in the RemovedWorkList are processed before any others.

2. If the item is a Program Flow graph edge from the RemovedWorkList, mark the ExecutableFlag as removed. If the value of the ExecutableFlag has been changed to removed, set the ValueCell to top. If the value of ValueCell has changed, do VisitExpression.

3. If the item is a Program Flow graph edge from the FlowWorkList, mark the ExecutableFlag as true. If the vertex is a loop vertex and the value of the ExecutableFlag has been changed to true, set the ExecutableFlag for the start vertex of the loop and evaluate the loop by the Loop Rules. Otherwise, evaluate the expression according to the rules in Expression Rules. If the value has changed, do VisitExpression.

4. If the item is a DefJoin edge from the DefWorkList, the source value is combined with the value at the join vertex according to the rules in Meet Rules. If this causes the lattice value to be changed, the new value is then propagated to the expression (if the expression is a loop vertex then the new value is propagated to the proper input variable in the loop's DefUse graph). Then if the ExecutableFlag is true for that expression, the expression is evaluated. If evaluation causes the LevelCell to be changed, do VisitExpression.

VisitExpression is defined as follows:

1. If the expression is part of an assignment, add all DefJoin edges starting at the definition for that vertex to the DefWorkList. If the ExecutableFlag of the vertex is true add the Program Flow graph edge from this vertex to its descendant to the FlowWorkList. Otherwise, if the ExecutableFlag of the vertex is the value removed, add the Program Flow graph edge from this vertex to its descendant to the RemovedWorkList.
2. If the expression controls a conditional branch, some Program Flow graph edges will have to be added to the FlowWorkList. If the ExecutableFlag of the vertex is marked removed add the Program Flow graph edges from this vertex to all of its conditional descendants to the RemovedWorkList. Otherwise, if the LevelCell has the value bottom, both exit edges are added to the FlowWorkList. If the value is constant, the Program Flow graph edge that is to be executed as the result of the branch is added to the FlowWorkList and the edge that is not executed is added to the RemovedWorkList.

This algorithm tries to obtain the output objects of a rule even if not all the input objects are available. Input objects that are no longer needed by the rule as a result of dead code elimination should be marked as such. These are input objects whose values are never used. This can be done by another pass through the DefUse graph using another worklist, NotUsedWorkList, which contains DefUse graph edges which are never used by the rule with the given values of the input variables.

**Redundant Input Object Removal Algorithm**

1. For every variable defined by a Program Flow vertex marked removed, add the DefJoin edges that are used to define this variable into NotUsedWorkList.

2. For every variable with all of its out DefUse edges in NotUsedWorkList, add the DefJoin edges that are used to define this variable into NotUsedWorkList.

3. For every input variable and loop input variable, if all the definition edges from the variable are in NotUsedWorkList then mark the variable as not_used.

When the algorithm terminates, all of the rule's input objects that are marked not_used are no longer required in order to be able to execute the rule.

**Loop Rules**

1. If there exists a loop in edge not marked not_used which is either not defined or inconsistent, or the input objec whose value has changed is not marked not_used, run the incremental ConditionalConstant algorithm on the loop's DefUse graph.

2. Use the input variable removal algorithm to mark the in-edges to the loop which are never used. Check whether all the loop input edges that are not marked not_used are defined and consistent. If so, execute
the loop and for every assignment expression executed, add all the DefJoin edges eminating from that assignment expression to the DefWorkList.

3.4.2.1 Partial Evaluation

Whenever an input object's value is changed, the partial evaluation algorithm is run for every rule that contains that object as an input object. The output values obtained are displayed to the user as deduced values. This cuts down the amount of information required from the user.

When the algorithm terminates, use the redundant object removal algorithm to mark unwanted objects in the input objects list. This also cuts the amount of information needed from the user in order to execute the rule, since it has been ascertained that some variables will never be used, and are therefore not required by this rule from the user.

