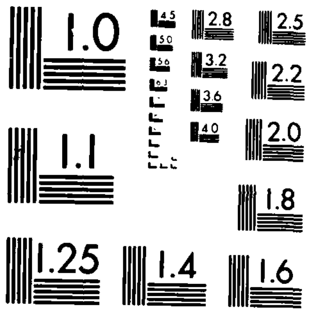
AD-A152 215 AN AUTOMATED/INTERACTIVE SOFTWARE ENGINEERING TOOL TO GENERATE DATA DICTIONARIES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. C W THOMAS 1/4

UNCLASSIFIED DEC 82 AFIT/GCS/ENG/84D-29 F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

AN AUTOMATED/INTERACTIVE SOFTWARE ENGINEERING TOOL TO GENERATE DATA DICTIONARIES

THESIS

Charles W. Thomas
Captain, USAF

AFIT/GCS/ENG/84D-29

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

(1)

AFIT/GCS/ENG/84D-29

AN AUTOMATED/INTERACTIVE SOFTWARE ENGINEERING TOOL TO GENERATE DATA DICTIONARIES

THESIS

Charles W. Thomas
Captain, USAF

AFIT/GCS/ENG/84D-29

AN AUTOMATED/INTERACTIVE SOFTWARE ENGINEERING TOOL TO GENERATE DATA DICTIONARIES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science

Charles W. Thomas, B.S.
Captain, USAF

December 1982

PREFACE

This report is the result of my efforts to accomplish a design and initial implementation of an automated and interactive software engineering tool which generates data dictionaries. The resulting implementation of this thesis investigation is an interactive data dictionary generation tool which accepts and maintains data dictionary information in support of three methods of software representation; SADT, structure charts, and code. A fourth software representation, data flow diagrams, is not supported in the initial implementation of the tool, however, all necessary design work for its inclusion in the tool was accomplished. This initial implementation of the data dictionary generation tool represents only a partial realization of the potential of a fully automated data dictionary generation tool. The last chapter of this report contains recommendations for future development of this tool.

I wish to express my sincere appreciation to Dr Gary B. Lamont, the advisor of this investigation, for his guidance and insight throughout the duration of this effort and Dr Thomas C. Hartrum for his assistance in the development and testing and evaluation of this tool. I also wish to thank Captain Pat Lawlis for serving on the thesis committee for this investigation.

Table of Contents

List of Figures

AFIT/GCS/ENG/84D-29

Abstract

The purpose of this investigation is to design and develop an automated/interactive software engineering tool which generates data dictionaries. This tool is provide the user with an interactive data dictionary tool to support the development of software in all phases of the software life cycle. The tool supports data dictionary information for specific methods of software representation. The initial implementation of this tool supported four methods of software representation: SADT, data flow diagrams, structure charts, and code. The requirements definition for the tool includes a discussion of the objectives and concerns associated with the tool development. Data flow diagrams are used to formulate a requirements model. The preliminary design specifies the type of information to be contained in the data dictionary for each of the methods of software representation supported and database design required to maintain the data dictionary information. The structural framework of the application software which provides the interface between the tool user and the dictionary database is specified and structure charts are used to model this structural framework. In detailed design, algorithms are developed for the tool's application software.

The dictionary database is implemented through the use of the INGRES database management system. The application software is .oded using the C programming language. The application software interfaces with the dictionary database by means of embedded EQUEL (INGRES Embedded Query Language) statements in the C language source code. The tool was implemented on the VAX 11/780 computer using the UNIX operating system.

AN AUTOMATED/INTERACTIVE SOFTWARE ENGINEERING TOOL TO GENERATE DATA DICTIONARIES

I. Introduction

Thesis Objective

The objective of this thesis investigation is to design and develop an automated/interactive software engineering tool which translates the information contained in graphical software engineering techniques into data dictionaries. This tool will employ a combination of user interaction and machine analysis to extract the required data dictionary information from the graphical representation. During machine analysis the tool will attempt to detect any specification errors detected in the graphical representation and bring them to the attention of the user. This tool will support the development of software through all phases of the software life cycle.

Background

Advances in both the application and affordability of electronic computer systems have greatly increased the demand for reliable, cost efficient, and maintainable software. Unfortunately, our inability to produce quality software products in a timely and cost effective manner has led to a situation which many people refer to as the software crisis.

For the computational power of the computer to be fully realized, we must place increased emphasis on improving the methods and tools used in software production.

> The problem of the 1980s is different. Now we must reduce the cost of electronic solutions; that is, reducing the cost you incur in using our device to build a product. Solving this problem will require a shift from the component integration of the 1970s to concentration on system level integration in the 1980s.
>
> We can now talk about putting the power of a mainframe CPU on a single chip. This buys you nothing as acustomer, however unless you can use that power. Hardware is commputing potential; it must be harnessed and driven by software to be useful. (1:22)

Software production problems can best be resolved by considering software development from a life cycle point of view. "The complexity of a large software system surpasses the comprehension of any one individual. To better control the development of a project, software managers have identified six separate stages through which a software project pass; these stages are collectively called the software development life cycle" (2:198).

The definition and nomenclature of the six phases which constitute the software life cycle vary from author to author. For the purpose of this investigation, the following is used to define the phases of the software life cycle: requirements definition, preliminary design, detailed Design, implementation,integration, and maintenance. Most versions of the life cycle include a phase which is dedicated to testing, validation, and verification. In this paper, testing, validation, and verification will be considered an integral part of all phases of the life cycle rather than as a separate and distinct phase.

The purpose of the requirements definition phase is to clearly define exactly what the proposed software project is to accomplish for the user. The primary emphasis during this phase is to define as precisely as possible the exact function or functions the software project is to perform. This life cycle phase will require a great deal of interaction between the ultimate user of the software and the software designer.

During the preliminary design phase, the information obtained during the requirements definition phase is used to determine the structure and framework of the software. "The preliminary design step is an attempt to develope software beginning from the top down. Information flow or structure, determined from requirements, becomes a tool that leads to an overall representation of software" (1:132). In this phase, the software project is broken down into modules which represent particular functions. These functional modules are further decomposed into sub-functions to obtain a hierarchial representation of the software project.

During the detailed design phase, the functions defined in the preliminary design phase are further detailed and decomposed . The functional modules are converted into specific algorithms which perform the function. "Detailed design provides a blueprint for coding. With the use of a design representation that may be graphical, tabular, or textual, a detailed procedural specification for the software is created. Like the blueprint, the detailed design specification should provide sufficient information for someone other than the designer to develop resultant source code" (1:133).

The implementation phase represents the actual coding of the software. Utilizing the detailed design information, the software project is translated into a particular programming language.

During the integration phase, the project software is installed on the target hardware. Extensive testing will take place during this phase to ensure that the software meets all specified requirements.

The maintenance phase of the life cycle consists of activities involved in the actual use of the software. These activities include the detection and correction of any errors and the modification of the software to meet any changing user requirements.

In order to support the development of software through the various stages of the life cycle, numerous software engineering tools and methodologies have been developed.

The aim of these efforts is to improve our ability to produce cost efficient and reliable software products and to help control the software crisis.

While there are many existing software engineering tools and methodologies in use, four in particular have gained widespread use within the Defense software community: Structured Analysis Design Technique (SADT), data flow diagrams, structure charts, and data dictionaries. The first three tools listed above are graphical techniques used in the requirements definition and design phases of the software life cycle.

SADT

"SADT (a trademark of Softech, Inc.) is a systems analysis and design technique that has been widely used as a tool for system definition, software requirements analysis, and system and software design" (1:120). SADTs consist of a graphical representation of the software project which enhances the analysis and communications process which is so critical during the requirements definition and the design phases of a software project. "SADT is a technique that enables people to understand complex systems in a complete and precise manner, and enables them to communicate their understanding" (3:A-2). The application of the SADT technique results in a model which describes what functions a system must perform, specifies how a system is to be designed and constructed, and explains how a system is to be used and maintained.

The SADT model consists of a series of diagrams that decompose a complex problem into its component parts. The initial diagram will present a general or abstract description of the problem. Subsequent diagrams will decompose the problem into smaller less complex components. As the decomposition process continues, the level of detail illustrated by the diagrams will increase. This iterative process will continue until a level of detail is reached where further decompostion is not possible.

The SADT diagram consists of boxes and arrows which illustrate the components of a system and their relationship to one another. "The notation employed is simple: boxes describe functions and arrows describe interfaces between functions. Diagrams, composed of boxes and arrows are used as the framework for expr ssing whole units of a system" (4:31).

The direction of the arrows and the point of attachment of the arrows to the box have a specific meaning in the semantics of the SADT diagram (see figure 1). "If a box represents an activity, then input data (on the left) are transformed into output data (on the right). Controls (on the top) govern the way the transformation is done. Mechanisms (on the bottom) indicate the means by which the activity is performed. A "mechanism" might be a person or a committee or a machine or a process" (3:A5).

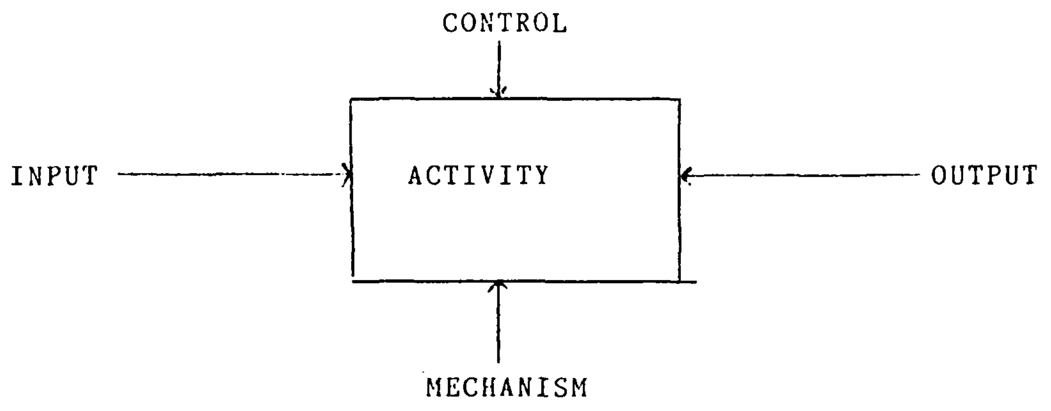


Figure 1. SADT Activity Diagram

"SADT diagrams show both the things (objects or data) *and* the *happenings* (functions or activities) in a system" (3:A2). Two separate types of diagrams are used in the SADT methodology. The activity diagram uses the boxes to represent activities and the labeled arrows constitute the input data, output data, control information, and mechanisms. The data diagram uses the box to represent a data item and the labeled arrows to represent acitvities involving the data.

Structure Charts and Data Flow Diagrams

Structure charts are a graphical representation of the sub-functions or modules of a software system and their relationship to one another. "Structure charts were originally developed by Constantine et al; [6] to specify modular characteristics of software during design" (5:1087.

The graphics used in structure charts provide a clear picture of the interaction between modules and the basic structure of the software system (see figure 2). "Structure charts can be drawn in several different ways. The approach proposed by Constantine utilizes three basic graphical forms

1(the rectangle, used to contain a module or module descriptor;
2(the vector, used to highlight interaction between modules (usually a call);
3(the arrow with a circular tail, used to depict transfer of data and control between modules" (5:1087-1088).

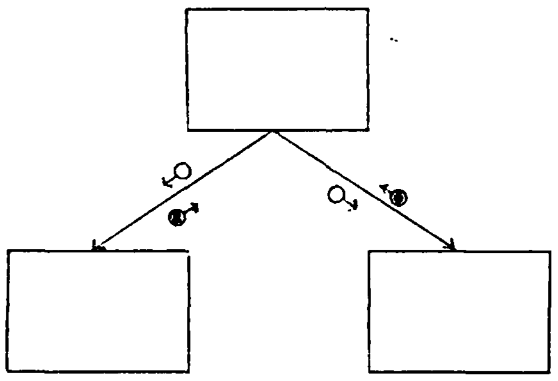


Figure 2. Structure Chart Diagram

In structure chart notation, the arrow whose circular tail is filled in ( ) represents the transfer of control information. If the circular tail of an arrow is open (not filled in), it represents the transfer of data between modules.

A data flow diagram is a graphical representation which depicts the information flows and the transforms that are applied to data in a software system. Data diagrams are also called bubble charts or data flow graphs (1:101). A data flow diagram consists of a series of circles interconnected with vectors (see see figure 3). The circles or bubbles represent functions or transforms which act upon incoming data, represented by incoming vectors, and produces output represented by output vectors.

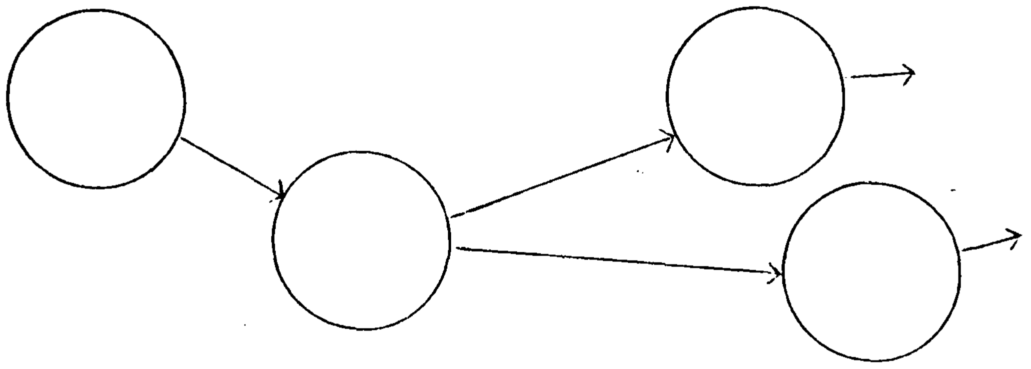


Figure 3. ·Data Flow Diagrams

A fundamental system model can be represented as a single bubble with input and output data. This initial diagram can be refined in a series of bubbles. "Each transform in the diagram (bubbles) could be refined still further to provide greater detail.... That is, the diagram may be layered to show any desired level of detail" (1:101).

When using data flow diagrams as a tool in the requirements specification and design phases of software development, two important factors should be kept in mind. "Since movement and transformation of data are the only characteristics represented by data flow diagrams, the concept of the passage of time along any single or several data flow path(s) is not present" (5:1090-1091). The other factor of importance is that the decomposition process produces a network of programs rather than a hierarchy of programs (6:23).

Both structure charts and data flow diagrams are utilized in a software design technique known as data flow design method. "The data flow design method was first proposed by Larry Constantine (Reference 2) and has since been propogated and extended by Ed Yourdon and Glen Myers (References 2,3). It has been called by several different names including Transform Centered Design and Composite Design" (7:305).

The data flow design method is based upon the functional decomposition of a software with respect to data flows. The data flow diagram is used to help the designer show the flows and transformations of data through the system. The data flow diagrams are then partitioned into three different types of transforms : efferent, afferent, and central.

The afferent transforms represent the input and are concerned with accepting and developing the system's input. The efferent transforms are concerned with delivery of the systems output data. "The central transform is the portion of the system DFD that contains the essential functions of the system and is independent of the particular implementation of the input and output" (9:226). The identification of these three portions of the data flow diagram leads to a hierarchial decomposition of modules which can be more effectively depicted using structure charts. The designer will iterate between the data flow diagram and structure chart representation of the project in order to decompose the system into smaller more manageable pieces.

Data Dictionaries

SADTs, data flow diagrams, and structure charts graphically illustrate the functional structure and flow of information or data in a software system. The information portrayed on these graphical representation tools can be used to determine the general content and some of the detailed information contained in a data dictionary. In turn, the data dictionary provides the definition and composition of items illustrated in the graphical representation techniques. "Basically, the use of a data dictionary is an attempt to capture, and store in a central location all definitions of data within an enterprise and some of their attributes, for the purpose of controlling how data is used and created and to improve the documentation of the total collection of data on which an enterprise depends" (11:1.1).

A data dictionary consists of dictionary entities and their attributes. An entity can be generally placed into one of the following three categories:

a. A data entity, such as a data item, group, file, etc., and among its attributes may be user names, system name, picture, description, etc.

b. A processing entity, such as a module, program, system, etc., and attributes may include name, description, programming language, etc.

c. A useage entity, such as a person, department, terminal, etc., and attributes may include name, security attributes. (11:1.3)

The use of data dictionaries can help reduce the rapidly growing costs associated with the documentation and maintenance of software systems. Software experts estimate that approximately 65% of the cost associated with software systems occur during the maintenance phase of the life cycle (2:201).

A data dictionary is an important tool during the maintenance phase of the software life cycle. However, a data dictionary can contribute significantly to all phases of the software life cycle.

> Using a DDS (Data Dictionary System) provides economic and technical benefits. A DSS may provide immediate savings, or it may facilitate a continuing technical process by making it easier or more reliable to perform. To summarize the benefits:
>
> Better control of the organization's data resources

through improved (i.e., centralized, rigorous, and standardized) data definitions, data handling and data collection.

Improved transportability of data and software between computing environments through standardized data elements and data definitions.

Improved documentation for databases, programs, and systems.

Automatic compilation of data definitions to be included in application programs or in DBMS database definition.

Increased security and access control for the database environent.

Effective aid to software development, modification, and maintenance through configuration management of system components of data and programs.

Increased cost effective use of data resources throughout the system development life cycle. (12:9)

The degree to which a data dictionary system can provide the benefits listed above is largely dependent upon the type of data dictionary system used and its level of integration into the databases and system software of the organization.

There are basically two methods of classifying data dictionaries. One classification is based upon the capability of the data dictionary system to provide data entity descriptions to other software. The second classification method is concerned with the dependence of the data dictionary system on other software for performing its functions.

When examining the capabiltiy to provide data entity descriptions to other software, a data dictionary can be classified as either passive or active. "A passive DDS is an information tool that is only accessed by personnel, to enter or retrieve entity descriptions. With a passive DDS, descriptions of the same data will exist concurrently in other software such as COBOL programs. Changes in DSS content do not automatically produce corresponding changes in the other data descriptions, and vice versa" (12:6). A passive DDS will serve as an aid to manual procedures for controlling data, but will not directly control an organization's data descriptions.

"An active DDS, through software interfaces and computer operating procedures, provides the ONLY source for data descriptions to other processing components such as compilers, assemblers, and DBMSs. The active DSS assists in the enforcement of data standards and usage throughout the organization and its computer applications" (12:6).

The dependence of a data dictionary system on other software can be classified as either stand-alone or dependent. "A stand alone DDS is self-contained; that is , its functions are performed without relying on any other general prupose software such as a DBMS" (12:7). "A dependent DDS is specifically tailored to operate in conjunction with another general purpose software system, usually a DBMS. It requires the DBMS facilities to perform DSS functions. In some cases, the dependent DSS is implemented as an application under a DBMS, wholly using

DBMS facilities" (12:7).

The software engineering tools discussed: SADT, data flow diagrams, structure charts, and data dictionaries provide valuable support to the software engineer during the various phases of the software life cycle. The automation of these tools can help relieve the burden of the many tedious tasks associated with applying these tools to software projects. Although automation improves the ease of use and effectiveness of software engineering tools, the application of the tools in a software development environment can provide increased flexibility and efficiency in the performance of software development and maintenance tasks.

"A software development environment is a collection and integration of automated software development tools that should adequately support the entire software life cycle" (13:9). An example of a software development environment is the Software Development Workbench (SDW) developed at the Air Force Institute of Technology (13:ix). The SDW constitutes a continuing research effort to provide support in the development of software products by providing integrated and automated software engineering tools to enhance software development in all phases of the life cycle.

Problem Statement

The purpose of this study is to design and implement an automated tool to generate data dictionaries from graphical software engineering tools such as data flow diagrams, structure charts, and SADTS. This tool will interpret the graphical software representation and interact with the user in order to obtain the necessary information for the data dictionary. The tool will attempt to identify any specification errors and bring them to the attention of the user.

The manual generation of data dictionaries is a tedious and time consuming task. There exists a need for an automated and interactive software engineering tool which generates data dictionaries with a minimum amount of user interaction.

Scope of the Thesis Investigation

This project will only be concerned with generating data dictionary information for the following software engineering graphical tools: SADTs, data flow diagrams, and structure charts.

Approach

The initial step in the project will be to perform an extensive literature search on the problem. This review of current information will have four primary goals:

1. Gain a thorough understanding of how SADTs, data flow diagrams, and structure charts graphically represent data elements and data flows.

2. Evaluate current data dictionary systems in order to determine an appropriate format for the data dictionaries generated by the proposed tool.

3. Understand the process for generating data dictionaries from SADTs, structure charts, and data flow diagrams.

4. Study existing automated data dictionary systems.

Utilizing the information obtained during the literature search, the process of generating data dictionaries will be modeled using a graphical representation technique. Emphasis will be placed on identifying those tasks in the process which can be performed more efficiently with the machine and those tasks which will require user interaction.

The next phase of the project will be to utilize a model to formulate a requirements definition for the automated tool. During this phase, the primary goals and objectives of the tool will be specified. The primary concern during this phase will be to clearly delineate exactly what the automated data dictionary generation tool is to accomplish for the user.

Using the requirements definition, the next step is to perform the preliminary design of the tool. During this phase, the component structure and framework of the tool will be determined. The sub functions or modules needed to meet the requirements of the tool will be identified.

The next stage involves the detailed design of the tool. During this phase, specific algorithms or procedures will be developed to perform the functions identified in the preliminary design phase. A test plan for the software associated with the tool will be designed.

The algorithms developed during the detailed design phase will be translated into an appropriate programming language during the implementation phase. Upon completion of this task, the tool will be loaded onto the target computer for testing and integration. The implementation of the tool will be followed by extensive testing to ensure that the tool meets all requirements specificiation.

II. Requirements Definition

Introduction

The Requirement's Definition is a clear statement of the goals and objectives of the proposed software system. This phase of software developement requires a great deal of interaction between the software devoloper and the software user. The software user will attempt to describe the functions or capabilities he or she expects the proposed software to provide. The software developer will take the user's concept and attempt to translate it into specific functional and performance objectives.

During the Requirements Definition phase, emphasis is placed on defining as precisely as possible the exact function or functions the proposed software is to perform in support of the user. In order for this software development step to be successful, the user and developer must be able to effectively communicate with one another. In developing software, the possibility for misunderstanding and misinterpretation is extremely high. "The dilemma that confronts a software engineer may best be understood by repeating the statement of an anonymous (infamous?) requester: "I know you believe you understood what you think I said, but I am not sure you realize that what you heard is not what I meant..."(1:94)

The primary component of the requirements definition document is a functional and/or data model of the proposed system. This model performs a dual role. It provides the developer with an excellent tool for defining the functional/data specifications for the system and enhances the communications process between the user and the developer.

In addition to the model, the requirements definition may also include a description of the fundamental concerns, constraints, and objectives that will guide the developement of the system. The Requirements Definition Document should contain a set of evaluation parameters and criteria. This will assist in eventually testing the system to ensure that it meets all specified requirements.