3.4.2.2 Example

The following is an example rule. Its purpose is to compute the value of end_value based on the values obtained from the user for max_sum, x and y. Below, we examine how the algorithm deals with the rule.

```
sum := 0
loop1: while (sum & Al. max_sum)
    if (max_sum & Al. 100) then
        loop2: while (x & Al. 11)
            begin
                sum := sum + 10*x
                x := x + 1
            end
        else
            loop3: while (y & Al. 12)
                begin
                    sum := sum + 15*y
                    y := y + 1
                end
            if (max_sum & Al. 100) then
                end_value := x
            else
                end_value := y
```

Figure 17. An example program for the incremental ConditionalConstant algorithm
In the following DefUse graphs DefUse edges are divided into DefJoin and JoinUse edges as in [WZ84]. Every variable used has a join box into which all of the definitions of the variable enter. There is a single edge from the join box to the use of the variable. An dead-end edge marks an output variable definition.

Figure 18. Program Flow graph for the example program
Let $D(V, E)$ be a rooted acyclic graph and let $R(I, O)$ be a rule with input objects $I$ and output objects $O$. Define $v \in V$ as possibly deductible by $W \subseteq V$ under rule $R(I, O)$ if $I \subseteq W$ and $v \in O$. A possibly deductible vertex is of lower priority for addition from $P$ to $S$. A possibly deductible vertex, if added to $S$, is marked as such so that the dialogue manager may notify the user of this fact and possibly place the vertex in a less accessible part of the screen.

In order to extend the DMP algorithm, internal vertices of $D$ are separated from the and external vertices of $D$ (leaves). For every source $w \in P$ possibly deduced by $S$ under some $R(I, O)$ that is to be added to $S$ in Step 2.b.2 of the DMP algorithm, create a new set $PD$ which contains the possibly deductible vertices in $P$ ($PD$ is initially the empty set): $P = P - \{w\}$, $PD = PD \cup \{w\}$.

If there are no vertices left in $P$ then the vertices in $PD$ are marked as possibly deductible and replaced into $P$.

When adding a source $v$ from $P$ to $S$, we check if there is some set of vertices $U$ in $S$ that are possibly deductible by the set $S \cup \{v\}$. For every $u \in U$:

1. if $u$ is an external vertex then Mark $u$ as possibly deductible.
2. if $u$ is an internal vertex then $u$ may be needed in order to keep its descendents in context. Therefore, for every $v \in S$ whose ancestors $w \in S$ are all in $U$ under some set of rules, then $w$ is also marked as possibly deductible.

This addition to the algorithm forces the rule base to be checked once for every object instantiated by the user that is to be placed in $S$. All the rules must be checked to decide which may be applicable to the current $S$ and which objects are possibly deduced by $S$.

### 3.5 Constraints

The constraints of the CONSTRAINTS section of a dialogue definition are used to define passive interdependencies between the values that the user assigns to various dialogue objects. A typical requirement of this nature would state that $f < g$ where $f$ is, say, the starting date of a project and $g$ is the planned
completion date. Since these are actually semantic constraints imposed on the dialogue objects, a new kind of error may occur once such constraints are introduced - two or more values entered by the user may conflict with each other with respect to some semantic constraint.

The handling of these and other errors is discussed in the next section.

3.6 Handling User Errors

While the introduction of selection criteria (Section 3.3.6), rules (Section 3.4) and contraints (Section 3.5) allow for more interesting dialogues, it also complicates the interaction with the user since now global errors are possible. A global error is an error caused not by the value of single object but by the interaction between two or more object values.

3.6.1 Violation of Selection Criteria

To allow the user to pinpoint the source of the error and correct it an error graph is built. A vertex v will participate in the error graph, if either the criterion attached to it or to one of its ancestors does not hold. Notice that even if the criterion attached to v does not hold, the criterion attached to the parent w of v may hold. If this is the case, then there is nothing to be gained by displaying w to the user. Thus, the error graph may have several roots $r_1, ..., r_n$. In such cases, a new artificial root r connected to $r_1, ..., r_n$ is added to the error graph.

For example, in Figure 13, if the user selects two descendants of vertex h, the error graph will consist of the graph defined by vertices h, i, j and k.

Having defined the error graph, what is left is to present it to the user and allow him to navigate through it, assuming that it cannot fit on one screen. For this purpose, the dialogue management algorithm can be used while clearly indicating to the user that this is actually an error correction mode. The user can then correct the errors by changing some value previously entered. Alternatively, s/he may choose to ignore the error and continue as if no problem occurred. However, whenever the constraints are rechecked the user is again
notified of the error and errors must be fixed before the information is communicated back to the
application.

After a consistent set of values has been obtained for the current display graph, a new display graph is
created from the dialogue management graph and the user may continue to enter values.

3.6.2 Violation of Semantic Constraints

The simplest way to handle violation of constraints would be to consider the last value entered by the user
as inconsistent, reject it, and request the user to reenter a new value. This really means that the most recent
input is always considered incorrect. An alternative approach is to assume that the older values are
erroneous and that the user entered the new values intentionally. Then the system has to and ask the user to
update the old values. The algorithm to realize this approach is given in this section.

Similar to the way violations of selection criteria are handled, an error graph could be built which would
include the conflicting objects. Again, an effort is made to build as small a graph as possible in order to
allow the user to concentrate on the objects that caused the problem. Still, since the entire dialogue
management graph is reachable, the user may choose to ignore the focus offered to him and work on other
sections of the dialogue first.

3.6.2.1 Check Constraints

The input to this routine consists of the constraints and objects. The output is a set of objects causing the
conflict.

The algorithm is run after the user enters a value or when a new value is deduced by the system.

1. If the conflicting value was deduced by the system then nullify the last value deduced, i.e. that which
caused the conflict, and check the constraints again.

2. Otherwise, the conflicting value was obtained from the user: and the list ERASED = {} will contain the
objects causing the conflict.
a. Check the constraints and create a list of all the false constraints, \( \{const_1, const_2...const_N\} \).

b. For every false constraint, create a list of the objects participating in the constraint, \( \{obj_1, obj_2...obj_n\} \).

User selected objects, not deduced objects are of interest here. If the object \( obj_i \) is a deduced object then find the rule which deduced \( obj_i \), i.e. a rule in which the input is defined and the output is \( obj_i \).

Continue in this fashion until all the objects in the constraint list are user selected objects.

c. Choose from every list the least recent object, i.e., with the smallest time-stamp. Create a list of these objects \( OLD = \{obj_{1,1}, obj_{2,1}...\} \).

d. Order the list \( OLD \) in ascending order by time-stamp.

e. Nullify the value of the first object in \( OLD \) while keeping the previous value, remove it from \( OLD \) and add it to \( ERASED \). Are the constraints consistent? If not repeat (e).

f. For every object in \( ERASED \), from the newest to the oldest, return the previous value of the object and check the constraints. If the constraints are consistent leave the value and remove the object from \( ERASED \); otherwise just leave the value.

3. The set \( ERASED \) contains the objects causing the conflicts. These objects are to be presented to the user as described earlier in this section along with notification of which constraints were violated.

### 3.6.2.2 Complexity of the Check Constraints Algorithm

The following notation is used in the complexity analysis of the Check Constraints algorithm:

- \( N \) - number of constraints.
- \( Obj_i \) - number of the objects used by constraint \( i \).

Statement 1 is simple; it takes constant time for every object obtained from the user.

Statement 2.a checks which constraints are false, therefore this statement together with Statement 2.b take \( O(\sum Obj) \) time. Statement 2.c goes through the objects in each constraint, so its complexity is \( O(\sum Obj) \).

Statement 2.d involves sorting, a list consisting of one object from every constraint obtained from Statement 2.c is sorted, so by using an efficient sorting algorithm the time required is \( O(N \log N) \).

Statements 2.e and 2.f each go through a list of length \( N \) created in Statement 2.d therefore take \( O(N) \) time each.

The total time complexity of this algorithm is \( O(\max(\sum Obj, N \log N)) \).
3.7 Extended Example

This example was taken from a US 1987 1040 tax return form. It is a shortened version, but still demonstrates the ease of defining actual dialogues and the assistance that the system gives to the dialogue user. Based on the example, the efficiency of the ensuing dialogue is demonstrated.