The purpose of this chapter is to develop the system requirements for an automated/interactive software engineering tool which translates the information contained in graphical software engineering techniques into data dictionaries. Initially a group of objectives and concerns fundamental to development of the tool will be listed and explained. With this background, the requirements for the tool will be defined. A two dimensional graphics technique, data flow diagrams, will be used for defining and describing system requirements. Finally a set of evaluation parameters and criteria is established to aid in testing the software to ensure the system meets specified requirements.

Objectives and Concerns

Before the functional model of the data dictionary generation tool was developed, an extensive literature search was conducted to identify objectives and concerns related to the design, developement, and use of data dictionaries as software engineering tools.

Data dictionaries are becoming recognized as an important tool in the management of an enterprise's data resources. "Corporate management is becoming aware of an important asset which, until recently, has been virtually ignored. The asset is data.... The idea of data being a corporate asset is relatively new, and has developed along with the influence of computers in business. The capacity of computer storage devices for holding data has increased and the relative cost of these devices has decreased" (14:118).

The recognition of the importance of data as an organizational resource has led to the development of data dictionary systems with a wide range of capabilities and features. The objectives of a data dictionary system depend upon the types of activities it supports. The objectives of the automated/interactive data dictionary generating tool will not only consist of those for the resulting data dictionary, but also the objectives of the portion of the tool which extracts data dicitonary information from the software representation method. The objectives and concerns regarding the data dictionary generation tool are described and listed in the following paragraphs.

Support All Phases of the Software Life Cycle. A primary objective of this tool will be to provide improved support for all phases of the software life cycle. The data dictionary generated by this tool can be used as the sole source for metadata (information about data) through all phases of the software life cycle. To better define objectives, each life cycle phase and the support it can receive from a data dictionary system will be described individually.

Support Requirement's Definition Phase.

"The use of the DD/DS (Data Dictionary/Directory System) in requirement's definition and analysis is critical. The DD/DS provides a framework in which the end user and analyst can communicate with each other using common terminology and definitions" (15:34). As discussed earlier, miscommunication between the user and designer can cause serious problems in any software project. By maintaining consistency in the data used, a data dictionary system can aid in averting potentially disastrous conditions caused by inexact or inconsistent data.

The data dictionary is also used in the requirement's definition phase to document requirements as they are defined and to support their analysis. The data dictionary records descriptions of processes, information about the operation of processes, potential uses of the processes, and the data elements required by the processes. It also contains information about the relationship between different processes and data elements.

Once data requirements are defined, it is necessary to determine how much of the data is currently available in the data resource inventory. This will help ensure that unnecessary redundancy is not introduced. "Another step in the analysis is to determine if the requirements can be satisfied by modifying existing data. The assistance of DD/DS is invaluable at this stage, especially if the DD/DS already has a complete inventory of the enterprise's data. This is further supported when the data defined in the DD/DS has common definitions which can facilitate the analysis process" (15:44).

Support Preliminary and Detailed Design Phases.

Design specifications require information about data or metadata. "Recording these metadata in the DD/DS is very useful because the DD/DS can provide a means for maintaining control over the system design specifications and can aid in insuring that requirements stated earlier are consistent with the implementation. This can be accomplished at the common denominator between the "what" and the "how", which is the data element" (15:46).

The data dictionary is a valuable tool in performing both system and database design. The data dictionary can be used for storing the descriptions of system components such as program modules, subsystems, data flows, and data structures. The descriptions will contain such information as functional characteristics, interaction between different components, and the data components require for operation.

"Database design involves describing the data required by the programs, beginning with previously developed definitions of the data elements, records, and descriptions of storage structures and access strategies. These are used to generate a desired data structure or schema for the database. Also from these descriptions, the programs view of the data, or subschema can be generated" (15:47). The data dictionary's ability to store these descriptions make it a valuable aid in the database design process.

Support Implementation Phase.

"Metadata about the program and about the data can be retrieved from the DD/DS to help in the programming task. Pertinent metadata retrieved from the DD/DS can be incorporated directly into the programs being coded as the the data definition block" (15:47).

Support the Integration Phase.

During the integration phase of the software life cycle, the testing and validation of the software project is an important step. The data dictionary can aid in this effort. "Use of metadata can be extended to testing and validation. Once the characteristics of the database are recorded, it would be easier and possibly more reliable to generate test data using metadata recorded in the DD/DS" (15:48).

Support Operations and Maintenance Phase.

The biggest contribution of a data dictionary system to the operation and maintenance phase of the software life cycle is as a tool for documentation and standards. Software documentation is a serious and costly problem for all organizations. "The DD/DS is one tool which can be used to overcome these difficulties by automatically producing documentation about the database and the system. In this light, the DD/DS should be used routinely to augment current documentation efforts, and to supplant a large percentage (60% to 70%) of existing systems and data documentation requirements. When used in the normal course of development, the DD/DS can lessen the monotony and repetitiveness of the task of documenting, and it can assist in completing the system development effort on time, delivering an end product which is well documented" (15:50). Although documentation is a task which should be done during every phase of the life cycle, a lack of good documentation is disastrous when attempting to operate and maintain software. The data dictionary is also an invaluable aid in determining the effect of a software modification on both the system and database. Its utilization can help reduce both the time and money involved in software maintenance.

A data dictionary can aid in the enforcement and use of standards in an organization's data processing endeavors. The use of standards can help to promote the sharing of data resources in a controlled environment. "In the computing field, especially, the same terminology is often used to mean different things in different contexts. Thus in some cases, standards are necessary so that everyone uses the same data to mean the same thing" (15:52).

Data related standards can be grouped into one of two types, data definition and data format conformance (ref 15). "Data definition refers to a standard way of describing data" (15:51). As an example, a naming standard could consist of rules or conventionss for assigning names to data entities. This would allow all users in an enterprise to know that when a data element is used in programs, reports, and files that it means the same throughout the organization.

"Data format conformance is content related. It means that a data element,in addition to having the same name throughout the enterprise, also must conform to a common set of format rules for the data element to retain the same meaning. Moreover, these must be accepted throughout the enterprise" (15:51-52). For example, the data element date should have the same format throughout the enterprise and only that format should be allowed. Another example would be the use of codes in an organization. If a two letter code is an accepted representation of a state (ie SC, VA,OH) then that code must be accepted by the entire organization and no other codes should be accepted.

"The DD/DS can failitate the introduction and enforcement of such standards, via a set of editing rules to be included in the DD/DS. These editing rules can, in effect, edit and validate acceptable codes, so that nonconforming codes are not acceptable. The DD/DS can be used as both the promulgator and the enforcer for data standards" (15:52).

Data Dictionary Will Support Information About Entities, Relationships, and Attributes.

The data dictionary can be considered a database whose contents is information about data. The domain of a data dictionary database consists of entities and their attributes and relationships. The following definitions should clarify this concept.

"Entity - any named concept, object, person, event, process or quantity that is the subject of stored or collected data.

Relationship - a pre-determined ordering between pairs of entities.

Attribute - a property or characteristic of an entity"(16:8).

A data dictionary entity represents an object, person, process, etc. It is not the actual data that might exist in a file or database, but a representation of that data. For example, the enitity called "social security number" would not consist of an actual number such as "247-82-4457".

An attribute is a characteristic of an entity. An attribute for a data dictionary entity could, for example, be length. In the case of the social security number entity, the length attribute would be nine.

The relationship between entities indicates the structure or ordering that exists between different entities. For example, the entity "Payroll Record" may contain the entity social security number. Relationships, like entities, may also possess attributes which describe their characteristics. Figure 4 graphically illustrates the entity-attribute-relationship structure.

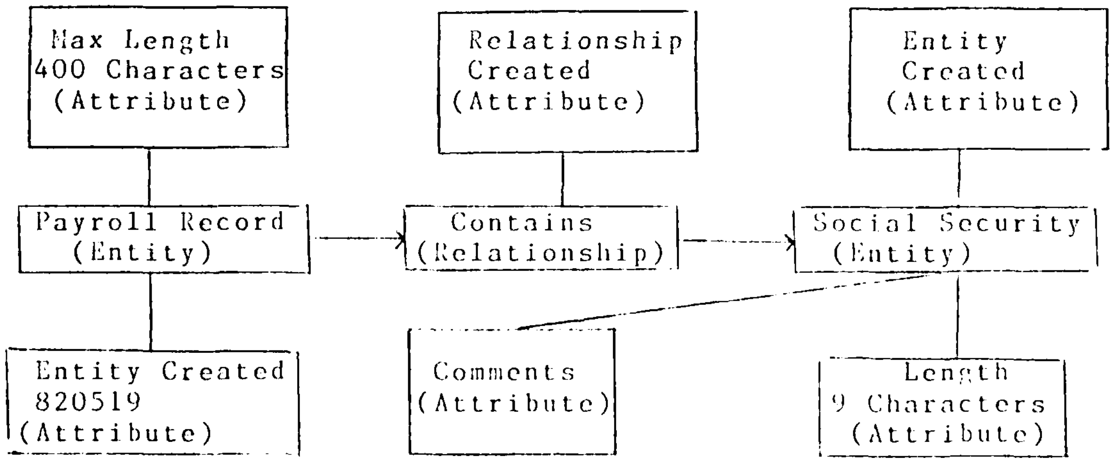


Figure 4. Entity, Attribute, Relationship Structure

"The basic unit in a data dictionary is the entity. Relationships connect pairs of these entities, and both entities and relationships have attributes assigned to them" (16:10).

Entities, relationships, and attributes can be organized into sets known as types. Attribute-types are organized so that each menber of a set represents a like characteristic. A typical attribute type could be "date created". In a similiar fashion, entities can be organized into entity-types. All examples of a specific entity-type would have similar or identical characteristics or attribute-types. Like entities and attributes, relationships can be grouped together to form relationship-types. Relationships which are examples of a particular relationship-type possess attributes from the collection of attribute-types associated with that relationship-type. Examples of relationships are system-contains-program and record-contains-element.

"These "types" form the basis of the dictionary schema --the collection of structures that describe the dictionary.... This entity-relationship-attribute construction used for the dictionary can be used to model the schema as well. Thus the DDS contains a "meta schema," or schema describing the schema (The concept of "meta" is defined as data about data.). At this "meta" level, the three concepts "entity-type", "relationship-type", and "attribute-type" are all "meta-entity-types". Instances of these concepts are "meta-entities which are conceptually connected by "meta-relationships". "Meta-attributes" can be associated with both the "meta-entities" and the "meta-relationships" " (16:12). Figure 5 gives examples of entries at each level.

SCHEMA MODEL LEVEL	SCHEMA LEVEL	DICTIONARY LEVELS
Typical Meta_Entity Types	Typical Entity-Types, Relationship-Types, and Attribute Types	Typical Entities Relationships, and Attributes
Entity Type	Element	Social Security Number Agency Name
	Record	Employee Record Payroll Record
	Document	Form 1040
Relationship_Type	Record-Contains Element	Payroll-Record Contain-Employee-Name
Attribute-Type	Length	9 Characters
	Creator	ADP Division

Figure 5. Examples Of Entity, Relationship, attribute Contructs.

<u>The Data Dictionary Will Support Data, Process, and User Entity Types.</u>

Data entities are a class of entities that describe or represent objects that are units or data or aggregates of data. The following list contains some specific classes of data-entity-types.

1. Element - describes instances of data belonging to an organization (Examples: social security number and agency name).

2. Document - describes instances of human readable data collections (Example: Form 1040).

3. Record - describes instances of logically associated data (Examples: Employee Record and Payroll Record).

4. File - describes instances of an organization's data collections (Examples: roster and accounts receivable). (16:14)

Process entities are a class of entities that represent processes and components that exist as part of the data processing environment. The following list contains some specific classes of process-entity-types.

1. System - A collection of programs (and indirectly modules) that can be associated with major functions of the organization (Examples: Personnel System and Supply System).

2. Module - A collection of processable code which is called by one or more programs and which may, in turn, call one or more other modules (Example: Sort Records).

3. Program - Describes instances of automated

processes (Example: roster update). (11:2.8-2.9)

User-entity-types describe members belonging to an organization who are responsible for data in the data dictionary. A useage entity can be a person, organization, terminal, or office.

The Data Dictionary Will Suport the Following Classes of Relationship-Types: Contains, Processes, Responsible-For, To, and Derived From.

The contains class of relationship-types describes instances of an entity being composed of other entities. "A typical CONTAINS relationship-type is RECORD-CONTAINS-ELEMENT, which has as a possible instance the relationship "Payroll-record-contains-employee-name"." (16:15)

The process class of relationship-types represents an association between data and process entity-types. "A typical PROCESSES relationship-type is SYSTEM-PROCESSES-FILE, which has as a possible instance the relationship "budget-system-processes-cost-center-file"." (16:15)

Associations between entities representing organizational components and other entities denoting organizational responsibilities are described by the responsible-for class of relationship entities. "A typical RESPONSIBLE-FOR relationship-type is USER--RESONSIBLE-FOR-DOCUMENT, which has as a possible instance the relationship "personnel-office-responsible-for-SF-171"."(16:16)

To describe associations between user and process entity-types, the runs class of relationship-types is used. It illustrates that a person or organization is responsible for running a certain process. "A typical RUNS relationship-type is USER-RUNS PROGRAM, which has as a possible instance the relationship "John-Doe-runs-system-backup"." (16:16)

The to class of relationship-types describes flow associations between process entity types. "A typical TO relationship-type is MODULE-TO-MODULE, which has as a possible instance the relationship "main-program-to-sort-routine," (indicating flow of control or data within a program)" (16:16).

The derived-from class of relationship-types describes associations between entities where the target entity is the result of a calculation involving a source entity. "A typical DERIVED-FROM relationship-type is DOCUMENT-DERIVED-FROM-FILE, which has as a possible instance "annual-report-derived-from-plans-file"."(16:16)

The Data Dictionary Should Support an Attribute-Type Which Indicates Software Life Cycle Phase of the Entity or Relationship Being Addressed.

The data dictionary generation tool will be used to support all phases of software development from requirements definition to operations and maintenance. To effectively perform its functions, the data dictionary must possess a means of determining the stage of development of the data and process descriptions it supports. This facility will help the designer keep better track as to the status of the project and will allow a means of tracing the evolution of a project component through the various development phases.

User Friendliness.

The data dictionary generation tool will be required to interact with the user in order to obtain complete descriptions for the data dictionary concerning the processes and data illustrated on the software representations. The user will also interact with the data dictionary when obtaining information about its contents. Failure to establish a freindly user interface which enhances communication could lead to the data dictionary containing erroneous or misleading information about the system's data and process entities.

The Data Dictionary Should Support User Defined Attributes.

The data dictionary will support a set of standard attributes in describing data and processes. However, by allowing the user to create his own attribute, the flexibility of the system will be greatly improved.

Error Checking.

When the tool extracts data dicitonary information from a software representation, it should check for errors in the representation and bring them to the attention of the user. Errors in format or meaning in the software representation could lead to the processing of erroneous or unclear descriptions in the data dictionary.

<u>The Data Dictionary Generation Tool Should be capable of Interpreting the Contents of Several Different Software Representations</u>.

There are currently numerous techniques and methodologies available for supporting software development. For the tool to be useful it must be capable of working with more than one of these techniques. Also it is not unusual for a single software project to employ more than one of these techniques in the course of software development.

<u>Provide Support for The Information System Planning</u>.

Most information systems must undergo continual change and modification in order to support the changing needs of the user. These changes and modifications are more efficiently controlled and implemented if a system's planning activity takes place. "The purpose of this planning activity is to determine the feasibility and the technical and economic trade-offs for a planned system, based on an assessment of the current environent and an analysis of current use and future requirements" (15:27).

The data dictionary system is an invaluable tool in supporting this planning activity. Information system planning requires an initial assessment of the current environment. This assessment helps the planner to determine the data that is available and to analize information requirements. "These activities... are used in determining the data needed to produce an information product, the data that is already available, the potential conflicts and redundancies, the impact on existing systems, and the potential users of the system" (15:28). The data dictionary supports this activity by recording and coordinating information needs from various members of the organization.

In performing the system's planning activity, it is important for the planner to analyze current usage and determine future requirements. To accomplish this task the system planner must understand how information is actually used to perform specific functions, how data entities are related to both each other and other system components, and the dependencies of the data on other entities and processes.

Many of the tasks associated with system planning activities are time consuming, tedious, and prone to error when performed manuually. "With the aid of an automated tool, such as the DD/DS, the tasks may be simplified, reliability may be increased, and consistency may be maintained, while facilitating coordination of these planning activities" (15:28).

The data dictionary system is an effective tool for information system planning. "It provides coordinated and consistent functional support for documenting the plan and its subsequent use as a control mechanism over development and operation since the DD/DS contains data about the enterprise's operational data, that is, it is an inventory of currently available data. Further, the DD/DS contains information about how the data is used, its relationship to other data entities, occurrences, dependencies, and constraints. Thus with a DD/DS fully operational, its use during the planning phase can increase control over developmental and operational aspects of the organization" (15:29).

Data Dictionary System Should Contain a Security Feature for The Protection of The Informations It Possesses.

The security of information is a concern when designing the data dictionary generation tool. Two reasons for this concern are:

"1. The dictionary database represents a complete inventory of the enterprise's processing system. An intelligent decision on anyone's part intent on accessing the enterprise database in an authorized manner would be to first peruse the contents of the dictionary database.

2. The basic concept of a data dicitonary system includes the idea of a central repository of data, the

dictionary database, which is considered to have a high degree of reliability and the confidence of the database users. As such, precautions should be taken to assure that unauthorized alterations, either accidental or intentional, will be prevented" (11:2-20).

A security facility for a data dictionary system should not simply restrict access to data dictionary information, but should differentiate between creating, reading, altering, and destroying existing data dictionary entities. "In the final count it must be considered that the security of the data dictionary system is related to the security of the entire computer system. The level of security existing there is influenced by the security of the installation, as well as the procedures used by the personnel of the enterprise" (11:2-21).

The Data Dictionary System Should Provide Both Reporting and Interrogation Facilities For Use By the User and Dictionary Administrator.

The benefits derived from the use of a data dictionary system are directly related to the quantity and the quality of the data the dictionary database contains about the information system. The usefulness of the system is also related to the reports generated and the flexibility with which the database can be interrogated in response to specific questions.

"The major categories of reports on the contents of the dictionary database are:

1. Listings of all dictionary entities of a given type, i.e., items, groups, etc. Such listings will in general contain some attributes for each entity.

2. Listings of all attributes for a specified entity of any type. Such listings may be limited essentially to the attributes placed in the dictionary database, or they may also contain data about all or selected dictionary entities which have a logical relationship to the specified item.

3. Usage reports show either the manner in which a given entity is used by other entities, or to which other entities use a given entity. The manner in which such reports are provided may vary in that only one level of usage may be included, i.e., from file to group or vice versa, or that all levels of the hierarchy may be included in the report. In the former case this facility must be invoked multiple times to obtain the same results.

4. A key-word-in-context or key-word-out-of-context facility that can be used to search specifies attributes for given key word. A facility of this nature can be useful in view of the fact that the previously described reports allow only the use of a

limited number of attributes as search arguments. A simple example of the use of the use of such a facility might be to query the dictionary database for a listing of all programs written in COBOL.

5. Some systems provide specialized programmer interfaces at which the dictionary database is made available in a prescribed format. Under such circumstances, it is then possible for an installation to provide extensions to the reporting facilities available" (11:2-14).

Functional Model For The Data Dictionary Generation Tool

With a list of concerns and objectives developed, sufficient background has been established to develop the defintion of the functional requirements for the Data Dictionary Generation Tool. This task is accomplished by formulating a functional model which defines and describes the tool's functional requirements.

A variety of techniques and methods are available for defining system requirements. From among these, the Data Flow Diagram technique was selected to define the requirements's for the Data Dictionary Generation Tool.

Data Flow Diagrams are especially useful in defining the requirements for software systems which contain a complex and varied array of data flows. The Data Dictionary

Generation Tool is this type of system. Because of the intricate data flows associated with the tool, the Data Flow Diagram is an excellent technique for defining the requirements for the Data Dictionary Generation tool.

The following figures and sections display and explain the Data Flow Diagrams associated with the higher levels of the functional model. The diagrams for the entire Data Dictionary Generation Tool functional model are presented in appendix A.

<u>Data Dictionary Generation Tool Functional Model: Top Level.</u>

The top level of the Data Dictionary Generation Tool functional model is displayed in figure 6.

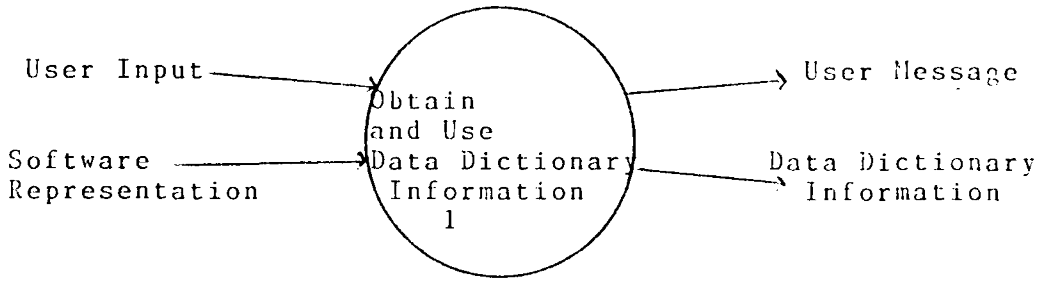

User Input
Software Representation
Obtain and Use Data Dictionary Information 1
User Message
Data Dictionary Information

Figure 6. Top Level Data Dictionary Generation Tool.

This top level diagram represents a vague and abstract idea for the software system. The Obtain and Use Data Dictionary Information operation is the process of extracting data dictionary information from the tool users and various software representation such as SADT, Sturcture Chart, etc and using this information to support the development of software in each phase of the life cycle.

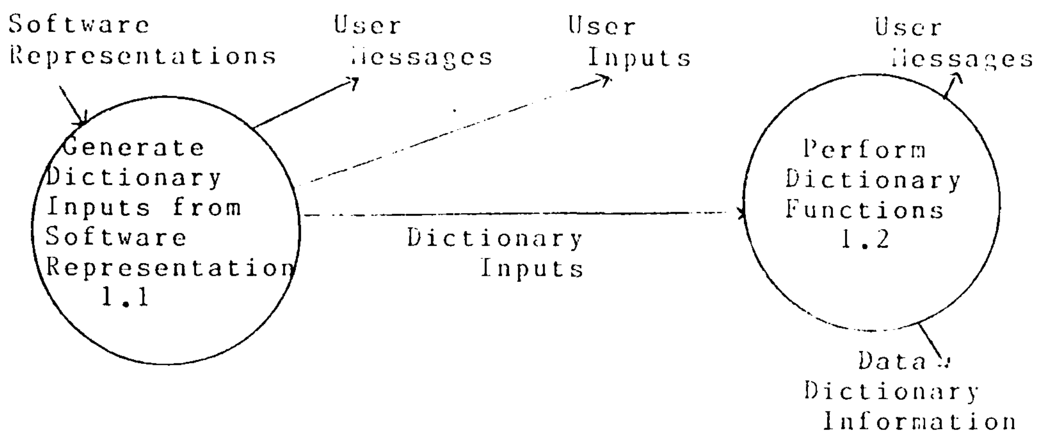

Software Representations
User Messages
User Inputs
User Messages
Generate Dictionary Inputs from Software Representation 1.1
Dictionary Inputs
Perform Dictionary Functions 1.2
Data Dictionary Information

Figure 7. Obtain and Use Data Dictionary Information

<u>Obtain an Use Data Dictionary Information</u>

Figure 7 displays the initial decomposition of the top level of the functional model. This decomposition shows the two primary functions or components of the Data Dictionary Generation Tool. The Generate Dictionary Inputs From Software Representations operation represents that portion of the tool which interprets automatically the information content and of various software representations and converts this information into data dictionary inputs. This operation requires user input to obtain data dictionary information which might not be portrayed on a software representation. The Perform Dictionary Functions operation consists of those tasks which maintain the data dictionary information. This operation also supports the tool user by providing a means by which information in the data dictionary can be retrieved, deleted, added, and modified.

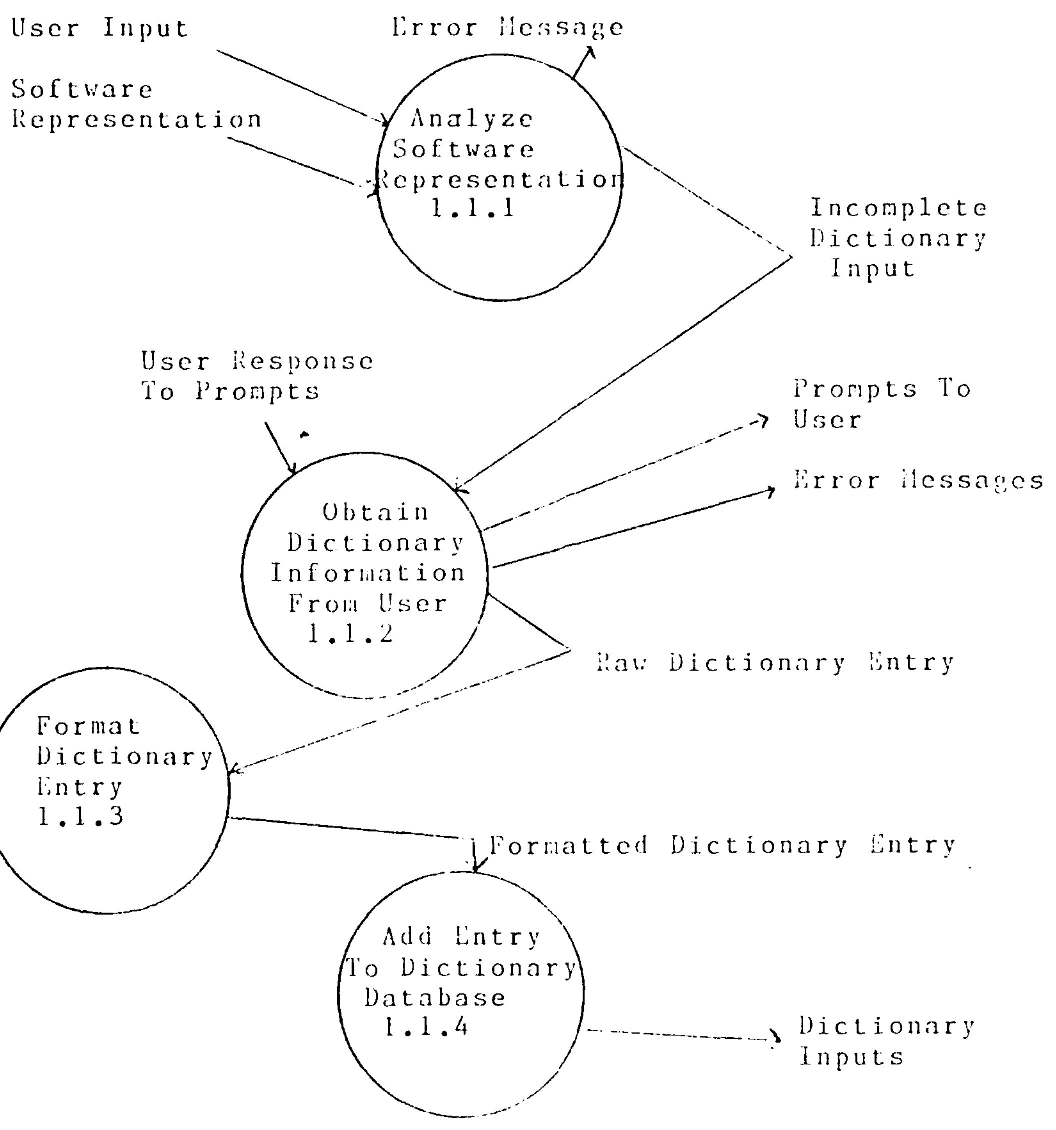

User Input
Software Representation
Error Message
Analyze Software Representation 1.1.1
Incomplete Dictionary Input
User Response To Prompts
Prompts To User
Error Messages
Obtain Dictionary Information From User 1.1.2
Raw Dictionary Entry
Format Dictionary Entry 1.1.3
Formatted Dictionary Entry
Add Entry To Dictionary Database 1.1.4
Dictionary Inputs

Figure 8. 1.1 Generate Dictionary Inputs From Software Representations.

<u>Generate Dictionary Input From Software Representations</u>

Figure 8 displays the decomposition of the Generate Dictionary Inputs From Software Representations operation into its component functions. Initially, the subject software representation is analyzed and the data dictionary information available is obtained. Information not depicted in the software representation is obtained from the tool user by means of the tool displaying prompts to the user and the user responding to the prompts. The information obtained both automatically from the software representation and interactively from the tool user form constitute the unformatted dictionary entry. This information is then formatted and added to the dictionary database, which maintains all data dictionary information.

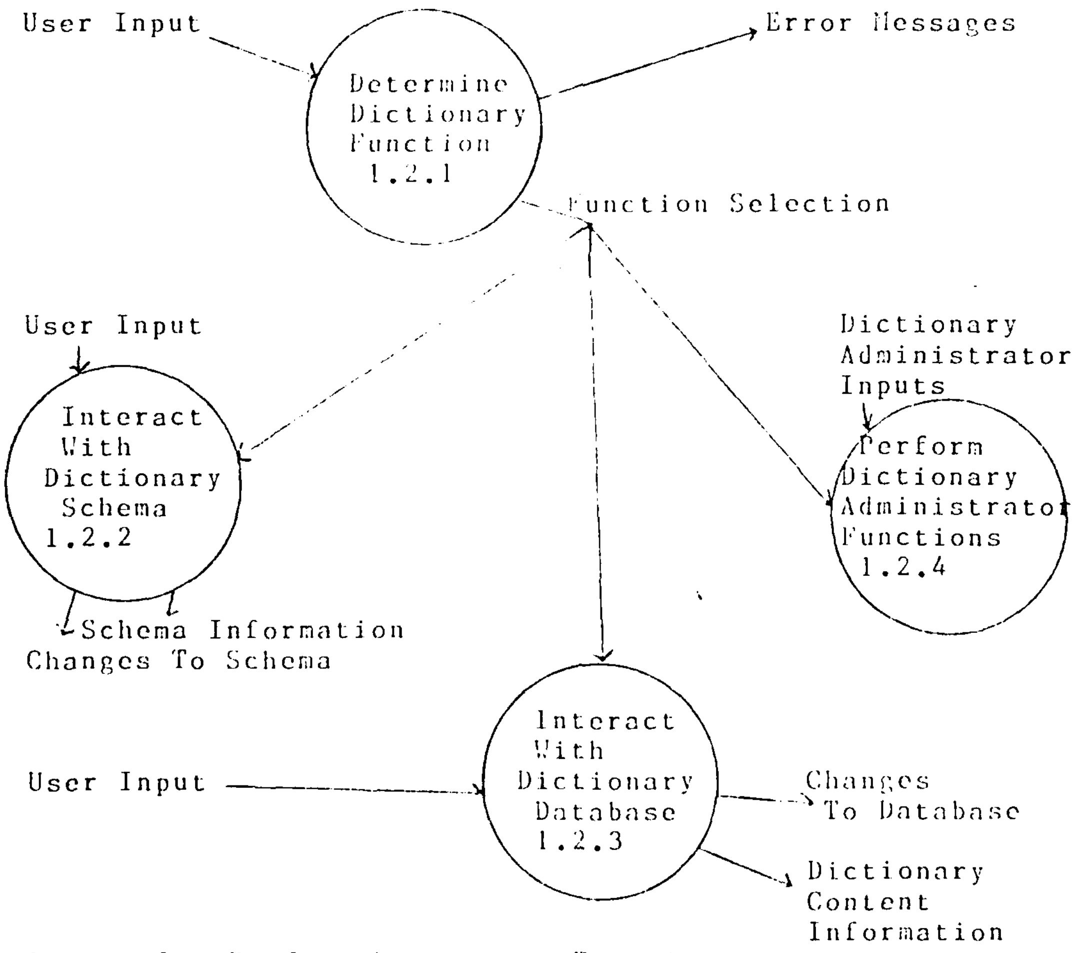

User Input
Error Messages
Determine Dictionary Function 1.2.1
Function Selection
User Input
Interact With Dictionary Schema 1.2.2
Dictionary Administrator Inputs
Perform Dictionary Administrator Functions 1.2.4
Schema Information
Changes To Schema
User Input
Interact With Dictionary Database 1.2.3
Changes To Database
Dictionary Content Information

Figure 9. Perform Dictionary Functions

Perform Dictionary Functions.

Figure 9 displays the decomposition of the Perform Dictionary Functions operation into its component functions. The functions portrayed in figure 9 represent the functions the tool user can request from the data dictionary. Operation 1.2.1, Determine Dictionary Functions, determines and selects the function the tool user has indicated he or she would like to perform. Errors in user input are also checked by this operation and error messages displayed Operation 1.2.2, Interact With Dictionary Schema, allows the user to obain information about the structure of the dictionary and to, if desired, modify that structure. Operation 1.2.3, Interact With Dictionary Database, allows the user access to the data dictionary information maintained in the dictionary database. Operation 1.2.4, Perform Dictionary Administrative Functions, performs tasks essential to the maintenance of the tool such as security and storage scheme selection.

Evaluation Criteria

In order to measure the success of the Data Dictionary Generation Tool in meeting its requirements, a set of evaluation criteria must be established. There are several parmeters which can be used to gauge to success of the Data Dictionary Generation Tool.

The first is the average time spent in learning to use the tool. This parameter will vary from individual to individual. However, the amount of time required for the average user should be minimal, probably in the range from two to four hours.

Another evaluation parameter, closely related to the amount of time required to learn the use of the tool, is the degree of user friendliness the system provides its users. The Data Dictionary Generation Tool is a highly interactive tool whose successful operation is heavily dependent upon the inputs supplied by the tool's users. The user friendliness demonstrated by the tool should be high. However the degree of user friendliness is a subjective and extremely difficult parameter to measure.

System responsiveness is another evaluation parameter which should be considered. As stated above, the Data Dictionary Generation Tool is a highly interactive tool requiring substantial communication between the tool and the user. If the tool's response time to user inputs is slow, it will cause user frustration and dissatifaction. A slow response time will decrease the advantages of the tool in comparision to manual generation of data dictionary information.

The most important parameter by which the Data Dictionary Generation Tool should be measured is how accurately it maintains data dictionary information. If the tool allows errors or contributes to errors in data dictionary information it usefulness is questionable.

III. Preliminary Design

Introduction

Preliminary Design refers to the software development stage during which the functional framework of the software system is determined. The purpose of Preliminary Design then is to establish the functional framework or structure which will reflect the system objectives or requirements specified during the Requirement's Definition Phase of software development. Within this framework or structure the algorithms for the software system are integrated. Without this structure, the associated algorithms would probably not be able to support the objectives of the software system. Therefore, the main purpose of Preliminary Design is to provide a sound framework for the software system.

In developing the Preliminary Design for a software system, the software engineer will seek to establish a hierarchial framework of managerial and functional modules. The framework begins with a single executive module at the top of the structure which can call or use other modules within the software system. These modules may also call or use other modules. Some modules perform the task of managing lower level modules while other modules perform the actual functions required to support the objectives of the software system. The modules constituting this hierarchial framework are linked together by their abililty to call or use one another. These modules may pass data, control and status information back and forth between each other.

The Preliminary Design for the Data Dictionary Generation Tool will be involved with establishing the functional structure for the software system. This chapter will also discuss a design strategy for the data dictionary generation tool. This tool is envisioned as a dynamic tool which will be able to evolve to accomodate new software development methods and their accompanying software representations. This design objective will be used in the initial design of the tool. The dictionary database design will be discussed at length. The database which maintains the data dictionary information is an essential element of the Data Dictionary Generation Tool. For that reason, the design of the dictionary database is an important issue in the overall preliminary design of the tool. The development of the structure of the software system will be discussed and structure charts will be used to provide a graphical representation of the hierarchial framework of the system. The Preliminary Design chapter will conclude with a discussion of how the preliminary design of the system satisfies the objectives and concerns expressed in the Requirements Definition Chapter.

Design Strategy

The Data Dictionary Generation Tool is envisioned as a dynamic tool capable of supporting the entire software development life cycle. To accomplish this goal, the tool must be capable of expanding to accomodate the wide variety of software representations available to the software designers. The design strategy for this tool must be capable of not only supporting existing software representations such as SADTs, data flow diagrams, structure charts, etc but must also possesss the flexibility to accomodate software representations which may be developed in the future. Of course, it is impossible to guarentee that any design strategy will be able to accomodate a unknown software engineering development. However, by providing a well defined and logical design strategy, the flexibility of the tool in supporting new software representation is greatly enhanced.

The initial step in the design strategy is to gain a thorough understanding of the software representation in question. A software representation is important in the software development process because it provides information about the software system under development. The type and quantity of information provided by a particular software representation technique will depend upon the nature of the representation. For example, data flow diagrams provide information about the activities which form a software system and the data inputs and outputs of these activities. Data flow diagrams also depict the flow of data between the various activities which constitute the software system. SADTs, on the other hand, not only depict the flow of data into, out of, and btween activities, but also allows for a data flow to be classified as a control data input for an activity. SADTs also provide the necessary conventions for designating a mechanisms or the means by which an activity performs a function. As the above example indicates, a thorough understanding of the nature of the software representation is essential.

The next step in the design strategy is to determine the information content of the data dictionary for a given software representation. A data dictionary is a repository of data about data. The software representation contains information about the software system it describes. The contents of the data dictionary for a software representation will to a large extent be driven by the nature of the representation. For example, a data dictionary entry supporting a SADT representation of a software system would contain information about control data and mechanisms. This information would not be included in a data dictionary entry supporting a data flow diagram or structure chart representation. When determining data dictionary content, it is important not to let the information content be limited to just the information contained in the software representation. If the tool user possesses additional information of value, it should also be included. For example, it is not possible to determine the data type (ie, character, integer,) from a data flow diagram, SADT, or struucture chart. However, if the system has reached a level of development where the user has knowledge of the data type of a particular data element it should be included in the data dictionary.

Once the information requirements for the data dictionary have been determined, the initial design of the dictionary database can be accomplished. In designing the dictionary database, an important point should be kept in mind. Although the differences in information content of the various software representations does exist, there is also a great deal of commonality between the software representations. For example, SADTs, data flow diagrams, structure charts, and code all depict the flow of data elements into, out of, and between activities. Where possible this commonality should be exploited in designing the data dictionary database. However, valuable information which may exist in only one particular type of representation should not be sacrificed for the sake of maintaining commonality. This point will be further clarified when the database design for the data dictionary database is discussed later in this chapter. The objectives of the initial database design should be to structure the database in a manner which reduces redundancy but maintains traceability and consistency.

With the initial database design accomplished, the next step in the design strategy is to designate the user's view of the database information. The user's view, when used in this context, indicates the manner in which the user interacts with the data dictionary system. In this step, the manner in which the user is presented with database information and the manner in which the user can manipulate database information are defined. The user will as a minimun want to be able to retrieve, insert, delete, and modify the data dictionary contents. The definition of the user's view will also determine the format in which a user will obtain data dictionary information. For example, if the user desires to know the input data for a particular activity, the view would define the format in which that piece of data dictionary information would be presented. The presentation could consist of the exact data element names of all inputs to the desired activity or it could be in the form of an activity definition which included the the names of all input data along with other information about the activity. The most important point to remember in defining the user's view of the dictionary database is to attempt to present the information in a manner which most effectively supports the user's needs.

The user's view represents the manner in which the user desires to manipulate or use the dictionary database. The database design represents the manner in which the information is conceptually maintained. In order to allow the user to perform data dictionary functions, application software is required to connect the user view and the dictionary database. The application software represents the dictionary portion of the data dictionary generation tool. The actual implementation of the dictionary portion of the tool will be heavily dependent upon particular method used to maintain the database and the level of user friendliness the system must support. For example, the database could be designed along either the network, relational, or hierarchial approach. The user might be required to have technical knowledge of the database management system used to maintain the database in order to manipulate the dictionary information or a user friendly menu driven interface which required no technical knowledge could be provided.

The development of the application software to connect the desired user's view with the dictionary database should follow the software development life cycle approach. The specification of the user's view and the initial database design will provide an excellent foundation for formulating the requirements definitions for the dictionary software. With the development of the dictionary software, the dictionary portion of the tool is complete. The information contained in a software representation along with other necessary data dictionary information is maintained in the dictionary database. A dictionary user can perform the necessary interactions with the database by specifying his/her desires through the user view. The application software will connect the user's view with the database and enable the user to perform the desired operation. The automatic information extraction portion of the data dictionary generation tool can now be addressed.

The initial step in developing this portion of the tool is to determine for a specific software representation which portions of a data dictionary entry fo. that representation can be determined directly from the representation and what information must be provided by the user. With this determination made, the basic requirements for this portion of the software system have been identified. In order to obtain data dictionary information from the software representation, the software system must access and interpret the contents of the representation. The system software must then extract the data dictionary information and convert it into a form suitable for insertion into the data dictionary database. The software system must communicate interactively with the user in order to obtain data dictionary information which can not be derived from the software representation.

The design strategy presented here is intended to provide an approach to follow in expanding the tool to accommodate new software representations and their associated data dictionary information. The following sections in this chapter will be concerned with developing the database and applications software for the initial set of software representation to be supported.

Data Dictionary Information Content

The Preliminary Design of the Data Dictionary Generation Tool will attempt to support four different types of software representations and their associated data dictionary information. The software representations to be supported are SADTs, structure charts, data flow diagrams and code. SADTs,data flow diagrams, and structure charts have been described earlier in this paper. The code software representation is the actual source code which makes up the software system in question. This representation is formulated during the implementation phase of the software life cycle. These four software representation were selected because of their widespread use in the Defense Community.

When discussing the dictionary content for each of these software representations, it is useful to consider the information for the data dictionary as being in one of two categories. The first category is actions, which contains all information elements about the various functional or managerial modules which make up the system. The second information category, data, represents the information which the action modules use or manipulate in performing their various functions. Both categories of information contain elements which relate data and actions to each other. For example, action information would identify the data inputs and outputs of a functional module. Data information, on the other hand, would identify the action modules which used or manipulated a particular data item.

Because of this categorization of information within each software representation, the data dictionary information desired for each representation will be discussed from both a action entry and data entry point of view. The discussion of the informaton content of both the action and data portions of the software representation will show that a large degree of commonality exists between the various representations. In addition, a large degree of commonality will also be seen btween the action and data information content. The complete listing of each representation's action and data information elements is presented in figures 10 and 11.

In discussing the various information elements that make up the data dictionary contents for the various software representations, those elements which are common across the range of the three representations and those which are also common among the data and action information categories for all representations will be discussed first. The following paragraphs will describe each of these information elements and explain its meaning or value as data dictionary information. It is important to remember that these information elements are present in both the data and action information categories in all three software representations under consideration.

Project.

The project information element identifies a group of software developers who are responsible for entry of information into the dictionary database. The project identifier is important because it allows more than one group to be working on the same software project at the same time. The designation allows different groups to use the same dictionary database without having to be concerned about interferring with the work of another group. This capability is especially important when the the data dictionary is being used to support a large software development effort,

Name

Name is the title given to the acitvity or data element represented in the dictionary database. The name element associated with an action or data element should be unique in that no other data element or action should have the same name. The name information element should, to the extent possible, describe the data element or activity it represents.

SADT	Data Flow Diagram	Sructure Chart	Code
Project	Project	Project	Project
Number	Number	Number	Number
Name	Name	Name	Name
Inputs	Inputs	Inputs	Inputs
Outputs	Outputs	Outputs	Outputs
Conrols	—	—	
Mechanisms			
Description	Description	Description	Description
Reference	Reference	Reference	Reference
Alias	Alias	Alias	Alias
		Input Flags	
		Output Flags	
		Global Data	Global Data
		Used	Read
		Global Data	Global Data
		Changed	Write
		Algorithm	Algorithm
Parent Node	Parent Node	Parent Node	
Child Nodes	Child Nodes	Child Nodes	
		Called By	Called By
		Calls	Calls
		Hardware	Hardware
		Read	Read
		Hardware	Hardware
		Written	Written
			Program
			Language
Date	Date	Date	Date
Originated	Originated	Originated	Originated
Original	Original	Original	Original
Author	Author	Author	Author
Modify	Modify	Modify	Modify
Date	Date	Date	Date
Modify	Modify	Modify	Modify
Versions	Versions	Versions	Versions

Figure 10. Software Representations Action Entity Information Elements.

Description.

The description information element is a text input which describes an activity or data element contained in the data dictionary. The description is the developers attempt to define the function or purpose of an activity or the nature of a data element.

SADT	Data Flow Diagram	Structure Chart	Code
Project Name	Project Name	Project Name	Project Name
Description	Description	Description	Description
Sources	Sources	Passed From	Passed From
Destination	Dstination	Passed To	Passed To
Composition	Composition	Composition	Composition
Part Of	Part Of	Part Of	Part Of
Data Type	Data Type	Data Type	Data Type
Min Value	Min Value	Min Value	Min Value
Max Value	Max Value	Max Value	Max Value
Valule Set	Value Set	Value Set	Value Set
Alias	Alias	Alias	Alias
		Storage Type	Storage Type
Reference	Reference	Reference	Reference
Original Date	Original Date	Original Date	Original Date
Original Author	Original Author	Original Author	Original Author
Modify Date	Modify Date	Modify Date	Modify Date
Modify Author	Modify Author	Modify Author	Modify Author

Figure 11. Software Representation Data Information Elements

Aliases.

An alias, when used in a data dictionary context, is another name for an existing activity or data element. The use of an alias can cause confusion in both the data dictionary and software system development. Their use should be avoided whenever possible.