OBJECTS
/* The goal of the tax return form is described. It consists of filing */
/* information, income information, tax payment information and tax */
/* preparer information. */
GOAL: (Filing_Info Income_Info Payments_Info Preparer_Info);
TEXT: "US Tax Return Form";

OBJECT: Name
TYPE : string;
TEXT : "Enter name";

OBJECT: Soc_Security
TYPE : string;
TEXT : "Enter social security number";

OBJECT: Address
TYPE : string;
TEXT : "Enter address";

OBJECT: MStatus
TYPE : enumerated;
RANGE : (Married Single Head_of_Household);
TEXT : "Enter marital status";

OBJECT: Dependents
TYPE : list;
VALUE : (...ATLEAST(0) Name);
TEXT : "Dependent Children";

OBJECT: Filing_Info
TYPE : list;
VALUE : (ALLOF Name Soc_Security Address MStatus Dependents);
OBJECT: Wages
TYPE : integer;
TEXT : "Enter wages";

OBJECT: Amount
TYPE : integer;
TEXT : "Enter dividend amount";

OBJECT: Dividends
TYPE : list;
VALUE : (ALLOF Name Amount);

OBJECT: Optional_Dividends
TYPE : list;
VALUE : (ATLEAST(0) Dividends);
TEXT : "Dividends";

OBJECT: Total_Dividends
TYPE : integer;
STATUS: display := SHOW;
TEXT : "Total Dividends";

OBJECT: Exclusions
TYPE : integer;
TEXT : "Enter exclusion amount";

OBJECT: Total_Gross
TYPE : integer;
STATUS: DISPLAY := SHOW;
TEXT : "Total Gross Adjusting Income";

OBJECT: Income_Info
TYPE : list;
VALUE : (ALLOF Wages Optional_Dividends Total_Dividends Exclusions Total_Gross);
TEXT : "Income information";
OBJECT: Tax_Withheld
TYPE: integer;
TEXT: "Federal Income Tax withheld";

OBJECT: Tax_Owed
TYPE: integer;
TEXT: "Federal Income Tax owed";

OBJECT: Pay
TYPE: integer;
STATUS: display := SHOW;
text: "Amount to pay";

OBJECT: Receive
TYPE: integer;
STATUS: display := SHOW;
text: "Amount Overpayed";

OBJECT: Pay_Receive
TYPE: list;
VALUE: (EXACTLY(1) Pay Receive);
TEXT: "Total Tax";

OBJECT: Payments_Info
TYPE: list;
VALUE: (ALLOF Tax_Withheld Tax_Owed Pay_Receive);
TEXT: "Payments Information";

OBJECT: Paid_Preparer
TYPE: list;
VALUE: (ALLOF Name Address);
TEXT: "Paid Preparer information";

OBJECT: Preparer
TYPE: enumerated;
RANGE: (Self Paid_Preparer);
TEXT: "Who prepared this tax form?";
OBJECT: Preparer_Info
TYPE : list;
VALUE : (ALLOF Preparer Paid_Preparer);
TEXT : "Preparer Information";

RULES
RULE:
/*This rule states that if the filer is single there is no need to */
/*to ask about thier spouse. It also states that if the form was */
/*prepared by the filer himself, there is no need to ask about the */
/*preparers name and address. */
BEGIN
IF Filing_Info.MStatus.value = "Single" THEN
    Filing_Info.Dependents.enabled = DISABLED;
ELSE
    Filing_Info.Dependents.enabled = ENABLED;
END.

IF Preparer_Info.Preparer.value = "Self" THEN
    Preparer_Info.Paid_Preparer.enabled = DISABLED;
ELSE
    Preparer_Info.Paid_Preparer.enabled = ENABLED
END.