The remaining five information elements which are common to both the data and action information elements of the three software representations under consideration are concerned with maintaining a historical record of the data element or activity they describe. The date originated and original author information elements identify the time and person or group that initially entered a data element or activity into the dictionary database. In a similar fashion, the modification date and modification author identify the time and person who made a change in the associated data or action entry. The version information element identifies each modification made by indicating its sequence. For example, the initial entry of an activity or data element would be identified as version 1 while the first modification to the initial entry would be designated as version 2.

The next group of data dictionary information elements discussed will be those data information elements which are common among the three software representations. The first four of these elements seek to describe the actual value which will be associated with the data element.

Data Type.

The data type information element describes the basic characteristics of the values associated with a data element. For example, if the data element value was either true or false then the associated data type would be boolean. In a similar fashion, if the value was always a number the data type could be, depending on the nature of the number, either integer or real.

The Min Value and Max Value.

The min value and max value information elements describe the highest and lowest values the data element can represent. If the possible values for the data element order were whole numbers between 1 and 10, the min value for data element order would be 1 and the max value would be 10.

Value Set.

The value set information element is used when a data element can only assume a limited number of values. For example, if due to the nature of the software system the data element could only assume three values: high, low, or medium, then the value set for the data element would contain each of these three values.

It is important to note that a data element will not always have a value set, min value, or max value information element associated with it. However, these information elements do provide valuable information in certain situations.

Composition and Part Of.

The compositon and part of data information elements provide data dictionary information about the make up of a data element and it relationship to other data elements. For example, the data element employee salary could be considered as part of the data element employee pay. In a similar fashion, employee pay is composed of employee salary as well as other data elements such as employee social security number or employee name.

Sources, Destinations, Passed From, and Passed To.

The source, destination, passed from, and passed to data information elements describe the flow of data into, out of, and between the various functional and managerial modules which make up the software system. Although these data elements are common in all three data portions of the software representations, there is a naming inconsistency which could lead to confusion. The SADT representation calls activities which output data elements sources and activities which accept or input data elements as destinations. Structure charts and code, on the other hand, designate activities outputting data elements as passed from and activities inputting data elements as passed to. The use of different terminology is not important because the meaning the information elements are the same in all three representations.

The next three data information elements discussed: requirements #, SADT data element, and SC parameter, provide a trace capabiltily between the three representations and more importantly a reference between the various stages of software development.

Requirement #.

The requirement # data information element is used to reference a data element used in the SADT representation to a previously defined requirement. This requirement represents a stated objective or goal of the system which was formulated by the system developer and/or system user. The use of this information element identifies a system requirement which the subject data element is intended to help resolve. Linking system requirements to a SADT data element is appropriate because the SADT representation is widely used in the requirements definition phase of the sotware lifecycle.

<u>SADT Data Item</u>.

The SADT data item information element relates data elements used in the structure chart representation to a data element used in the SADT representation. The relationship between these two representations is appropriate. The hierarchial structure in the structure chart representation makes it a valuable tool in the design phase of software development. The link between these two representations allows for a tracing of data elements as they develope from the requirements phase to the design phase.

<u>SC Parameter</u>.

The SC parameter data information element relates the actual data element or variable used in the code representation to the corresponding data elements or parameters used in the structure chart representation. The information element allows for the linking of information in implementation phase, represented by the code representation, to corresponding information in the design phase, represented by the structure chart representation.

The value of the trace data information elements is especially valuable in the error correction and modification of software systems. For example, if an error is detected during the implementation phase the error can be traced back through the design and requirements phases. This will help to ensure that the error is removed completely from the system and enhances the designers abiltiy to track software problems to their source. When a modification to a software system is proposed, it is extremely valuable to be alble to determine the overall effect of the modification on the entire system. By tracing the effected data elements through all phases of development, the designer can better determine the influence, both positive and negative, that a modification will have on the system.

Storage Type.

The final data information element to be discussed is storage type. This information element is common to the structure chart and code software representations. The SADT representation does not contain this information element. Storage type represents a classification of the data element as it is viewed or used by the software system. There are two classifications associated with this information element: passed and global. The global classification indicates that the data element value can be both accessed and changed by any portion of the software system. It is known throughout the system and can be used or changed by any functional module in the system. The passed classification indicates that the data element is only known in a portion of the system and for its value to be either used or change requires that the data elements value and type be passed or sent to other portions of the system.

This completes our discussion of the information elements associated with the data portions of the software representations. The action information elements will now be discussed. As before, the action information elements which are common among the four representations will be discussed first.

Inputs, Outputs, Input Data, and Output Data.

The inputs outputs, input data, and output action information elements identify the data elements which a action or activity uses or produces in performing its function. As the names indicate, the action takes the input data and uses or manipulates it. The results of the activity are the output data which flows or is passed out of an activity. The SADT, DFDs, and code representations designate the data elements associated with an activity as inputs and outputs. Structure charts, on the other hand, use the naming convention input data and output data. SADT representations use another input data designation known as control which is not present in the structure chart and code representations. This information element will be discussed later in this section.

Parent Node, Children Nodes, Called By, and Calls.

The parent node, children nodes, called by, and calls action information elements depict the composition or make up of an action. Parent node and children node are terms used in the SADT representation to depict the logical decomposition of an activity into its component parts or children. For example, the activity Find Average could be considered as a parent node with the children nodes Read Entry, Add To Sum, Divide By Number of Entries. The structure chart and code representations use the terms calls and called by to depict the composition of activities. The term call is normally associated with the use of an activity by another activity. This meaning is slightly different from the parent/children scheme used in SADT. Although this difference does exist, both sets of terms still depict a composition relationship and contain sufficient commonality to be grouped together.

Requirement #, SADT #, and SC #.

The requirement #, SADT #, and SC # action information elements provide a trace capability between actions depicted in the three representations and the requirements, design, and implementation phases of software development. These information elements serve the same purpose for the software system's actions as the requirements #, SADT data item, and SC parameter information elements did for a software system's data elements. This completes our discussion of the action information elements which are common among the three software representations.

The next two action information elements discussed,controls and mechanisms, are only present in the action information for the SADT representation. The control action information element is depicted as an input to an SADT activity. The control information element identifies input data flows which an activity uses to control its execution. For example, a control input could be used to determine the flow of execution inside the activity. The mechanism action information element is also depicted as an input into an activity in the SADT representation. A mechanism represents the means by which an activity performs its functions.

The discussion of mechanisms and controls completes the discussion of information elements, both action and data, for the SADT representation. We will now turn our attention back to the action information elements and discuss those elements which are common in both the structure chart and code representations.

<u>Global Data Used</u>, <u>Global Data Changed</u>, <u>Global Data</u>

Read, and Global Data Written.

As discussed earlier, global data refers to data elements which can be both accessed and changed by any activity or module in the software system. While global data elements are extremely handy in developing software, they do present an opportunity for introducing serious errors into the system due to their easy access. The action information elements global data used and global data changed depict the effect an action or module has on a global data element in the structure chart representation. The action information elements global data read and global data written perform the same function in the code representation.

Algorithm.

The algorithm action information element is a text dscription of the method or manner in which an activity or module performs its function. For example, if the function of a module was to calculate the average employee salary the following formula could be use to describe the algorithm: total salary all employee/number of employees = average employee salary.

Files Read and Files Written.

The files read and files written action information elements represent the obtaining of information and the outputting of results to and from existing files in the system by the action or module. In many cases, an activity will obtain input information from a previously created file in the system. The action information element files read identifies the name of the subject file. In a similar fashion, once an activity has completed processing its input information its outputs or writes the results to a file. The action information element files written identifies the name of this file.

<u>Hardware Read and Hardware Written</u>.

The hardware read and hardware written action information elements indicate the interaction of the software activity with the computer haredware which supports the system. An example of a hardware read or hardware written information could be an input/output port number.

This completes our discussion of the action information elements which are common between the structure chart and code representations. The remaining action information elements are unique to a particular representation.

Input Flags and Output Flags.

The input flag and output flag action information elements indicate the use of a boolean data element to send control information to an activity. For example, module error check could send a boolean data element to another module to indicate that no error exists. Input and output flag information elements are used in the structure chart representation.

<u>Program Language</u>.

The program language information element indicates the actual program language which is used in writing the source code for an activity or module. Examples of the program language information content are: Pascal, Fortran, and Cobol.

This completes our discussion of the information elements which constitute the data dictionary information for the subject software representations. This discussion has briefly described these information elements and pointed out the commonality which exists among the representations.

Database Design

In the previous section, the information content of the data dictionary was discussed in detail. This provided a clearer understanding of the type of information the dictionary database must support. The previous sections also identified numerous areas where the information content of the three subject representations are common. By defining all common areas, the identification of those information elements which differ among the three representations made more meaningful. With this background, the logical structuring of the information in a manner which best supports the dictionary database can begin. The process of logically structuring the information is known as database design. Two of the major goals in database design are to reduce data redundancy or information duplication, where possible, and to strengthen data independence, the lack of data structure dependence on application software.

There are three basic approaches to database design: relational, hierarchial, and network (17:63). "The hierarchial approach sees a hierarchy of objects as the most typicallly useful data structure. Relationships between an object and several subordinate objects, e.g., between a manager and his or her employees or between suppliers and the parts they supply, are hierarchial relationships... (18:97)." The hierarchial approach views the data structure in the database as a series of parent/children relationships which is often depicted as a simple tree structure. The advantages of the hierarchial approach are: the familiarity of many users with the hierarchial structure and the significant degree of data independence supported (19:106). On the other hand, the major disadvantages of the hierarchial approach are: the manner of dealing with many to many relationships is clumsy, the basic database operations such as insertion and deletion are overly complex, deletion of a parent element results in the deletion of all information about its children data elements, and information about a child is accessible only through its parent (19:106-109).

The network approach sees "...hierarchial relationships as a special case of a network relationship between objects. For example, in a manufacturing application each part may have many suppliers and each supplier may supply many parts. Each of these relationships is hierarchial, however, the overall relationship between suppliers and parts is a network relationship. A network system assumes that each object may participate in network relationships" (18:97). The major advantage of the network approach is that it easily implements the many to many relationships which exist in real life. "The main disadvantage of the network model is its complexity. The applications programmer must be familiar with the logical structure of the data base because she/he has to "navigate" through different set occurrences with the help of connector type record occurrences" (19:121).

The relational approach does not "distinguish between objects and relationships. The basic construct is a relation, or group of related data elements. A relation may represent an object, say a part, or a relationship, such as the relationship between parts and suppliers" (18:97). The major advantage of a relational database is its simplicity. The relation can be equated to an information table which greatly enhances user understanding. Other advantages associated with the relational approach are that it provides a relatively higher degree of data independence than the hierarchial and network approaches and that it is based upon a well developed mathematical theory or relations (19:95).

The relational approach was used to design the database for the data dictionary generation tool. The relational approach was selected because of its simplicity and ease of understanding. In the relational approach, information is organized into tables or relations. A table or relation contains information elements which are related or logically belong together. The columns of the table represent the attributes of the relation. The rows or tuples of the relation represent single entries into the relation. For example, a part relation could contain attributes which describe a part such as part number, color, weight, and quantity. A tuple in a part relation would contain values for the various attributes which apply to a specific part. Figure 12 provides a graphical display of these concepts.

Part Relation

Part Number	Part Name	Color	Weight	Quantity
62ABY	bolt	red	26	12
3GC1F	screw	blue	5	9
49HV6	nut	white	83	20

Figure 12. Example Relational Table.

The organization or information elements into tables or relations closely parallels the manner in which humans think about information organization. Because of this, the relational approach is easier to both understand and use than either the hierarchial or network approaches to database design.

During the first part of this chapter, the information elements essential to the data dictionary database were identified and discussed. The database design problem is concerned with organizing these information elements into relations or tables in a manner which supports the functions of the data dictionary, reduces data redundancy, and enhances data independence.

"The process of crystallizing the entities and their relationships in table formats using relational concepts is called the normalization process. Normalization theory is based on the observation that a certain set of relations has better properties in an updating environment than do other sets of relations containing the same data (19:91)." Normalization concepts provide a useful aid in the organization of information elements into tables or relations which can be supported by a relational database.

Before discussing normalization any further, it is important to understand the concept of key attributes in a relation. A key is an attribute or combination of attributes with values that are unique within a relation and can be used to identify the tuples of that relation (17:87). Consider a relation which contains information about parts (Figure 13A) which contains the attributes part name, part number, part color, and weight. If the part number uniquely identifies each tuple in the relation then it, the part number attribute, can serve as a key for the relation. Now consider figure 13B. In this relation, the part number alone is insufficient to uniquely identify each tuple in the relation. In this relation, a combination key consisting of both the part number and part color attributes are required to identify the individual tuples in the relation. "Not every relation will have a single attribute key. However, every relation will have some combination of attributes that, when taken together, have the unique identification property.... The existence of such a combination is guaranteed by the fact that a relation is a set. Since sets do not contain duplicate elements, each tuple of a given relationship is unique with respect to that relation, and hence at least the combination of all attributes has the unique identification property (17:88)."

A. Parts Relation 1

Candidate Key Candidate Key

Part Name	Part Number	Color	Weight
bolt	124	red	6
screw	138	blue	5
nut	159	green	8

B. Parts Relation 2

Primary Key Primary Key

Part Name	Part Number	Color	Weight
bolt	1246	red	6
bolt	1246	blue	6
bolt	5892	blue	4

C. Shipment Relation

Primary Key Foreign Key

Shipment #	Part Number	Quantity
21	124	267
25	159	1200

Figure 13 Use of Keys In Relations.

Figure 13A illustrates another situation which often arises in a relation. The attribute part name also possesses the property of being unique for every tuple in the relation. In this situation, the relation is said to possess two candidate keys, part name and part number. In this particular situation, it would be appropriate to designate one of these attributes as the primary key and the other attribute as an alternate key for the relation.

Figure 13C illustrates another use of keys in a relational database. Relation shipment contains the attributes part number, quantity and shipment #. Notice that the attribute part number in this relation constitutes an index into the parts relation illustrated in figure 13A. An attribute such as part number in the shipment relation is known as a foreign key into the parts relation. Foreign keys are useful in designating relationships between different tables or relations in a relational database.

It is important to realize that tuples in a relation represent entities in the real world. For example, a tuple in the parts relation represents information about a particular part that could be used or produced by an organization. In a similiar fashion, a tuple in the shipment relation provides information about the content and size of a particular parts shipment. The keys which exist in these relations serve as a unique identifier for the entities represented in the tuples of the various relations.

Keys are an important concept in the relational approach to database design. Because of their importance, two important integrity rules are imposed. Integrity Rule 1 is concerned with maintaining the integrity of entities. It simply states that no component of a primary key value may be null in a tuple of a relation (17:88). Because the key serves as a unique identifier for each tuple within a relation, an identifier which was null in value would be a contradiction in terms and can not be allowed.

Integrity Rule 2 is concerned with maintaining referential integrity. It is common for one relation to contain references to another relation. For example, the shipment relation in figure 13C, by means of foriegn key part number, is able to reference the parts relation shown in figure 13A. If the part number value in a particular tuple of the shipment relation did not exist in the part relation, it would be a violation of referential integrity. The subject tuple in the shipment relation would be describing a shipment of parts which, as far as the parts relation was concerned, did not exist. Simply stated, Integrity Rule 2 specifies that if a tuple in a relation references a tuple in a different relation, that tuple must exist (17:90). To state this in another manner, an attribute which represents a foreign key key may only possess a null value or a value which exists in the referenced relation.

With an understanding of key attributes in a relation, the discussion on database design and the normalization process can continue. As stated earlier, normalization is the process of grouping data elements into tables representing entities and their relationships. "The reason one would use the normalization procedure is to ensure that the conceptual model of the data base will work. This means, not that an unnormalized structure will not work, but only that it may cause some problems when applications programmers attempt to modify the data base (19:130)."

Normalization theory is built around the concept of normal forms. A relation is said to be in a particular normal form if it satisfies a certain specified set of constraints.

"Numerous normal forms have been defined... Codd originally defined first, second, and third normal forms (1NF, 2NF, 3NF)...." (17:238). Figure 14 displays the currently existing normal forms.

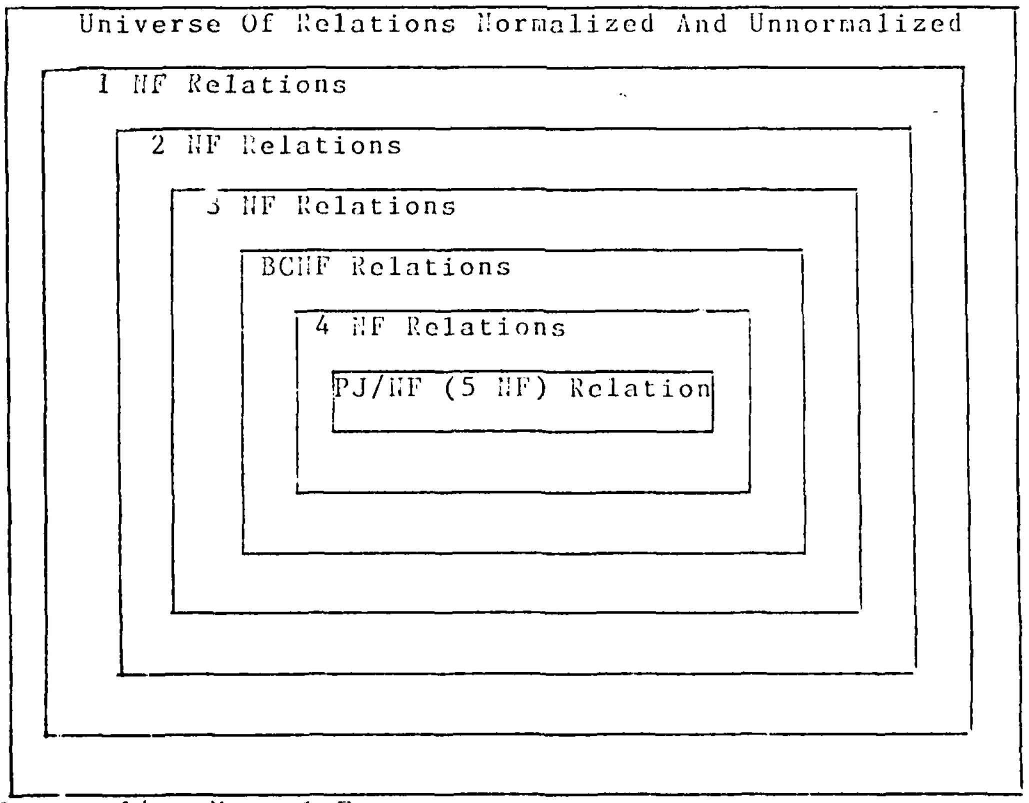


Figure 14. Normal Forms

As figure 14 suggests, "all normalized relations are in 1NF; some 1NF relations are also in 2NF; and some 2NF relations are also in 3NF. The motivation behind the definitions was that 2NF was "more desirable" than 1NF,...,and 3NF was more desirable than 2NF. That is, the designer should generally choose 3NF relations in designing a database, rather than 2NF or 1NF relations (17:238-239)."

For the purposes of this investigation, relations were only formally normalized to the third normal form. However, other normal forms do exist and are displayed in figure 14 and briefly summarized below.

"Codd's original definition of 3NF suffered from certain inadequacies.... A revised (stronger) definition due to Boyce and Codd, was given... -stronger in the sense that any relation that was in 3NF by the new definition was certainly 3NF by the old, but a relation could be 3NF by the old definition and not by the new. The new 3NF is sometimes called Boyce/Codd Normal Form (BCNF) to distinguish it from the old form. Subsequently, Fagin defined a new "fourth" normal form (4NF) and more recently another form which he called "projection-join normal form" (PJ/NF, also know as 5NF) (17:239)."

As stated earlier, the data dictionary database design considered only the first three normal forms to be important. Before discussing the meaning and constraints associated with these normal forms, it is important realize that normalization theory does not constitute a hard and fast process for database design, but rather a set of guidelines which aid in the design process. "Normalization theory is a useful aid in the design process, but it is not a panacea. Anyone designing a relational database is advised to be familiar with the basic techniques of normalization..., but we certainly do not suggest that the design shouldbe based on normalization principles alone (17:238)."

A relation is in first normal form (1NF) if and only if all underlying domains contain atomic values only (17:243). To state this in another manner, every value in the relation, each attribute value in each tuple, is nondecomposable so far as the system is concerned. A relation is considered to be in first normal form when there exists at every row and column position in the table only one value, never a set of values.

A relation is in second normal form if and only if it is in first normal form and every nonkey attribute is fully dependent on the primary key (17:246). This means that a relation is in second normal form when the value of the primary key attributes destermine the value of the other attributes in the relation. For example, the part relation shown in figure 13A demonstrates this idea. The primary key , part number determines the value of the other attributes in a particular tuple of the relation.

A relation is in third normal form if and only if it is in second normal form and every nonkey attribute is nontransitively dependent on the primary key (17:248). When one nonkey attribute can be determined with one or more nonkey attributes, there is said to be transitive functional dependency between the two (17:247). As an example of a relation which possesses transitive dependence, consider a relation named supplier. This relation contains three attribute fields: supplier number, city, and status. In this relation, the primary key is supplier number and city and status are nonkey attributes. As a condition of this relation, assume that status is determined by the city in which the supplier is located. Based upon this condition, the status attribute value can be determined by the primary key value or the nonkey attribute value for city. Although the city attribute value is determined by the supplier number, the fact that the status value can be determined from the city value leads to a situation where transitive dependency exists.

A method of removing this transitive dependence is to decompose the supplier relation into two new relations, supplier city and city status. Figure 15 displays both the original supplier relation with transitive dependency and the two newly formed relations in third normal form.

Supplier Relation

Primary Key

Supplier Number	City	Status

2NF With Transitive Dependence

City Status Relation

City	Status

Supplier City Relation

Supplier Number	City

New Relations In Third Normal Form

Figure 15 Transitive Dependence

The design of the data dictionary database utilized the normalization process in formatting all relations to at least the third normal form. As pointed out earlier, the use of the normalization process alone will not ensure a good database design. Numerous factors and trade offs come into play during the design process. An important point to keep in mind is the intended purpose of the database under design. A thorough understanding of how database information will be used and changed in the course of normal operations is essential. Also of primary concern is the effect the database design will have upon applictions software written to interact with the database. In the following section, the data dictionary database will be presented and discussed.

Data Dictionary Database

In this section, the relations which makeup the dictionary database are discussed. The contents of the relations and how they solve the problem of meeting the information maintenance requirements for the various software representations are also be discussed. The alternatives considered when designing the database are presented and the rationale for making certain design decision are discussed.

When the information content of the dictionary was studied, it was recognized that a great deal of commonality existed among the various software representations which are supported by the data dictionary generation tool. In discussing the relations which make up the dictionary database, initial discussion focus on those relations which the software representation have in common. Discussion will then be directed to those relations which are unique to specific software representations.

Description Relation.

The description relation contains the textual description of the action and data entities for all software representations supported by the dictionary. In reality, there are eight different description relations contained in the database. A description relation exists for both the data and action entities for all four software representations. Figure 16 provides a graphical display of the attribute fields contained in this relation and a list of the eight relations which use this format and there associated software representation. Also included in figure 16 is the entity type of the item described by this relation which would correspond to the classification of the value contained in the name field of this relation

Description Relation

Project	Name	Line	Description
*	*	*	

Database Relation	Software Representation	Entity Type
a_description	SADT	activity
d_description	SADT	data item
b_description	Data Flow Diagram	bubble
df_description	Data Flow Diagram	data flow
pr_description	Structure Chart	process
p_description	Structure Chart	parameter
m_description	Code	module
v_description	Code	Variable

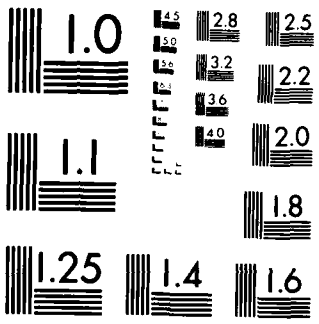
* primary key value

Figure 16. Description Relation.

The primary key for this relation is a combination of the project name, name, and line attributes. The project attribute identifies the team or individual responsible for this particular entry into the dictionary database. The name attribute identifies the particular action or data entity being described. The line attribute identifies the particular line of text which an individual tuple in the

AD-A152 215 AN AUTOMATED/INTERACTIVE SOFTWARE ENGINEERING TOOL TO GENERATE DATA DICTIONARIES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. C W THOMAS 2/4

UNCLASSIFIED DEC 82 AFIT/GCS/ENG/84D-29 F/G 9/2 NL



relation contains. The description attribute contains for each tuple in the relation 60 characters of text which describes the entity identified in the name attribute. Figure 17 diplays what the relation when it contains some actual values.

Project	Name	Line	Description
team 1	qty	1	numeric value which represents
team 1	qty	2	the number of items required by
team 1	qty	3	the customer to complete a sale.

Figure 17 Description Relation Example.

<u>History Relation</u>.

The history relation provides information concerning the modification or change a dictionary entity undergoes within a particular development phase. This relation maintains information about when and by whom a dictionary entity is modified. Figure 18 presents a graphical display of this relation and its attribute fields. It also lists the eight relations in the database where this format is used and the associated software representation. Also included is the entity type of the item described by this relation.

History Relation

Project	Name	Version	Date	Author
*	*	*		

Database Relation	Software Representation	Entity Type
a_history	SADT	acitivity
d_history	SADT	data item
b_history	Data Flow Diagram	bubble
df_history	Data Flow Diagram	data flow
pr_history	Structure Chart	process
p_history	Structure Chart	parameter
m_history	Code	module
v_history	Code	variable

* primary key

Figure 18. History Relation.

The primary key in this relation is a combination of the project, name, and version attribute fields. The project attribute identifies the group or individual responsible for the dictionary entry. The name attribute identifies the dictionary entry being described. For example, in relation p_history the name attribute would be the name of a particular parameter in the structure chart software representation. The version attribute sequentially identifies the modifications to a particular dictionary entity. For example, when an entity is initially entered into the data dictionary its version is identified as 1.0. When this entity is modified, the version attribute become

1.1 in a new tuple in this relation. The date attribute designates the month, day, and year when the entity was modified. The author attribute identifies the individual responsible for changing some aspect of the entities meaning in the dictionary. Figure 19 provides a demonstration of what this relation might look like when actually used to document entity modifications.

Project	Name	Version	Date	Author
team1	data	1.0	8-9-84	Ted
team1	data	1.1	8-24-84	Bill
team1	data	1.2	12-14-84	Mike

Figure 19 History Relation Example.

The dictionary database only maintains the current information on a dictionary entity. In other words, when any information content on an entity is modified the old information content is not maintained for reference purposes. The history relation, however, does provide a means for maintaining a record of all entity modification which take place as well as the time when they occured and the individual responsible for the change. By maintaining this relation, it is possible to recover this information from the author of the change or old printed cpoies of the dictionary contents.

Hierarchy Relations.

The hierarchy relations contain information about the logical decomposition of action and data entities into other action and data entities. The concept of logical decompostion of the action and data elements associated with a software project is very important in the Top Down Design method of software design.

The Top Down Design Method initially considers a software project to consist of only one action entity and its associated data entities. This single action and its associated data entities are then decomposed into a series of more detailed entities. These newly derived action and data entities are then, themselves decomposed into still more detailed components. This decomposition process continues until a level of detail is reached where further decomposition is not possible.

This process allows the software designer to begin with a highly abstract concept of the software project, represented by the initial action entity and its associated data entities, and logically decompose the project into smaller more detailed components represented by the derived action and data entities.

The concept of Top Down Design is supported by the SADT, data flow diagram, and structure charts methods of software representation. Figure 20 demonstrates the Top

Down Design Method as it might be portrayed in the data flow diagram representation

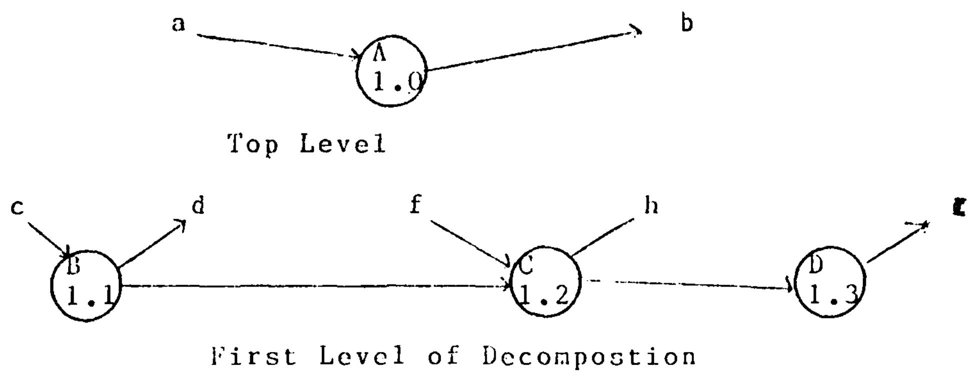


Figure 20. Logical Decomposition Using Data Flow Diagrams.

In order to support the logical decomposition process, the hierarchy relation, by means of its attributes, ties each component entity to the entity from which it was derived. Figure 21 presents a graphical display of the attribute fields contained in this relation. It also lists the relations in the database which use this format. The corresponding software representation supported and the entity type described by these relations are also listed.

Hierarchy Relation

Project	High_Name	Low_Name
*	*	*

Database Relations	Software Representations	Entity Type
a_hierarchy	SADT	activity
d_hierarchy	SADT	data_item
b_hierarchy	Data Flow Diagram	bubble
df_hierarchy	Data Flow Diagram	data flow
pr_hierarchy	Structure Chart	process
p_hierarchy	Structure Chart	parameter
v_hierarchy	Code	variable

* primary key

Figure 21. Hierarchy Relation.

The primary key for this relation consists of a combination of all attributes. The project attribute identifies the individual or group responsible for this dictionary entry. The high_name attribute identifies the name of the entity which is the parent of the lower level entities. The low_name attribute identifies the entities which are children or were derived from the parent entity identified in the high_name attribute.

The code software representation uses this relation in a slightly different manner than the other three representations. The concept of logical decomposition does not come into play in the v_hierarchy relation which supports the code representation. In this case, the relation supports the idea of a variable being derived from a data structure supported by a particular programming language. A good example of this situation is the record structure in the Pascal programming language. A record is a data structure which can consist of many different fields. A variable derived from a Pascal record would be considered the child of that record in the hierarchy relation.

Figure 22 presents an example of how the hierarchy relation would maintain information about the logical decompostion process in the structure chart representation. The example displayed in figure 20 is documented in this figure.

df_hierarchy (data entities)

Project	High_Name	Low_Name
team 1	a	c
team 1	a	f
team 1	b	i
team 1	b	d
team 1	b	h

b_hierarchy (action entities)

Project	High_Name	Low_Name
team 1	A	B
team 1	A	C
team 1	A	D

Figure 22. Hierarchy Relation Example.

Reference Relation.

The reference relation contains information which allows the development of an action or data entity to be traced through the software design process. Different software representations will be used to develope software in the various stages of the software life cycle. The reference relation contains information which identifies the particular software representation used in the previous development stage and the reference or references which identify the entity in the previous development stage. Figure 23 provides a graphical display of the attributes which make up this relation. Figure _23 also lists the relations using this format, the software representation supported and the entity type described in each of these relations.

Reference Relation

Project	Name	Reference	Ref_Type
*	*	*	*

Database Relations	Software Representations	Entity Type
a_reference	SADT	activity
d_reference	SADT	data item
b_reference	Data Flow Diagram	bubble
df_reference	Data Flow Diagram	data flow
pr_reference	Structure Chart	process
p_reference	Structure Chart	parameter
m_reference	Code	module
v_reference	Code	variable

* primary key

Figure 23. Reference Relation.

The primary key for this relation consists of a combination of all attributes contained in the relation. The project attribute, as in the previous relations discussed, identifies the project or individual responsible for the dictionary entry. The name attribute identifies the entity being described. The reference attribute designates the identity of the entity in the previous development stage.

The software representations supported by the dictionary designate an action entity by both a name convention and a numeric designation. For this reason, the reference for an action entity can be either a name or a number. Both action and data entities can contain a reference to a written requirements document by including under the reference attribute the number of the section of the document which applies to the entity being described.

The ref_type attribute identifies the particular representation and the method used (number or name) to designate a reference to a previous development stage. Figure 24 diplays the allowable ref_type attribute values for each of the four representations supported by the dictionary.

SADT - Activity

Requirements Number	DFD Bubble Number
DFD Bubble Name	

SADT - Data Item

Requirements Number	DFD Data Flow Name

Data Flow Diagram - Bubble

Requirements Number	SADT Activity Number
SADT Activity Name	

Data Flow Diagram - Data Flow

Requirements Number	SADT Data Item

Structure Chart - Parameter

Requirements Number	SADT Data Item
DFD Data Flow	

Structure Chart - Process

Requirements Number	SADT Activity Number
SADT Activity Name	DFD Bubble Number
DFD Bubble Name	

Code - Variable

SC Parameter Name	SADT Data Item
DFD Data Flow	Requirements Number

Code -Module

SC Process Number	SC Process Name
SADT Activity Number	SADT Activity Name
DFD Bubble Number	DFD Bubble Name

Figure 24. Ref_Type Attribute Values

Alias Relation.

The alias relation documents the situation in a software epresentation where an action or data element is identified by more than one name. The format for the alias relation exists in two forms, one for data entities and one for action entities. Figure 25 provides a graphical display of the attributes which make up the two forms of the alias relation. Also shown in figure 25 are the actual database relations which use the displayed formats, the name of the software representation which is supported by the relation, and the entity type described by the relation.

Alias Relation For Data Entities

Project	Name_1	Name_2	Comment	Where_Used
*	*	*		

Database Relations	Software Representations	Entity Type
d_alias	SADT	data item
df_alias	Data Flow Diagram	data flow
p_alias	Structure Chart	parameter
v_alias	Code	variable

Alias Relation For Action Entities

Project	Name_1	Name_2	Comment
*	*	*	

Database Relations	Software Representations	Entity Type
a_alias	SADT	activity
b_alias	Data Flow Diagram	bubble
pr_alias	Structure Chart	process
m_alias	Code	module

* primary key

Figure 25. ALias Relations.

Both forms of the alias relation use a combination of the project, name_1, and name_2 attributes as their primary key. These three attributes taken together can uniquely identify any tuple in the alias relation. The project attribute designates the team or individual responsible for this dictionary entry. The name_1 attribute contains the alias name or the "other name" by which a data or action entity can be identified. The name_2 attribute specifies the original or primary name which identifies an action or data entity.

The selection of these three attributes as the primary key provides a unique identifier for each tuple in the relation. The project attribute ensures that the information in the tuple not will be confused with another software project. The name_1 and name_2 attributes form a unique identifier within the software project. While it is conceivable that a entity could be identified by more than one alias name, an alias name can not be allowed to be associated with more than one original entity name.

Both the data and action forms of the alias relation, contain a comment attribute. The comment attribute provides a place for the tool user to include a comment concerning the alias name for a dictionary entity. This comment should attempt to explain why an alias name was used to identify the entity. This is a valid question. The entity obviously existed and was identified by an original name.

Why was the primary name not used to identify the data entity ? The use of alias names should be closely monitored and wherever possible should be eliminated. Having two names for one entity leads to communications problems and confusion in the operations of the data dictionary and the development of software in general.

The only difference between the two forms of the alias relation is the existence of an attribute field labeled where used in the data entity version of this relation. This attribute identifies the action entity or entities in a project which use the alias name to identify a data entity with which they interact. For example, assume that an SADT activity named "getdata" takes as an input a data item named "new data". Also assume that the data item "new data" is not the primary name for the data entity but an alias name for the data item "sales data". The alias relation depicted in figure 26 provides a graphical picture of how the alias relation would document this situation.

Project	Name_1	Name_2	Comment	Where_used
Team 1	New Data	Sales Data	Design Error	Get data

Figure 26. Alias Relation Example.

Value Set Relation.

The value set relation is used to identify the values a particular data entity can assume. This relation is only useful in providing meaningful information about a data entity when the set of values that a particular data entity can assume is both finite and reasonably small. If the set of values for a data item were infinite, the relation containing these values would have no size limit. In much the same manner, if the number of values associated with a data item was extremely large, the cost of storing this information in the database would exceed the benefit of having access to the information. However, if the number of values is small, maintaining them in the dictionary is beneficial. As a general rule of thumb, a data entity which can assume only ten or less values should have these values included in the dictionary database.

Because this relation is only concerned with data entities, it only supports the data entity portions of the software representations supported by the dictionary. Figure 27 displays a graphical representation of the attributes which make up the value set relation. Also included in figure 27 are the names of the database relations which use this format, the software representation supported, and the entity type described.

The project attribute identifies the person or group responsible for this entry into the dictionary. The name attribute identifies the data entity being described in the relation. The value attribute contains the value which the data entity identified in the name attribute can assume.

Value_Set_Relation

Project	Name	Value
*	*	*

Database Relations	Software Representations	Entity Type
d_value_set	SADT	data item
df_value_set	Data Flow Diagram	data flow
p_value_set	Structure Chart	parameter
v_value_set	Code	variable

*primary key

Figure 27. Value Set Relation.

The primary key for this relation consists of a combination of all the attributes which make up the relation. This is necessary to ensure that all tuples in the relation can be uniquely identified. Since it is not only possible but highly likely that a data entity will have more than one value associated with it, the inclusion of the value attribute in the primary key is necessary to ensure the unique identification property.

As stated earlier, this relation is only useful when a finite and reasonably small set of values exist which the data entity being described can assume. For example, if a data entity named city could only assume the names of four cities in a particular software application, the use of the value set relation would be appropriate. On the other hand, if the data entity could assume the name of any city or town in the United States, the set of values would be so large as to render the use of the value set relation worthless.

Figure 28 gives a visual example of how the value set relation would support the first case of the data entity city described in the previous example.

Project	Name	Value
Team 1	city	Boston
Team 1	city	New York
Team 1	city	Atlanta
Team 1	city	Washington

Figure 28. Value Set Relation Example

<u>Algorithm</u> <u>Relation</u>.

The algorithm relation contains information which explains how an action entity performs its function. An algorithm is a step by step procedure for solving problem or performing a task or operation in a finite amount of time. This relation allows the tool user to specify the step by step procedure by which the action entity being described operates. Because this relation is only concerned with action entities, it is only applicable to the action portion of the software representations. In fact, the algorithm relations is only applicable to the structure chart and code software representations. The SADT and data flow diagram methods of software representation are most useful during the requirements definition phase of the software life cylce. During this initial phase of development, the software designer has not determined what algorithms will be used to perform the desired actions. For this reason, the algorithm relation is not included among the database relations which support these software representations.

Figure 29 provides a visual display of the attributes which make up the algorithm relation. Also included in figure 29 are a list of the database relations which use this format, the software representation supported, and the entity type described.

Algorithm Relation

Project	Name	Line	Algorithm
*	*	*	

Database Relations	Software Representations	Entity Type
p_alg	Structure Chart	Process
m_alg	Code	Module

* primary key

Figure 29. Algorithm Relation.

The project attribute identifies the person or group responsible for the dictionary entry. The name attribute identifies the action entity described in a particular tuple of the relation. The line attribute identifies the particular text line of the total action algorithm which is contained in a particular tuple. The algorithm attribute contains 60 characters of a text which provides a portion of the overall algorithm for the action entity.

The primary key for this relation consists of a combination of the project, name, and line attributes. These three attributes taken together are able to uniquely identify every tuple in the relation.

Figure 30 provides an example of an example of what the algorithm relation would look like when supporting an actual dictionary entry.

Project	Name	Line	Algorithm
Team A	Sort	1	If A>B Then
Team A	Sort	2	Put A in File 1
Team A	Sort	3	If A<B Then
Team A	Sort	4	Put A in File 2
Team A	Sort	5	If A=B Then
Team A	Sort	6	Put A in File 3

Figure 30. Algorithm Relation Example

In format and operation, the algorithm relation is identical to the description relation discussed earlier. The only difference between these two relations is the nature of the information they maintain.

This concludes the discussion of the dictionary relations which are common among the four software representations. The remainder of the relations will be discussed within the context of a particular software representation. This does not mean that the remaining relations do not contain elements which are common among the various representations. However, the discussion of these relations is more effective when the strengths and constraints of the individual software representations are taken into consideration.

<u>SADT Relations</u>.

There are three dictionary relations which support the SADT software representation which have not already been discussed. These three relations are the activity, activity_io, and data_item relations. These relations will be discussed individually in the following sections.

<u>Activity Relation</u>.

The activity relation can be considered as the main relation in the dictionary for identifying the action entities or activities depicted in a SADT software representation. Figure 31 provides a graphical display of the attributes which make up this relation.

Activity Relation

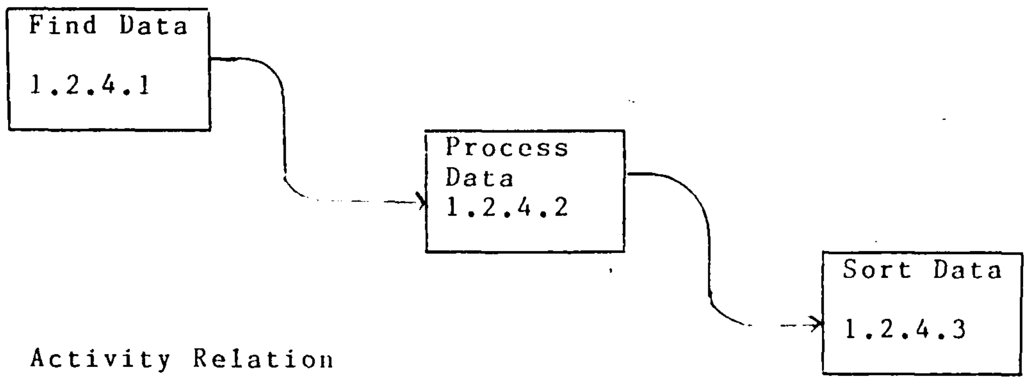
Project	Name	Number
*	*	

* primary key

Figure 31. Activity Relation.

The project attribute identifies the team or individual responsible for this entry into the data dictionary. The name attribute identifies a particular acitivity within a software project. The number attribute contains the activity number associated with a particular activity on a SADT diagram.

The project and name attributes form a unique identifier for each tuple in the activity relation. For this reason, a combination of these two attributes form the primary key for the activity relation. Figure 32 gives an example of an SADT diagram and how the activity relation would identify the various activities contained in the diagram.



Activity Relation

Project	Name	Number
Team 1	Find Data	1.2.4.1
Team 1	Process Data	1.2.4.2
Team 1	Sort Data	1.2.4.3

Figure 32. Activity Relation Example.

Activity_IO Relation.

The activity_io relation identifies the elements of an SADT software representation which interact with an action entity or activity. The elements are normally SADT data entities or data items. These data entities represent the inputs, outputs, controls, and mechanisms which are used and produced by an SADT activity.

Figure 33. provides a graphical display of the attributes which make up the activity_io relation.

Activity_IO Relation

Project	Aname	Dname	Element_Type
*	*	*	

* primary key

Figure 33. Acitivity_IO Relation.

The project attribute identifies the person or group of persons who are responsible for the dictionary entry. The Aname attribute contains the name which identifies the SADT activity being described. The Dname attribute contains the name of an SADT data entity which interacts with the activity identified by the Aname attribute. The element_type attribute classifies the manner in which the data entity identified in the Dname attribute interacts with the activity identified in the Aname attribute.

The classification of the interaction between an activity and a data entity in a SADT diagram or representation is determined by the position of the data entity with respect to the activity on a SADT diagram. The graphical display provided in figure 34 should help to clarify this concept.

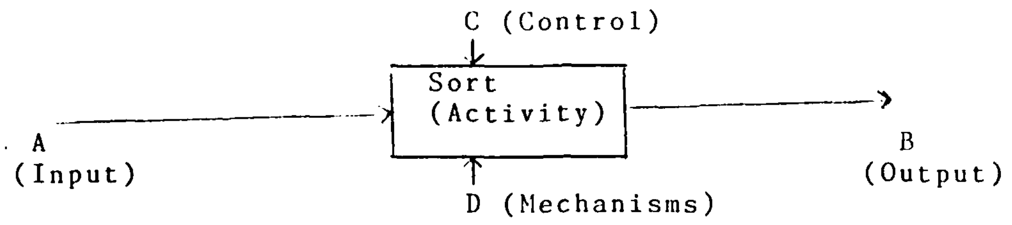


Figure 34. SADT Activity and Data Item Interactions.

As figure 34 demonstrates, there are four classifications for activity and data entity interaction in the SADT software representations: inputs, outputs, controls, and mechanisms. The element_type attribute will contain one of these classification for a particular tuple in the activity_io relations.

Figure 35 demonstrates how the acitvity_io relation would document the activity and data entity interactions depicted in the example in figure 34.

Activity_IO Relation

Project	Aname	Dname	Element_Type
Team 1	Sort	A	Input
Team 1	Sort	B	Output
Team 1	Sort	C	Control
Team 1	Sort	D	Mechanism

Figure 35. Activity_IO Relation Example.

The primary key for this relation is a combination of the project, aname, and dname attributes. The combination of these values form a unique identifier for each tuple in the activity_io relation.

Data_Item Relation.

The Data_Item relation contains information about the data entities used in an SADT representation of a software project. Each tuple in this relation contains information which describes a particular data entity. Figure 36 provides a graphical display of the attributes which make up the data_item relation.

Data_Item Relation

Project	Name	Data_Type	Low	High	Data_Span
*	*				

* primary key

Figure 36. Data_Item Relation.

The project attribute identifies the individual or group of individuals responsible for this entry into the dictionary. The name attribute identifies the particular SADT data entity which a tuple in the relation describes.