RULE:
/*This rule adds the dividends entered by the user and places their */
/*sum into the Total_Dividend object. */
VAR
    sum : INTEGER;
    counter : INTEGER;
BEGIN
    sum := 0;
    counter := 0;
    WHILE counter < NUMBER_OF_COPIES(Income_Info.Optional_Dividends.Amount) DO
        BEGIN
            counter := counter+1;
            sum := sum + Income_Info.Optional_Dividends.Amount[counter].value;
        END;
    Income_Info.Total_Dividends.value := sum;
END.
RULE:
/*This rule sums the wages, dividends and exclusions in order to */
/*obtain the total gross income. */
VAR
  sum : INTEGER;
BEGIN
  sum := 0;

  IF Payments_Info.Wages.selected = SELECTED THEN
    sum := sum + Income_Info.Wages.value;
  END

  IF Payments_Info.Optional_Dividends.selected = SELECTED THEN
    sum := sum + Income_Info.Total_Dividends.value;
  END

  IF Payments_Info.Exclusions.selected = SELECTED THEN
    sum := sum - Income_Info.Exclusions.value;
  END

  Payments_Info.Total_Gross.value := sum;
END.

RULE:
/*This rule states that if the preparer overpaid his taxes he should*/
/*receive a refund, otherwise he owes taxes. It also computes the */
/*amount of the taxes or the refund. */
VAR
  x : INTEGER;
BEGIN
  x := Payments_Info.Tax_Owed.value - Payments_Info.Tax_Withheld.value;

  IF x > 0 THEN
    BEGIN
      Payments_Info.Pay_Receive.Pay.enabled := ENABLED;
      Payments_Info.Pay_Receive.Receive.enabled := DISABLED;
      Payments_Info.Pay_Receive.Pay.value := x;
    END
  ELSE
    BEGIN
      Payments_Info.Pay_Receive.Pay.enabled := DISABLED;
      Payments_Info.Pay_Receive.Receive.enabled := ENABLED;
      Payments_Info.Pay_Receive.Pay.value := 0 - x;
    END
  END
END.
CONSTRANTS

/* This constraint that the preparer is not allowed to exclude more */
/* than the amount of his wages. */

CONSTRANT: Income_Info.Wages.value > Income_Info.Exclusions.value;
text: "Illegal to exclude more than your wages";

This example demonstrates many of the features of the dialogue definition language. The first and third rules are examples of how the optimizations of Section 3.4 can save questions. Without optimizations, in order to run the first rule both the objects MStatus and Preparer must have obtained input. The optimizations allow the rule to be partially evaluated even if only one of the objects received a value.

In order to execute the third rule without optimizations, all of the objects Wages, Total_Dividends, and Exclusions must have values before the rule can be executed and Total_Gross be assigned a value. The optimizations allow Total_Gross to be assigned a value as soon as any subset of those three objects have values.

The constraint demonstrates the use of constraints in defining dialogues. The constraint does not cause any value changes but validates the user input and makes sure that certain semantic constraints are kept.

An example of a screen of 5 objects containing the above example:

```
[US Tax Return Form]
  ▶ Filing Information [All of the following] (more...) -
    ▶ Enter Name:
    ▶ Income Information (more...) -
    ▶ Payments Information (more...) -
    ▶ Preparer Information (more...) -
```

The selection of objects to display is by the algorithm of Section 3.3.3. The user can enter values into external objects (e.g. Enter Name:) or select an internal object (e.g. Income Information) in order to see more objects. The top row of buttons allow the user to navigate in the dialogue as described in Section 3.3.5 or terminate the dialogue.
3.7.1 Actual Screens

This section shows actual screen displays of the previous example.

Figure 26. Initial screen for the dialogue of the example of Section 3.7.
Figure 27. The user moves mouse in order to select field.

Figure 28. User selection of a field. The user selected the field marked with a check and then selected the continuation button.
Figure 29. Next screen after user selection of a the continuation field. The user moves the mouse in order to select from an enumerated range.

Figure 30. The user selects a value from the enumerated range.
Figure 31. Rule 1 is used to deduce that Paid_Preparer is no longer needed. The user moves the mouse in order to select the Up button.

Figure 32. After the user requests to see the context (Up button). The user moves the mouse in order to select Income Information. Notice that the Preparer field is marked completed.
Figure 33. The user selected the field and moves the mouse in order to continue the dialogue.

Figure 34. Fields under the new context. The user moves the mouse in order to select the Dividends field.
Figure 35. The user enters his wages and chooses to enter dividends.

Figure 36. The user enters dividends and requests to return to the defining context.
Figure 37. Total dividends has been automatically computed by rule 2.

Figure 38. The user returns to the defining context.
4.0 Conclusions and Further Research

This thesis investigated the division of interactive systems into application and dialogue components. The focal point of the thesis was how to design a dialogue manager for large dialogues so as to bother the user as little as possible. A framework for defining precisely what is meant by bother is used to propose the formal definition of the term efficient dialogue. An efficient dialogue is defined as a dialogue which displays a large amount of information on the screen while keeping the information clear and in context, and by abstaining from redundant questions. The model describes the division of a man-machine dialogue into several subtasks:

1. dialogue definition language
2. application to dialogue manager interface
3. dialogue manager
   a. dialogue logic
   b. screen manager

The application to dialogue manager interface can actually be considered a compilation of the dialogue definition language which is then executed by the dialogue manager in order to obtain input from the user.

A prototype dialogue management system was constructed to implement these ideas. The dialogue designer uses the dialogue definition language to give a high-level description of what is required from the user. The interaction is then automatically created from this description.

The dialogue definition language is a non-procedural structured language based on objects, and relations and constraints between objects. After the dialogue designer defines the dialogue using this language together with
the maximum number of objects allowed on one screen, the ensuing interaction is automatically created by the prototype.

The prototype was implemented in C to ensure portability. The screen manager component was implemented on a SUN workstation in order to take advantage of the display tools available.

The following theoretical problems arising from the need for the dialogue to be efficient were also studied. Part II of the thesis contains the definitions of the problems and the solutions suggested.

1. The first problem addressed was that of the representation of hierarchy between objects. In order to obtain a dense representation, the hierarchy was represented as a directed acyclic graph, as opposed to the usual tree representation. The problem of selecting objects from the graph for display was discussed.

   Graph theoretical techniques were used to aid with the exact formulation of the problem. This formulation was found to be NP-Hard in general and without solution for certain graph and screen sizes. Therefore an approximation of the problem was defined and a computationally efficient algorithm found.

2. The second theoretical problem addressed was that of the rules used to automatically compute object values based on known values. The ordering of the rules was not investigated, since that is known to be a combinatorically hard problem. Instead, customizing the set of rules based on values obtained from the user and the integration of rules into the object selection algorithm was investigated.

   In order to customize the rules, a PASCAL-like language was used for rule definition. This allowed classic compiler techniques to be used and augmented. The classic dead code elimination and constant propagation heuristics were extended to include loop execution.

   An alternative approach could use a declarative language, but the customization techniques would be different and such languages are less popular in the application oriented community. Future investigation could focus on adapting the system to other language types, e.g. a declarative PROLOG-like language.
Another problem of theoretical merit has to do with the fact that the interaction between rules, objects and user input defines a large number of possible screens without the dialogue designer having to explicitly define those screens. We have neglected to investigate exactly how many screens are possible. This measure could be used to compare the dialogue definition language presented here with other methods, e.g. graphs.

3. The third theoretical problem addressed is that of constraints and user errors arising from the constraints.

Instead of using simple backtracking, a time-based backtracking heuristic was proposed. The premise for the choice was that the user prefers his latest replies and the mistake is actually in earlier answers relating to the contradicted constraint. This, of course, in no way restricts error correction since as the user confirms his older answers they become more recent and the new answers become older.

4. The final problem addressed is that of actual screen display. This problem had to be solved in order to implement a working prototype. Since human factors research was beyond the scope of this thesis, common sense choices were made about the actual display. However, the modularity of the system allows display techniques to be easily altered.

Human factors research on whether the backtracking (problem 3) and display techniques (problem 4) implemented are convenient for the user would be an interesting extension of the thesis.
5.0 Bibliography


[ISPF] Interactive System Productivity Facility Users Guide, *IBM Publication #SH20-9245-0*


[LEX] M. E. Lesk, E. Schmidt, Lex - A Lexical Analyzer Generator, *Unix documentation*


84


[SHN87] B. Shneiderman, Designing the User Interface, Addison-Wesley Publishing Company, 1987


[TOP] TopView Programmer’s Toolkit, IBM Publication #G509-2399


[YACC] S. Johnson, Yacc: Yet Another Compiler Compiler, Unix documentation

Bibliography 85