The data_type attribute attempts to classify the data entity in terms of the type storage structure required to represent the data entity in a programming language. This attribute may not even contain a value. Since the SADT representation is primarily used during the requirements phase of software development, it may be impossible to specify a data type for a data entity at that stage of development. However, if that information is available it enhances the description of the data entity. The dictionary supports the documentation of four standard data types: integer, real, character, and boolean. However, if these four types are not sufficient to describe the data type of the data item, the tool users may enter their own data type for a data entity. The data_type attribute will contain either one of the four standard data types or a user defined data type.

The low attribute contains the minimum value a particular data entity can assume. Like the data type attribute, the low attribute may not contain any value. For some data entities a minimum value will not exist. For example, a data item which represented the cities of the United States would not possess a minimum value which could be maintained in a low attribute field.

The high attribute contains the maximum value a data entity can assume. Like the low attribute a particular tuple in the data_item relation may not contain a value for the high attribute.

The data_span attribute contains a 60 character description of the range of values a particular data entity can assume. This attribute is extremely useful in describing the characteristics of a data entity. For example, if a data item represented the cities of the United States, it is obvious that neither a low or high attribute value could be specified. However, the data_span attribute could easily represent this situation by including the statement "all cities in the US" in its attribute field for the tuple which described this particular data entity. Like the low and high attributes, a tuple in the relation may not contain a value for the data_span attribute.

This concludes our discussion of the relations which support SADT method of software representation. The following sections will discuss the relations which support the data flow diagram method of software representation.

Data Flow Diagram Relations.

There are three relations which support the data flow diagram representation which have not been previously discussed. These relations are the bubble, data_flow, and bubble_io relations. From the discussion of these relations, it will become obvious that these relation are almost identical in format to the three relations dicussed in the previous section which supported the SADT representation. However, although similiar these relations support a respresentation which uses entirely different graphical symbols to represent the elements of a software project. These relations will be discussed in the following sections.

Bubble Relation.

The bubble relation is the primary relation for identifying the action entities depicted in the data flow diagram representation. Figure 37 provides a graphical display of the attributes which make up the bubble relation.

Bubble Relation

Project	Name	Number
*	*	

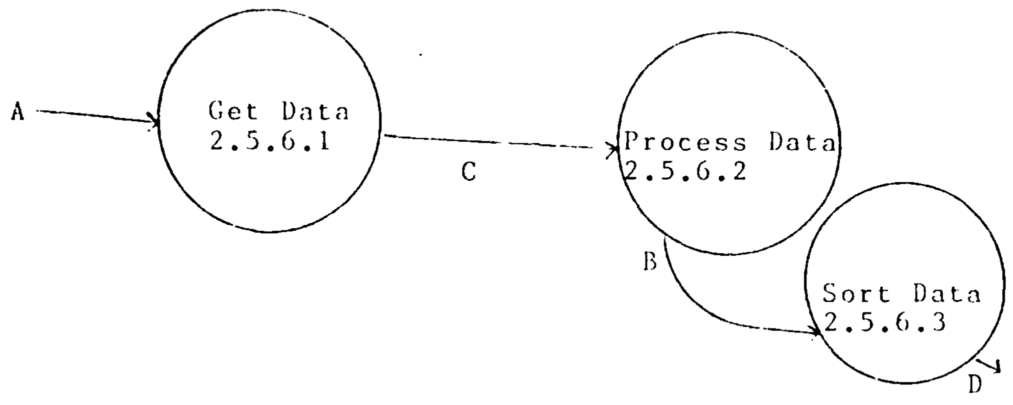
* primary key

Figure 37. Bubble Relation.

The project attribute identifies the person or group responsible for this entry into the dictionary. The name attribute identifies the particular data flow diagram action entity being described. The number attribute contains the number associated with the action entity on a data flow diagram.

The combination of the project and name attributes serves to uniquely identify every tuple contained in the bubble relation. For this reason, the combination of these two attributes serve as the primary key for this relation. Figure 38 displays a data flow diagram and how the bubble relation would document the actions entities depicted.

The bubble relation is identical in format to the activity relation discussed earlier. Both relations serve to identify the action entities associated with their respective software representations.



Bubble Relation

Project	Name	Number
Team 1	Get Data	2.5.6.1
Team 1	Process Data	2.5.6.2
Team 1	Sort Data	2.5.6.3

Figure 38. Bubble Relation Example.

<u>Bubble_IO Relation</u>.

The bubble_io relation contains information about the interaction between data entities and action entities in a data flow diagram representation. Figure 39 displays the attributes which make up the bubble_io relation.

Bubble_IO Relation

Project	Bname	Dname	Direction
*	*	*	

* primary key

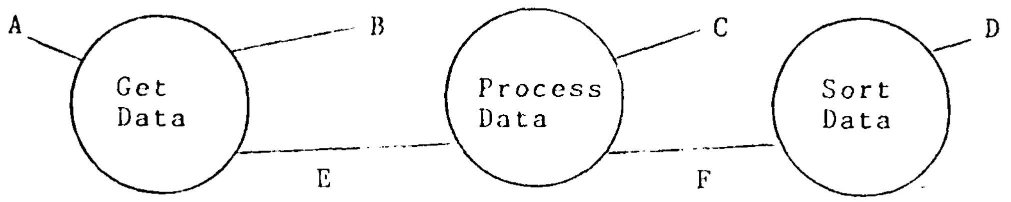
Figure 39. Bubble_IO Relation.

The project attribute identifies the person or group responsible for this entry in the data dictionary. The bname attribute identifes the structure chart action entity described by a particular tuple in the relation. The dname attribute identifies a structure chart data entity which is either an input or an output to the action entity being described. The direction attribute indicates whether the data entity identified in the dname attribute is an input or an output to the action entity described by a relation tuple.

A combination of the values contained in the project, bname, and dname attributes uniquely identify each tuple in the bubble_io relation. Both the action entity name and the data entity name are required for unique tuple identification, because an action entity may have several data entities it interacts with and a data entity may be used by more than one action entity. Because of this unique identification property, the project, bname, and dname attributes serve as the primary key for the bubble_io relation.

Figure 40 presents an example of a data flow diagram and how the bubble_io relation would document the interaction between the action and data entities depicts in the example data flow diagram.

The format of the bubble_io relation is almost identical to the format of the activity_io relation discussed earlier. Both relations maintain information about the interaction between data and action entities in their respective software representations. The only difference between the two is that the values in the direction attribute of the bubble_io relation only indicate if a data entity is an input or output of the action entity. The corresponding attribute, element_type, in the activity_io relation allows a data entity to be classified as a control, mechanism, input or output.



Bubble_IO Relation

Project	bname	dname	direction
Team 1	Get Data	A	Input
Team 1	Get Data	B	Output
Team 1	Get Data	E	Output
Team 1	Process Data	E	Input
Team 1	Process Data	C	Output
Team 1	Process Data	F	Output
Team 1	Sort Data	F	Input
Team 1	Sort Data	D	Output

Figure 40. Bubble_IO Relation Example

Data_Flow Relation.

The data_flow relation describes the data entities which exist in the data flow diagram software representation. The information contained in a tuple of this relation describes a particular data entity. The attributes which make up this relation are shown in figure 41.

Data_Flow Relation

Project	Name	Data_Type	Low	High	Data_Span
*	*				

* primary key

Figure 41. Data Flow Relation.

The project attribute identifies the person or group responsible for this entry in the data dictionary. The name attribute identifies the data flow diagram data entity described by a tuple in the relation. The data_type attribute indicates the storage structure the data entity would require in a programming language. The value of this attribute may be one or four standard data types directly supported by the dictionary; integer, real , character,and boolean; or a user input value for the data_type attribute. The low and high attributes contain the minimum and maximum values, respectively, which the data entity can assume. The data_span attribute consists of a sixty character description of the range of values the data entity can assume. The data_type, low, high, and data_span attributes may not contain a value for some data entities documented in this relation.

A combination of the attribute values for the project and name attributes serves to uniquely identify each tuple in the data_flow relation.

The format of the data_flow relation and the format of the data_item relation discussed in the section on SADT relations are identical. Both relations describe the data entities of their associated software representations.

This completes the discussion of all relations which support the data flow diagram method of software representation. The next section begins discussion on the remaining relations associated with the structure chart method of software representation.

Structure Chart Relations.

There are five relations which support the structure chart representation that have not been previously discussed. These relations are the process, process_io, pr_call, pr_passed, and parameter relations. These relations will be discussed in the following sections.

Process Relation.

The process relation identifies the action entities depicted in the structure chart software representation.

Figure 42 displays the attributes which make up the process relation.

The project attribute identifies the person or group of persons responsible for this entry in the data dictionary. The name attribute identifies the particular action entity or process being described. The number attribute contains the number associated with an action entity when it is depicted on a structure chart diagram.

Process Relation

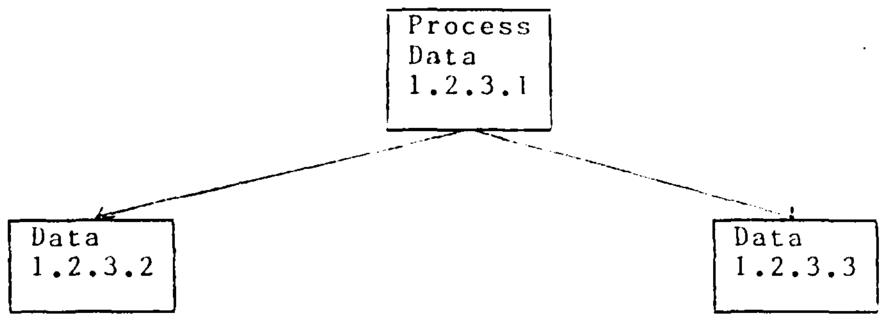
Project	Name	Number
*	*	

* primary key

Figure 42. Process Relation.

A combination of the values contained in the project and name attributes serve to uniquely identify each tuple in the relation. For this reason, a combination of these two attributes serve as the primary key for the process relation.

Figure 43 displays an example of a structure chart diagram and how the process relation identifies the action entities depicted by the structure chart example.



Process Relation

Project	Name	Number
Team1	Process Data	1.2.3.1
Team 1	Get Data	1.2.3.2
Team 1	Sort Data	1.2.3.3

Figure 43. Process Relation Example.

Process_IO Relation.

The process_io relation describes the interaction of a structure chart action entity and a structure chart data entity. Action entities are often called processes and data entities are often called parameters in structure chart terminology. The process_io relation identifies the parameters which provide input to or constitute the output from an action entity or process. A parameter in the process_io relation can be a file or hardware item as well as a parameter. The important characteristic is that the data entity described by this relation represents the action entities interface or interaction with the rest of the software project under development. Figure 44 presents a graphical display of the attributes which make up the process_io relation.

The project attribute identifies the person or group of persons who are responsible for this entry into the data dictionary. The name attribute identifies the action entity or structure chart process being described by a tuple

Process_IO Relation

Project	Name	Pname	Direction	P_Type	Class	Order
*	*	*				

* primary key

Figure 44. Process_IO Relation.

in the relation. The pname attribute identifies a parameter or other item which interacts with the action entity identified in the name attribute. The value contained in the pname attribute may identify a file or a hardware item as well as a structure chart parameter.

The remaining four attributes of the process_io relation (direction, p_type, class, order) describe the nature of the interaction between the action entity and the data entity and the nature or classification of the data entity. The values contained in these attributes are highly dependent upon the nature of the associated data entity.

The class attribute identifies the nature of the data entity interacting with the action entity. The values which can appear in the class attribute are: local, global, file, and hw (hardware). The local and global values indicate that the subject data entity is a structure chart parameter. If the value is global, it indicates that the parameter can be accessed by any process or action entity in the program or project under development. If the value is local, it indicates that the parameter can only be accessed by the portion of the program where it is known or identified. If the action entity is interacting with a file or hardware item, then the class attribute value will be file or hw respectively.

The direction attribute classifies the nature of the interaction between the action and data entity. This classification indicates how the action entity uses the subject data entity. There is a definite correlation between the values contained in the class attribute and the values contained in the direction attribute. Figure 45 displays the four different values contained in the class attribute and the corresponding allowable values for the direction attribute.

Class Attribute	Direction Attribute
local	input
	output
global	used
	changed
file	read
	write
hw	read
	write

Figure 45. Class and Direction Attribute Values.

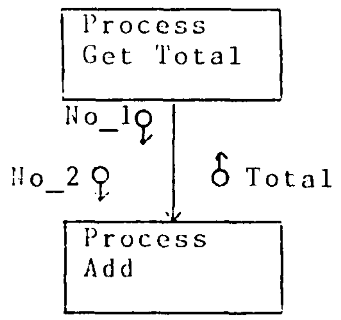
As figure 45 shows, a local parameter is considered to act as an input or output to the action entity. Since the value of a *global parameter* is known throughout the program, the area of interest is whether the action entity only uses the value attached to the global parameter or if the action entity actually changes the value. Files and hardware items communicate with the action entity by being read from or written to. It is not unusual for a data entity to possess both associated direction values for a particular class attribute value. For example, a structure chart process could easily read in values from a file, perform calculations using these values, then write the results back to the file.

The p_type attribute describes the nature of a parameter which interacts with action entity. It basically identifies if the parameter, either local or global, represents flag or data information for the action entity. Data information would be such things as the name of a city, a number required for a computation, or a month of the year. Flag information would be such things as the answer to a specific yes or no question such as the value true or false for a parameter which indicate if a certain number is negative or positive.

The order attribute identifies the order in which a local parameter is received by the action entity. For example, if a structure chart process interacted with three local parameters, one of the parameter would have a order of one, another would be the second in order, and the final local parameter would have an order value of three.

A combination of the values contained in the project, name, and pname attributes serve to uniquely identify each tuple in the process_io relation. The project attribute differentiates between the various software projects supported by the data dictionary. The combination of the action entity name, contained in the name attribute, and the data entity name contained in the pname attribute provides a unique identifer for all action and data interactions which take place within an individual software project. Because an action entity can interact with more than one data entity and a data entity can be used by more than one action entity, it requires the name values of both components to provide the unique identification property. For the above reasons, the project, name, pname attributes serve as the primary key for the process_io relation.

In using the process_io relation to support the structure chart method of software representation, it is necessary to establish a standard convention or framework for referencing the information portrayed on a structure chart diagram. The data entities which enter at the top of the structure chart process are those entities which interact with the action entity and are documented in the process_io relation. Figure 46 provides an example of a structure chart diagram and how the process_io would document the data and action entity interaction.



Process_IO Relation

Project	Name	Pname	Direction	P_Type	Class	Order
Team 1	Add	No_1	Input	Data	Local	1
Team 1	Add	No_2	Input	Data	Local	2
Team 1	Add	Total	Output	Data	Local	3

Figure 46. Process_IO Example 1.

As figure 46 shows, the process_io relation documents the interaction between a structure chart action and data entity which occurrs at the top of the box symbol for an action. In this example, all data entity inputs or outputs are received from or sent to the Get Total action entity. Since none of the data entities included in the example enter or exit from the top of the Get Total action entity, It does not appear in the process_io relation. The relationship between the data entities and action entity Get Total is depicted in the pr_passed relation which will be discussed later in this paper.

The situation depicted in figure 46 is a rather simple example. There are situations which may occur in the structure chart representation in which the name of the data entities documented in the process_io relation would not appear on the corresponding structure chart diagram. Such a situation is depicted in figure 47.

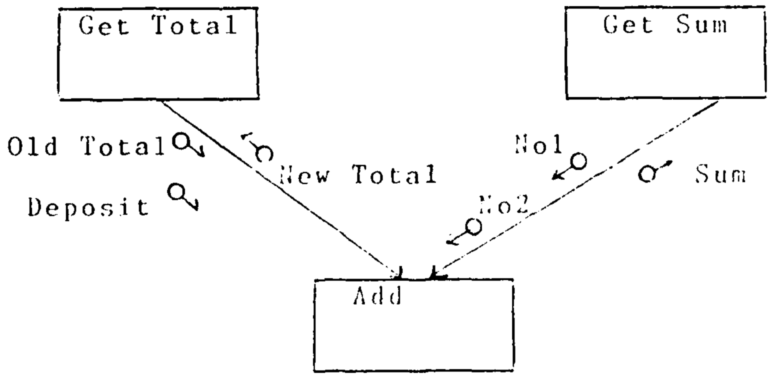


Figure 47. Structure Chart Diagram.

In the example depicted in figure 47, the Add process accepts as input two data entities representing whole numbers. The Add process will calculate the result of the addition of these two numbers and output this result. The structure chart depicted in figure 47 could be interpreted in such a manner as to indicate that the process Add accepts four inputs- No1, No2, Old Total, Deposit- and creates two outputs, Sum and New Total. From a structure chart point of view, this interpretation would be correct. However, the data dictionary relation process_io is not interested in the number of higher level processes which use the Add process nor the names of the parameters passed to and from the process from calling processes. This information is documented in the pr_call relation and the pr_passed relation which will be discussed in the following sections. The process_io relation documents the inputs needed and the outputs produced by the Add process. In the situation depicted in figure 47, the tool user would need to specify a name for the two input numbers and the output result which interact with the Add process. These names would be different from any of the names depicted in the structure chart in figure 47. This creation of names for the input and output parameters of process Add is needed to uniquely identify the interface characteristics of the process.

If the situation as depicted on the structure chart in figure 47 was documented in the process_io relation, it would give the erroneous conclusion that the Add process required four input parameters to perform its function and created two output parameters.

A possible set of process_io entries which would correctly document the situation portrayed in figure 47 is shown in figure 48.

It is important to remember that the situations discussed previously represent problems encountered in representing the structure chart representation in the data dictionary. The solutions dicussed for these problems represent standard conventions which were established for using the Data Dictionary Generation Tool and not standard conventions which can be applied to the Structure Chart method of software representation.

Process_IO Relation

Project	Name	Pname	direction	Ptype	Class	order
Team 1	Add	Addend1	Input	Data	Local	1
Team 1	Add	Addend2	Input	Data	Local	2
Team 1	Add	Result	Output	Data	Local	3

Figure 48. Process_IO Example 2.

Pr_call Relation.

The pr_call relation depicts the use of or call to an action entity by another action entity. Often an action entity, in performing its function, will use another action entity to perform a calculation or other jobs. This is known as a call by one action entity to another action entity. The attributes which make up the pr_call relation are displayed in figure 49.

Pr_Call Relation

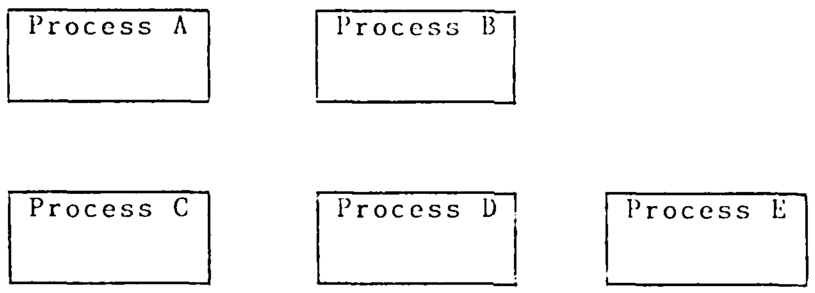
Project	Calling	Calls
*	*	*

* primary key

Figure 49. The Pr_Call Relation.

The project attribute identifies the person or group of persons responsible for this entry in the data dictionary. The calling attribute identifies the action entity or structure chart process which enlists the aid or calls another process in performing its function. The calls attribute identifies the process used by the process identified in the calling attribute.

The primary key for this relation consists of a combination of all attributes contained in the relation. The project attribute is needed to differentiate between different software projects contained in the data dictionary. Because a process can call more than one process and a process can be used by more than one process, both the calling and calls attributes are needed in the primary key to uniquely identify each tuple in the pr_call relation. Figure 50 provides an example of a structure chart diagram depicting the calling of processes by other processes and how the pr_call relation would document the situation.



Pr_Call Relation

Project	Calling	Calls
Team1	A	C
Team1	A	D
Team1	B	D
Team1	B	E

Figure 50. Pr_Call Relation Example.

Pr_Passed Relation.

The pr_passed relation describes the data entites sent to and returned from an action entity when it is called or used by another action entity. The attributes which make up the pr_passed relation are displayed in figure 51.

Pr_Passed Relation

Project	Name	Destination	Source	Order	P_Type	Class
*	*	*	*			

* primary key

Figure 51. Pr_Passed Relation.

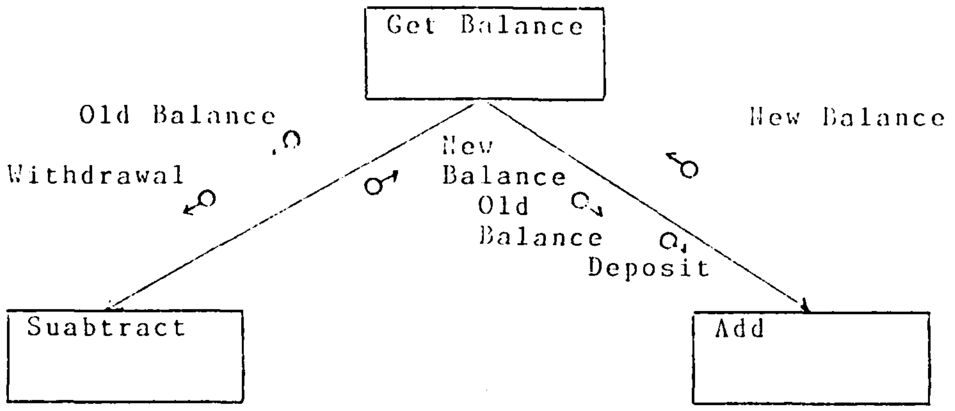
The project attribute identifies the person or group of persons responsible for this entry into the data dictionary. The name attribute identifies the data entity or parameter which is being transfered in a call between two processes. The destination attribute identifies the process which is receiving the parameter identified in the name attribute. The source attribute identifies the process from which the parameter was transmitted or sent. The order attribute indicates the order of this particular parameter among all parameters involved in this call between two processes. The p_type attribute indicates if the parameter being passed represents flag or data information. The class attribute indicates if the parameter identified in the name attribute is returned by the called process to the calling process or if the parameter is passed from the calling process to the called process.

A combination of the values contained in the project, name, destination, and source attributes serve to uniquely identify each tuple in the pr_passed relation. The project attribute serves to differentiate between different projects which are contained in the data dictionary. Because a parameter may be passed from and passed to several different processes, the names of the parameter, its source process, and its destination process are required to uniquely identify a tuple in the pr_passed relation.

On a structure chart diagram, the pr_passed relation is interested in those data entities which depart and arrive at the bottom of the box structure which represents a process.

Figure 52 displays a sample structure chart diagram and how the pr_passed relation would document the passing and return of parameter during calls between processes.

As figure 52 shows, the pr_passed relation documents the parameters passed between processes during process calls. The point of reference for documenting this situation in the pr_passed relation is the bottom portion of the box which represents the calling action entity in a process call.



Pr_Passed Relation

Project	Name	Destination	Source	Order	P_Type	Class
A	Old Balance	Subtract	Get Balance	1	Data	passed
A	Withdrawal	Subtract	Get Balance	2	Data	passed
A	New Balance	Get Balance	Subtract	3	Data	return
A	Old Balance	Add	Get Balance	1	Data	passed
A	Deposit	Add	Get Balance	2	Data	passed
A	New Balance	Get Balance	Add	3	Data	return

Figure 52. Pr_Passed Relation Example.

Parameter Relation.

The parameter relation describes the local and global parameters, files, and hardware items which exist in a structure chart representation of a software project. Figure 53 provides a visual display of the attributes which make up this relation.

Parameter Relation

Project	Name	Data_Type	Low	High	Data_Span	Class
*	*					

* primary key

Figure 53. Parameter Relation.

The project attribute identifies the person or group of persons responsible for this entry into the data dictionary. The name attribute identifies the parameter, file, or hardware item described by an individual tuple in the relation.

The data_type attribute indicates the storage type which would have to be assigned to a parameter before it could be described in terms of a programming language. It would not be unusual for this attribute to contain no value for a specific parameter being described. The dictionary tool supports the inclusion of four values in this attribute: integer, real, boolean,and character. However, if these values to do sufficiently describe the parameter, the user has the option of entering his own value for the data_type attribute.

The low and high attribute contain the minimum and maximum values, respectively, that the parameter being described in a relation tuple can assume. It is not unusual for no value to be assigned to these attributes, as some parameter will not possess a minimum and maximum value. Also, depending upon the stage of development of the project, it may not be possible to specify these values.

The data_span attribute allows the tool user to enter a sixty character description of the range of values a particular parameter may assume. The attribute may not contain any values depending upon the nature of the parameter being described and the current stage of development of the project.

The class attribute designates the entity being described as either a file, hardware item, local parameter, or global parameter.

This concludes our discussion of the relations which support the structure chart method of software representation. The next and final software representation to be discussed will be the code representation. A great deal of commonality exists between the relations which support the code and structure chart methods of software representation. The area of commonality between the two is pointed out in the discussion of the relations supporting the code method of representation.

Code Representation Relations.

There are five remaining relations which support the code method of software representation which have not yet been discussed. These relations are the module, module_io, m_call, m_pass, and variable relations. Each of these relations will be discussed in the following section. The code representation constitutes the actual translation of project design into a functional programming language. The code relations were designed to support a wide variety of programming languages. The actual use of these dictionary relation may vary depending upon the accepted conventions of the language being supported.

Module Relation.

The module relation documents the action entities found in the code representation. The attributes which make up the module relation are described in figure 54.

Module Relation

Project	Name	Number	Filename	Type	Library
*	*				

* primary key

Figure 54. Module Relation.

The project attribute identifies the person are group of persons responsible for this entry into the data dictionary. The name attribute identifies the particular action entity being described by an individual tuple in the relation. The number attribute contains the module number associated with the action entity identified in the name attribute.

The filename attribute identifies the name of the computer system file where the actual code for this module can be found. This attribute would be especially valuable in locating a module which was part of a project where the code was contained in several different file and linked together for execution.

The library attribute indicates if the module being described was created as part of the software project or if the module is part of a system library of common module which were already available for use. Many system maintain an extensive collection of commonly used modules for the programming languages it supports.

The type attribute indicates if the module being described is a function or a procedure. A procedure is an action entity that is executed when called by another action entity. A function is a action entity which returns a value when called by another action entity.

The project and name attributes form the primary key for the module relation. Since each tuple in the relation describes a particular action entity, only these two attributes are needed to uniquely identify each tuple in the relation.

The module relation is very similar to the process relation discussed in the structure chart representation relations. Both relation seek to provide information about the action entities which constitute a software project.

Module_IO Relation.

The module_io relation describes how an action entity or code module interfaces with the rest of the software project. It identifies the data entities the module requires to perform its function and the data entities which constitutes the output of the module being described. Figure 55 displays the attributes which make up the module_io relation.

Module_IO Relation

Project	Name	Vname	Direction	P_type	Class	Order
*	*	*				

* primary key

Figure 55. Module_IO Relation.

The module_io relation is identical in format and meaning to the process_io relation discussed earlier. Both relations describe the interface characteristics of the action entities in their respective representations.

The project attribute identifies the person or group of persons responsible for this entry into the data dictionary. The name attribute identifies the module or action entity whose interface characteristics are being described. The vname attribute identifies a data entity or code variable which is either a required input or an output for the action entity being described. The value in the vname attribute can represent a file, local or global variable, or hardware item.

The direction attribute describes the nature of the interface between the data entity identified in the vname attribute and the action entity identified in the name attribute. The actual value contained in the direction attribute will depend on the nature of the data entity. For example, the direction value for a data entity representing a file or hardware item would be either read or write to indicate whether the module writes to or reads from the subject file or hardware item. In the same manner, a value of input or output would be associated with a local data entity or variable. The value of interest with respect to global variables would be whether or not the module or action entity used or changed the value of the global variable it interfaced with.

The p_type attribute indicates if a local variable involved in interfacing with the action module contains data or flag information. The order attribute indicates the order of this variable among the other local varibale which form the interface with the module.

The class attribute indicates if the data entity identified in the vname attribute represents a file, hardware item, local variable or global variable.

The primary key for the module_io relation is a combination of the values contained in the project, name, and vname attributes. A combination of these values uniquely identifies each tuple in the module_io relation.

M_Call Relation.

The m_call relation indicates which modules in a software projcect call or use other modules and the names of the modules called by a particular module. Figure 56 displays the attributes which make up the m_call relation.

M_Call

Project	Calling	Calls	Type
*	*	*	

* primary key

Figure 56. M_Call Relation.

The m_call relation is similar in format and meaning to the pr_call relation discussed in the previous section on relations supporting the structure chart representation. Both relation describe the calling or use of action entities by other action entities.

The project attribute indentifies the individual or group of individuals responsible for this entry into the data dictionary. The calling attribute identifies a module or action entity which calls or uses another action entity in performing its function. The calls attributes identifies the name of one of the action entities used by the code module identifed in the calling attribute. The type attribute designates the module identified in the calling attribute as either a function or procedure.

The combination of the values contained in the project, calls, and calling attributes uniquely identify each tuple in the m_call relation. For this reason, these attributes serve as the primary key for the relation.

M_Pass Relation.

The m_pass relation describes the transfer of data entities between modules that takes place when one module calls or uses another module. The attributes which make up the m_pass relation are displayed in figure 57.

M_Pass Relation

Project	Name	Destination	Source	Order	P_Type	Class
*	*	*	*			

* primary key

Figure 57. M_Pass Relation

The m_pass relation is identical in format and a similar in meaning to the pr_passed relation discussed in the previous sections concerning the dictionary relations which support the structure chart method of software representation. Both relation describe the transfer of information involved a call between two action entities.

The project attribute identifies the person or group of persons responsible for this entry into the data dictionary.

The name attribute identifes the data entity which is being send or returned during a call between two modules. The source attribute identifies the module which sent or transmitted the data entity and the destination attribute identifies the module which receives the data entity identified in the name attribute. The order attribute indicates the order of the data entity with respect to the other data entities involved in the module call. The p_type attribute indicates if the data entity or variable represents data or flag information.

The class attribute indicates if the data entity is passed from the calling module to the called module or if the data entity is returned from the called module to the calling module. This attribute also indicates if the value being returned to the calling module is the result of a call to an action entity which is a function. When a function is called, the value returned by the function does not take the form of a physical variable. The function simply returns itself as a value to the calling module.

The primary key for this relation is a combination of the values contained in the project, name, source, and destination attributes. In order to uniquely identify all tuples contained in the m_pass relation, the combination of the values in these attributes are required.

<u>Variable Relation</u>.

The variable relation describes the individual data entities used in the code representation of a software project. The attributes which make up the variable relation are displayed in figure 58.

Variable Relation

Project	Name	Data_Type	Low	High	Data_Span	Class
*	*					

* primary key

Figure _58. Variable Relaion.

The variable relation is identical in meaning and format to the parameter relation discussed in a previous section. Both relations describe the data entities for their respective software representations.

The project attribute identifies the person or group of persons responsible for this entry into the data dictionary. The name attribute identifies the particular data entity being described by a tuple in the relation. The data_type attribute indicate the storage structure required to represent the data entity in a particular programming language. The low and high attributes identify the minimum and maximum values the data entity can assume. The data_span attribute provides a sixty character description of the range of values the data entity can assume. The class attribute indicates if the data entity identified in the name attribute is a file, local variable, global variable, or a hardware item.

The combination of the values contained in the project and name attribute provide a unique identifier for each tuple in the variable relation. For this reason, these attributes serve as the primary key for this relation.

This concludes the discussion of the relations which support the code representation. This also concludes the discussion on the relations which comprise the data dictionary database.

The following sections will examine some of the design alternatives which were considered in the design of the dictionary database.

Database Design Alternatives

Several design alternatives were considered during the design of the dictionary database. Two of these alternatives will be examined in the following sections. The advantages and disadvantages of these alternatives will be discussed and the rationale behind their rejection presented. The two alternatives are representative of the types of design decision and tradeoffs which were encountered during the dictionary database design.

Sharing Common Relations Among The Various Software Representations Supported.

The first design alternative or decision to be made was to determine if a relation should be allowed to contain information about more than one software representation. The current design of the dictionary database provides a separate set of relations to support each of the four methods of software representation. As was continually pointed out during the database design section, a great deal of commonality exists between the relations which support the various software representation methods. In some cases, the relations are identical in both format and meaning.

For example, the format of the description, hierarchy, history, and reference relations are identical in format for both the action and data entities of all four software representations supported by the dictionary.

If the concept of allowing a single relation to contain information about more than one type of software representation had been adopted, the number of relations needed to support the dictionary could have been greatly reduced. As an example, a single common reference relation could have replaced eight individual relations which are now included.

This alternative possessed two main advantages, reduction in the number of relations required to support the data dictionary and possibly the reduction of the amount of application software required to manipulate the contents of the dictionary database. The reduction in application software would have come about because the need to access different relations based upon the type of software representation used would not have been required. For example, in the current database separate application software is needed to perform retrieval and input operations on the description relations for each of the four software representation supported by the data dictionary. Under the common relations alternative a single software procedure could have performed this function for all of the different software representations.

Although the above alternative possesses its advantages, the accompanying disadvantages were judged to be sufficient to warrant rejection. There were three main disadvantages to this alternative.

The use of the common relation alternative would have decreased the efficiency of the data dictionary in performing retrieval, modification, and deletion operations on the dictionary database. The use of common relations would have reduced the number of relations in the database, but it would not have reduced the amount of information the dictionary must maintain. In other words, the number of tuples contained in the common relation would be equal to the sum of the number of tuples contained in the various individual relations it replaced.

Under the common relation alternative, the tradeoff would have been between several moderate sized relation or a reduced number of extremely large relations. The increased size of the common relation would have, in general, increased the amount of time to search the relation and locate the particular tuple of interest. This increase in time would have correspondingly reduced the efficiency of the modification, retrieval, and deletions of information from the dictionary database.

Another disadvantage of the common relation approach would have been the increased requirement for system memory. When the dictionary information is maintained in individual relations based upon the software representation and the whether or not the entity being described represents data or actions, there is no need to explicity state this information in the relation itself. For example, the pr_desc relation supports only the description of action entities involved in the structure chart method of software representation. If the common relation approach is taken, two additional attributes must be added to each tuple in the relation: one to identify the type of entity supported (data or action) and one to identify the software representation method (SADT, Code, etc).

The third and final reason for not utilizing the common relation alternative had to do with future development of the automated data dictionary tool. By maintaining the individual relations for each representation, the memory requirements and efficiency constraints can be better studied for the different representation supported.

Use of Attributes In One Relation to Indicate the Existence of Information In Another Relation.

The use of attributes in a relation to provide information about the information which existed in another relation was actually implemented on a small scale during the early development of the data dictionary. As an example of this concept, figure 59 displays two versions of the SADT data_item relation. In figure 59, (A) represents this relation as it now exists in the dictionary and (B) displays an alternative design.

(A) Data_Item Relation (Actual Format)

Project	Name	Data_Type	Low	High	Data_Span

(B) Data_Item Relation (Alternative)

Project	Name	Data_ Span	Low	High	Data_ Type	Is_Value Set	Is_ Alias	Is_ Reference

Figure 59. Data_Item Relation Design Alternative.

The last three attributes displayed in part B of figure 59 constitute the only differences between the forms of the data_item relation displayed in figure 59. The three attributes would contain a value which represented true or false (T or F). They would indicated if any information about the data item identified by this tuple existed in the value_set, alias, or reference relations for SADT data entities.

The advantage of this alternative is that it allows these excess attributes to act as a flag or signal for the dictionary application software. For example, assume that the dictionary is attempting to retrieve information about a particular data item. Upon accessing the proper tuple in the data_item relation, the application software can determine if it is necessary to access the reference, alias and value_set relations. If the value in these attributes is false, the application software has increased its efficiency in performing its operation because it knows it does not need to search these relations for additional information.

The disadvantage of this alternative is that it greatly reduces the data independence of the relations. Now the database relations are closely tied to the applications software. Another problem introduced by this alternative is the potential for data inconsistency. For example, if the value_set relation is modified so that it contains information about a particular entity and the attribute in the data_item relation is not changed to reflect this change, the information in the value_set relation will exist in the dictionary, but never be retrieved. These two disadvantages far outweighed any improvement in efficieny which could be gained by its use. For these reasons, the alternative was rejected.

The following section will discuss the design of the data dictionary user interface or the user's view.

Design of the User Interface and User's View of the Data Dictionary.

This section will describe the manner in which the data dictionary user interacts, through application software, with the dictionary database. Several different methods or dialog styles can be used in desgining an effective user interface. "Dialog styles describe the nature of the interface between the system and the user" (20:198). Some of the most popular dialog styles used in designing an effective user interface are: question and answer, command language, menu, and input form/output form (20:199-202).

With Q/A (question/answer) dialogs, the systems asks the user a question. The user responds to this question. This process continues until the system obtains sufficient information to perform the desired operation for the user. "Q/A dialogs tend to be most successful for inexperienced or infrequent users..." (20:200). Q/A dialogs tend to be least successful for sophisticated or frequent users, who get tired of proceeding through the questions.

A popular dialog style is that of using menus. A menu dialog lets the user select form a menu of alternatives instead of having to type commands or other information. "The menu dialog seems to be quite effective for inexperienced or infrequent users..." (20:200).

"The command language dialog style uses a command language for invoking system functions. "The usual format of command dialog involves verb-noun pairs (e.g. Plot Sales) with a short spellings (e.g. six to eight characters) for the nouns and verbs(20:200)." "For simple applications, a command language is easily learned, but it will probably need to be relearned by infrequent users. For complicated applications, a command language can easily become a programming language thereby requiring more skill to use"(20:200).

Input form/output form dialogs provide input forms in which the user enters command and data, and output forms on which the system provides responses. "Input form/output form dialogs can be very successful if there is a correspondence between the input/output form ... and paper forms or thought patterns which are familiar to the user" (20:202).

The data dictionary generation tool utilizes a combination of the question and answer, menu, and command language dialogs to provide user interface with the system. The question and answer and menu dialog styles provide the primary interface between system users and the data dictionary database. These styles allow the user to communicate the information the system needs to add, delete, modify, and retrieve data dictionary information from the database. The command language dialog is primarily used in performing dictionary maintenance or administrative functions. The query language used by the database management system which supports the dictionary database forms the command language dialog. These commands are used by dictionary administrator or maintainer to modify, delete, or add relations in the dictionary database.

The Data Dictionary Generation Tool provides the system user with a single view or method of looking at the information maintained in the dictionary database. This view consists of the definition of a particular data action entity for any of the four represe tations supported. The system user is not aware of the storage structure used in the dictionary nor are they aware that the data dictionary information about a particular entity is scattered among several different relations in the database. As far as the user is concerned, the data dictionary is simply composed of the definitions of the action and data entities associated with a given software representation for a project.

In the section on Data Dictionary Information Content , the information content required to support the data and action entities for all four software representation were discussed. The information elements which were required for each of these action and data entities constitute the entity definition. These information elements were displayed in figures 10 and 11.

When a dictionary user inputs an entity definition, the dictionary, by means of the question and answer and menu dialog styles, obtains the information necessary to satisfy all the information content requirements for the specific entity type and software representation. When the user retrieves information from the dictionary, the system provides the definition of action or data entities maintained in the dictionary database. For example, if a user wanted to know the inputs and outputs associated with an activity in a software project which was represented by the SADT representation, the system would need to obtain the following information: 1. That the inforamtion dealt with a action entity, 2. That the SADT software representation was used, 3. The name of the software project involved, and 4. The name of the action entity. When this information was received by the system, it would respond by retrieving the complete definition of the action entity and presenting it to the user. The information desired by the user, in this example the inputs and outputs associated with a SADT activity, would be contained in the definition retrieved from the dictionary database. In other words, all dictionary operations, with the exception of modification, consider the definition as the basic informational unit to be placed in and retrieved from the dictionary database. The modification operation does allow user to change or update components of entity definitions, but all other operations only manipulate entity definitions.

Data Dictionary Generation Tool Structural Model

The objective of the Data Dictionary Generation Tool structural model is to illustrate the hierarchial composition of the Data Dictionary Generation Tool components into a functional environment. This structural model serves as the framework for the development of the Data Dictionary Generation Tool. This model identifies the managerial and functional modules needed to support the objectives of the system as stated in chapter two of this paper.

Methodology Utilized for The Structural Model.

Three design techniques are candidates for representing the structural model of the Data Dictionary Generation Tool. These techniques are IBM's HIPO (Hierarchy plus Input Process Output), Higher Order Software's HOS technique, and the classical structure chart technique. All of the above named techniques utilize a hierarchy of design modules and specify the inputs and outputs of each module.

The HIPO technique uses a special digraph called a tree to illustrate it's Function Chart (25:139). An example fo a function chart is displayed in figure 60.

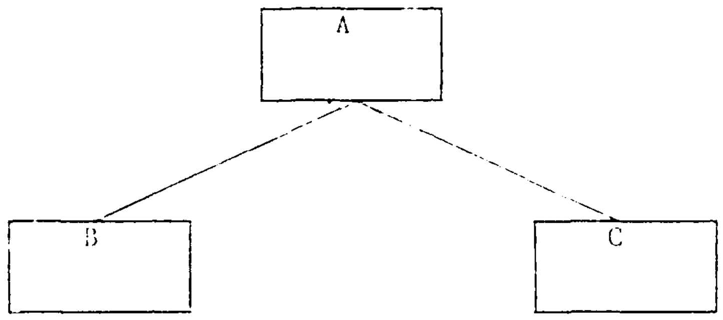


Figure 60 Sample HIPO Function Chart

In figure 60, module A is the main module and it calls modules B and C. Each module is described in greater detail by the IPO (Input Process Output) chart. This chart identifies the inputs, processes, and outputs of each module. A sample IPO chart is shown in figure 61.

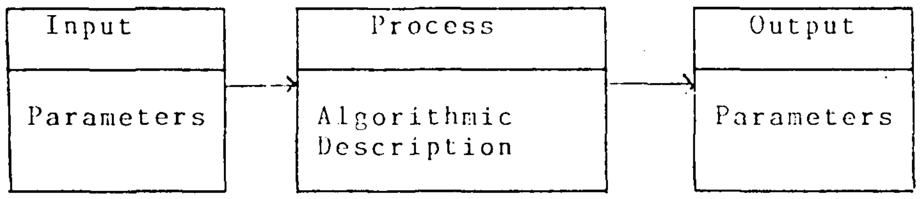


Figure 61. IPO Diagram Sample.

The disadvantage of the HIPO technique are that it does not specify an ordering or conditions on the calling of subordinate routines nor does it depict the passing of parameters between modules.

The HOS technique utilizes a hierarchial structure similar to the function chart displayed in figure ___. However, each box represents a function with the inputs and the outputs placed to the left and right of the box respectively. This use of the box is illustrated in figure 62.

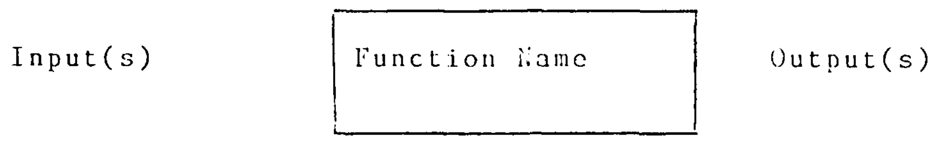


Figure 62. HOS Function Specification.

The disadvantage of the HOS technique is that it does not differentiate between control and data parameters.

The structure chart method also uses the basic function chart hierarchy shown in figure 60. In addition, the structure chart method possesses conventions for illustrating the passing of parameters between modules and a means of differentiating beween data and control parameters.

Of three candidate methods for representing the structural model of the tool, the structure chart method is best suited for display the important aspects of the structural model. The Data Dictionary Generation Tool Structural Model will need to clearly depict the passing of parameters between the various functional modules and the dictionary database. Since this is a strong point of the structure chart representation, it will serve as the methodology for the structural model. Because the structure chart method is one of the four representations support by the dictionary, its selection is even more appropriate as the method of depicting the tool's structural model.

Data Dictionary Generation Tool Structural Model

The structural model is a specification of the functional and managerial modules which make up the Data Dictionary Generation Tool. The structural model provides a functional framework for the application software needed to meet the objectives of the tool and to provide an interface between the tool users and the dictionary database.

The structural model, with some exceptions, display all the functional modules required to meet the objectives for the tool as stated in chapter two. The above mentioned exceptions are those dictionary maintenance and administrative functions performed through the use of the database management system (DBMS) which supports the dictionary database. These functions will be discussed in chapter 5 when an the DBMS selection and use are discussed.

Figure 63 displays the top level of the Data Dictionary Generation Tool structural Model.

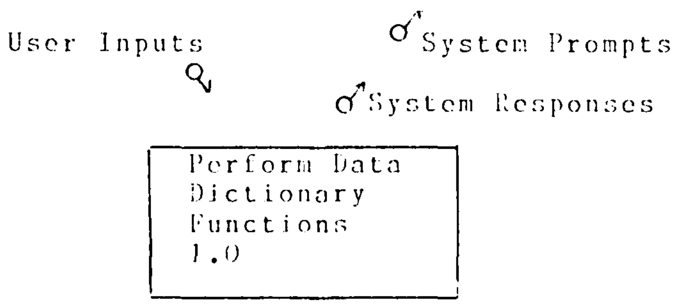


Figure 63. Top Level Structural Model (Structure Chart).

The tool user is provided with prompts by the system. These prompts are in the form of menu selections and questions. The user inputs are the tool users answer to the system prompts. Based on the user's input, the tool will perform the requested data dictionary operation. These basic operations will include the addition, retrieval, deletion, modification, listing, and printing of information contained in the dictionary database. Figure 64 presents the decompostion of the structural model top level.

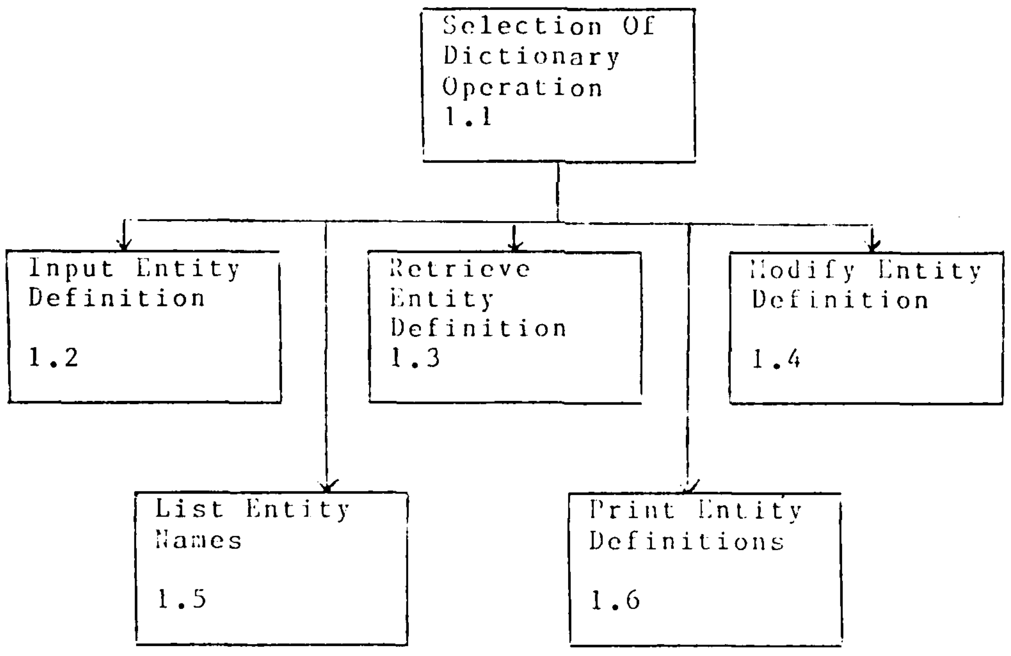

Selection Of Dictionary Operation 1.1
Input Entity Definition 1.2
Retrieve Entity Definition 1.3
Modify Entity Definition 1.4
List Entity Names 1.5
Print Entity Definitions 1.6

Figure 64. Perform Data Dictionary Functions (Structure Chart).

In figure 64 the Selection of Dictionary Operation module, represents the managerial modules which, based upon the user's inputs, select the correct functional modules required to satisfy the user's request. The remaining modules represent the collection of modules which perform the basic dictionary operations on the particular software representation and entity type as indicated by the tool user. For example, the Input Entity Definition module is a representation of all the functional modules which query the user for the information required to formulate a entity definition, format this user information in a form acceptable by the dictionary database, and append this information to the individual relations which make up the dictionary database.

A separate collection of functional modules exists to support the dictionary operation for each entity type (action or data) and software representation supported (SADT, Data Flow Diagrams, Structure Charts, or Code).

Figure 65 displays the decompostion of the Selection of Dictionary Operation module initially displayed in figure 64. These modules perform the managerial tasks necessary to determine the three pieces of control information required to select the correct functional module to perform the operation desired by the tool user. The Determine Operation from User Module presents the tool user with a menu of operation which the tool can perform. The user selects the desired operation by responding to the menu with the number associated with the desired menu option. The module accepts the user response and passes it to module Determine Entity Type From User.

This module queries the tool user to find out if the entity of interest is an action or a data entity. This module passes this information, along with the selection of operation information, to module Determine Software Representation From User and Call Functional Module.

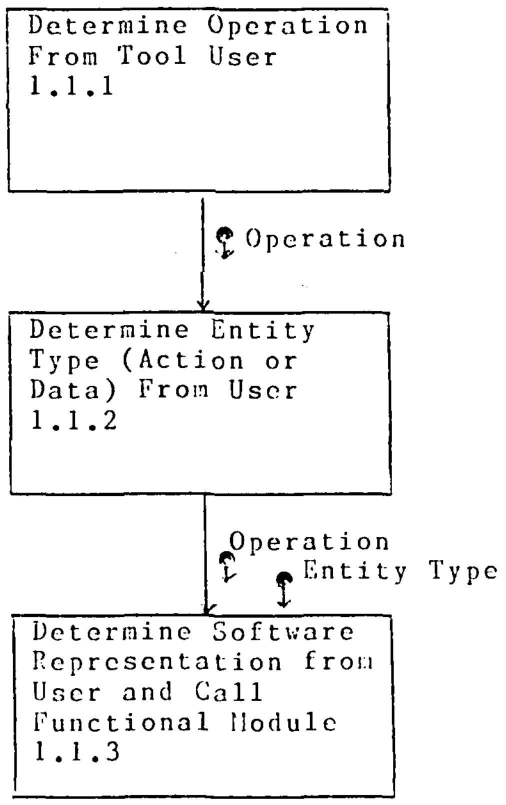

Determine Operation From Tool User 1.1.1
Operation
Determine Entity Type (Action or Data) From User 1.1.2
Operation
Entity Type
Determine Software Representation from User and Call Functional Module 1.1.3

Figure 65. Selection of Dictionary Operation (Structure Chart).

This module will query the user to determine which of the four software representations supported by the tool the user wishes to deal with. When this information is obtained from the user, this module is in possession of the three pieces of control information needed to select the proper functional module to perform the user's desired operation: dictionary operation, entity type of effected entity, software representation used to support the entity. Based upon this information, the module calls the proper functional module.

Figure 66 displays the logical decomposition of the Input Entity Definition module initially displayed in figure 64. The modules displayed represent the eight functional groupings of modules needed to input an entity definition for both the action and data entity for all four software representations supported by the data dictionary.

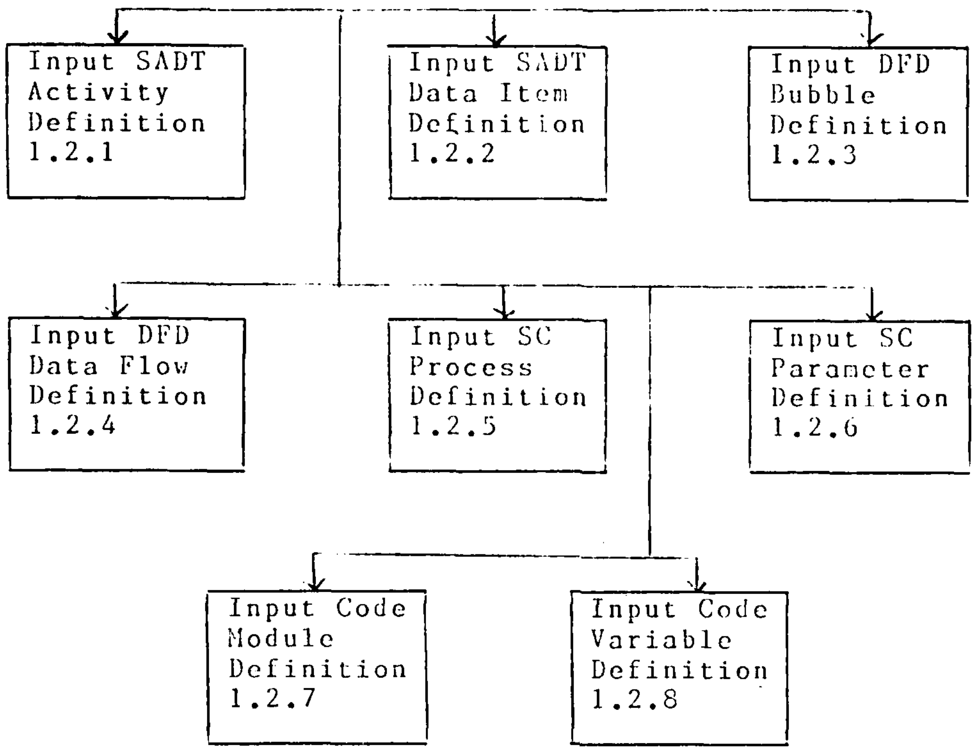

Input SADT Activity Definition 1.2.1
Input SADT Data Item Definition 1.2.2
Input DFD Bubble Definition 1.2.3
Input DFD Data Flow Definition 1.2.4
Input SC Process Definition 1.2.5
Input SC Parameter Definition 1.2.6
Input Code Module Definition 1.2.7
Input Code Variable Definition 1.2.8

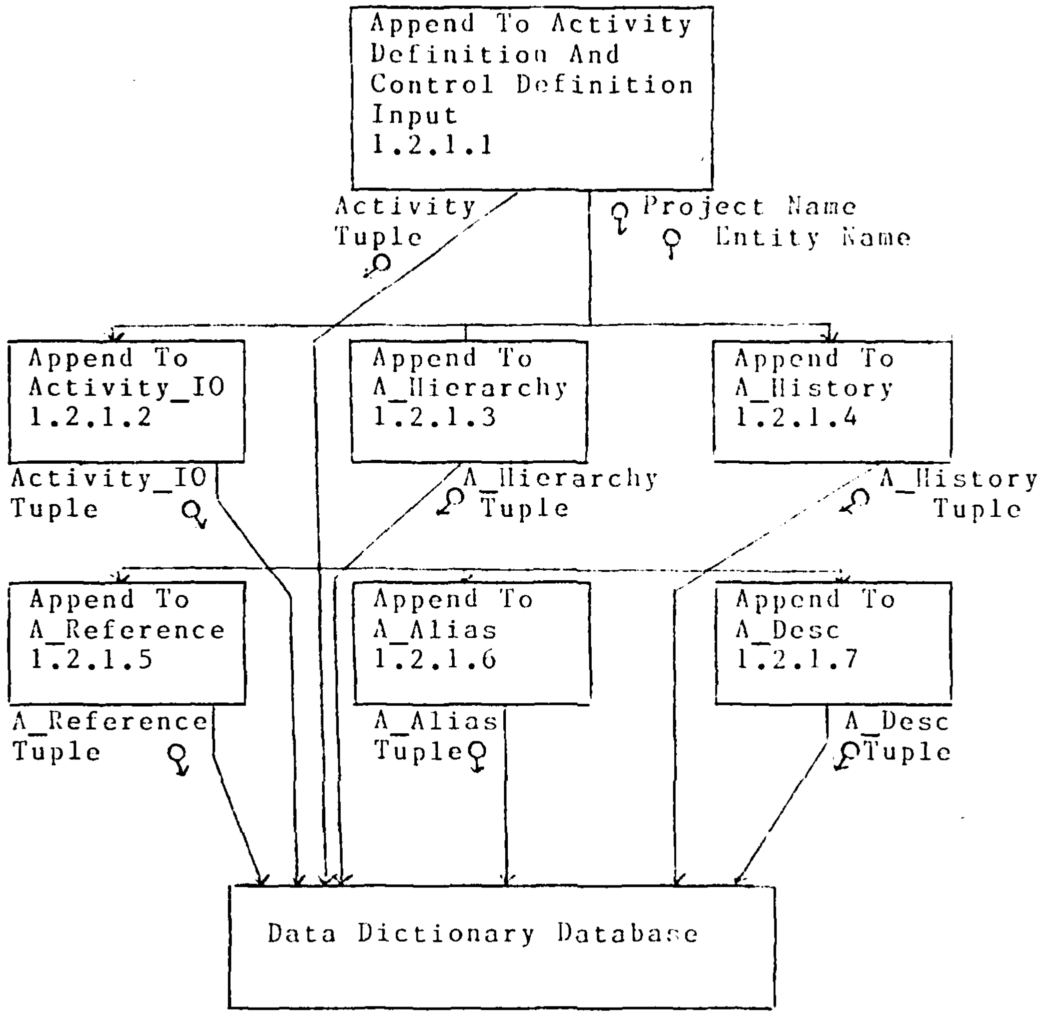
Figure 66. Input Entity Definition (Structure Chart).

The modules displayed in figure 66 would be called by the managerial modules displayed in figure 65 . The functional modules represented by each of the modules displayed in figure 66 would obtain entity information from the tool user, format this information for addition to the dictionary database, and add this information to the appropriate database relation in the dictionary database.

Figure 67 displays the functional modules which represent the logical decomposition of module Input SADT Activity Definition initially displayed in figure 66. The modules displayed in figure 67 query the tool user to obtain information for a particular database relation which contains a portion of the definition for a SADT activity. Once this information is obtained, the individual modules will format this information in a manner which is acceptable to the database and append a new tuple to the subject relation.

In Data Dictionary Database section, the individual relations which support the action and data entity definitions of the various software representation were described and discussed. These modules perform the actual function of placing information in these relation to support the definition of the subject entity.

The Obtain User Input to Append To Activity Relation and Control Input Of SADT Activity Definition module serves the dual role of actually adding information to a dictionary database relation and acting as the control module for the input of a SADT activity definition. From this module, the other functional modules are called which will add



Append To Activity Definition And Control Definition Input 1.2.1.1
Activity Tuple
Project Name
Entity Name
Append To Activity_IO 1.2.1.2
Append To A_Hierarchy 1.2.1.3
Append To A_History 1.2.1.4
Activity_IO Tuple
A_Hierarchy Tuple
A_History Tuple
Append To A_Reference 1.2.1.5
Append To A_Alias 1.2.1.6
Append To A_Desc 1.2.1.7
A_Reference Tuple
A_Alias Tuple
A_Desc Tuple
Data Dictionary Database

Figure 67. Input SADT Activity Definition (Structure Chart).

information to other relations in the database. This module also obtains from the user two important pieces of information which are passed to the other modules, project name and entity name. When the relations were discussed in Data Dictionary Database Section, two attributes were always included in every relation in the dictionary database. The project name and the name of the entity being described. These two attributes were also a component member of the primary key for each relation discussed.

Each module collects the required information from the user to complete the attribute values of the particular relation it supports. When this is complete the modules interact with the database a append a new tuple to the appropriate relation in the dictionary database.

The previous figures and discussion have displayed a portion of the structural model of the Data Dictionary Generation Tool from the abstract high level of the model down to the actual low levl functional modules for one particular operation for one particular representation and entity type. The remainder of the structural model is contained in Appendix B.

This concludes the preliminary design of the Data Dictionary Generation Tool. In the following chapter the detail design phase of the project is discussed.

IV. Detailed Design

Introduction

The detailed design stage of the software life cycle deals with the development of algorithms or procedures for performing the functions or tasks assigned to each module specified during the preliminary design. The modules specified during preliminary design represent the functions the system must perform in order to satisfy the objectives and requirements identified during the requirements definition stage of development. The main effort during the detailed design stage of development is the formulation of precise algorithms which will actually accomplish the designated tasks or functions of the modules identified during preliminary design. The objective of this chapter is to present and describe the algorithms associated with the functional modules of the Data Dictionary Generation Tool.

The algorithms for the Data Dictionary Generation Tool are expresssed in Structured English (21:48-49). In Structured English, English language phrases are used to represent the control constructs required to express functional algorithms.

Two other software engineering options that can be used to specify algorithms are Decision Tables and Decision

Trees (21:49). Neither of these two methods are very applicable to the development of the algorithms for the Data Dictionary Generation Tool. Because of its use of English phrases, the Structured English method is flexible and easy to learn and use. For these reasons, Structured English is the most appropriate method to use in expressing the algorithms for the Data Dictionary Generation Tool.

As stated earlier, the entity definition constitutes, from the user's point of view, the basic informational unit of the data dictionary. The definition of an entity is maintained in the dictionary database by several different relations. Each of these relations maintain a subset of the information required to formulate an entity definition. The individual functional modules of the Data Dictionary Generation Tool are designed to perform a single dictionary operation (delete, addition, modification, print, modify, retrieve) on a single database relation. Several of these individual relations constitute the complete dictionary definition for either an action or data entity for one of the four software representations (SADT, Data Flow Diagrams, Structure Charts or Code) supported by this tool.

A great deal of commonality exists between the functional modules which perform the same dictionary operation. For example, the functional module which adds information to the Activity relation in support of an SADT action entity will utilize basically the same general algorithm as the functional module which adds information to the Variable relation in support of a code data entity definition. This commonality of algorithms exists irregardless of the type of entity or software representation involved.

For the above stated reasons, the algorithms for the functional modules which make up the Data Dictionary Generation Tool will be discussed in terms of the seven major categories of functional modules which correspond to the seven basic data dictionary operations. These seven categories are: selection of the dictionary operation, input of an entity definition, retrieval of an entity definition, modification of an entity definition, listing of entity names, printing of entity definitions, and deletion of entity definitions. Each of these seven categories of modules and their corresponding algorithms will be discussed in the following sections. The first category to be discussed will be the selection of dictionary operation modules.

Algorithms for The Selection of Dictionary Operation Modules

The Selection of Dictionary Operation Modules determine from the tool users the exact data dictionary operation they desire to perform and on what portion of the dictionary database this operation is to take place. These modules then call or select the appropriate functional modules to perform this operation. Three pieces of information are need by these modules in order to correctly identify the proper functional module to meet the tool users needs, the type of operation (addition, deletions, list, print, modify,or retrieve), the entity type of interest (action or data), and the software representation being used (SADT, Data Flow Diagram, Structure Charts, Code). When these three pieces of information are obtained from the user, the Selection of Dictionary Operations modules can select the appropriate functional module to accomplish the users desired operation. The Selection of Dictionary Operations modules consist of the modules 1.1.1, 1.1.2, and 1.1.3 in the structural model of the Data Dictionary Design Tool specified during the preliminary design. The algorithm for these three modules are presented below.

Selection of DIctionary Operation Algorithm

```
(Module 1.1.1 Determine Operation From User)

     DISPLAY Menu of Dictionary Operations
     GET User Operation Selection
     CALL Module 1.1.2 (Operation)

(Module 1.1.2 Determine Entity Type From User)

     DISPLAY Menu of Entity Types(Data or Action)
     GET User Entity Type Selection
     CAll Module 1.1.3 (Operation, Entity Type)

(Module 1.1.3 Determine Software Representation And Call
 Functional Module)
```

```
DISPLAY Menu of Software Representations Supported
        (SADT, DFD, Structure Charts, Code)
GET User Representation Selection

IF   Operation=Input  and  Entity  Type=Action  and
     Representation=SADT THEN
         CALL Input SADT Action Entity Definition
IF   Operation=Delete  and  Entity  Type=Data  and
     Representation=Code THEN
         CALL Delete Code Data Entity Definition
```

The IF THEN statement for this module continue until every possible combination of entity type, dictionary operation, and software representation have been tested or the correct combination is found and the functional module which controls the accomplishment of the user's desired operation is called.

This completes the discussion of the algorithms which accomplish the Selction of DIctionary Operations modules function. The next category of modules and algorithms to be discussed are the Add Entity Definition modules.

Add Entity Definition

The Add Entity Definition modules query the tool user for the necessary information to formulate the definition of an entity. These modules then format this information in a manner which is acceptable to the database relations and append the information to the proper relation in the database. There are eight sets of entity definition input module within the Data Dictionary Generation Tool. One set of modules for both the action and data entities of the four software representations supported. Each set of functional modules consists of the individual modules responsible for appending new information to each relation which maintains information for the definition of a particular entity type and representation and a control module which executes the calls to these modules based on user inputs.

In actually, the control modules performs a dual role. It not only controls the input of the entity definition, but also appends a database relation. The presentation of the algorithm for a control module will serve to present not only the algorithm used to control the input of an entity definition but also to display the algorithm need to add information to an individual database relation. To present these algorithms, the control module for inputting an SADT action entity defintion will be displayed. This module is the Append To Activity Relation and Control Definition Input module (1.2.1.1) specified in the preliminary design structural model. The algorithm is displayed below:

```
PROMPT User For Project Name
GET Project Name
PROMPT User For SADT Activity Name
GET Activity Name
PROMPT User For Activity Number
GET Activity Number
INPUT NEW TUPLE IN Activity Relation
  Project Attribute=Project Name
  Name Attribute=Activity Name
  Number Attribute=Activity Number
CALL   Append  To  Activity_IO  Relation(Project  Name,
       Activity Name)
CALL Append To A_Desc Relation(Project  Name,  Activity
       Name)
```

```
CALL   Append To A_Hierarchy Relation(Project Name,
       Activity)
PROMPT User, Do Any References To Previous Development
       Stages Exist For This Activity?
GET User Response
IF Response=Yes THEN
   CALL  Append To A_Reference Relations(Project Name,
         Activity)
PROMPT User, Do Any Alias Names Exist For This
       Activity?
GET User Response
IF Response=Yes THEN
   CALL  Append To A_Alias Relation(Project Name,

           Activity Name)
CALL   Append To A_History Relation(Project Name,
       Activity Name)
```

This algorithm displays both the control of a defintion input operation and the actual addition of information to a database relation. Not all relations which support an entity definition will be called during an actual input operation. As the algorithm displays, in some cases the information maintained in a relation may not exist for the definition of a particular entity. This concludes the discussion of the algorithms which support the Input An Entity Definition modules. The next category of functional modules to be discussed will be the Retrieve Entity Definition modules.

Retrieve Entity Definition Algorithms

The Retrieve Entity Definition functional modules obtain from the tool user the necessary information to identify the particular entity definition the user wishes to retrieve. These modules then extract the relevant information from the database relations which contain the entity defintion and diplay them to the tool user. As in the Input Definition modules discussed in the previous section, there are eight sets of Retrieve Definition modules contained in the Data Dictionary Generation Tool. A particular set of these modules will retrieve either an action or data entity definition for a particular software representation. The individual modules which perform the definition retrieval operation for a particular entity type and representation will interact with one of the dictionary database relations which maintains a portion of the entity definition. One of the modules in each of the eight sets of Retrieve Definition modules will, in addition to retrieving information from a database relation, also act as the control module for the entire definition retrieval.

An examination of a Definition Retrieval Control module will display not only the algorithm for controling definition retrievel, but also the algorithm for retrieving information from a database relation. The following is the algorithm for the Retrieve From Module Relation and Control Definition Retrievel module which is module number 1.3.7.1 in the structural model developed during the preliminary design stage. This module controls the retrieval of a action entity definition from the code software representation.

Retrieve From Module Relation and Control Definition Retrieval (1.3.7.1)

```
PROMPT User For Project Name
GET Project Name
PROMPT User For The Name Of The Entity To Be Retrieved
GET Entity Name
WRITE TO TERMINAL Code Module Defintion
WRITE TO TERMINAL Entity Name
WRITE TO TERMINAL Project Name
Retrieve From Relation Module the Value Contained In
 Number Attribute for The Relation Tuple Where
 Project Attribute = Project Name And Name Attribute =
 Entity Name.
WRITE TO TERMINAL Entity Number
CALL Retrieve From M_Desc Relation
CALL Retrieve From Module_IO Relation
CALL Retrieve From M_Alias Relation
Call Retrieve From M_Call Relation
CALL Retrieve From M_Pass Relation
CALL Retrieve From M_Reference Relation
CALL Retrieve From M_Hierarchy
CALL Retrieve From M_History Relation
CALL Retrieve From M_Alg Relation
```

When attempting to extract information from a relation, sufficient information must be provided to identify the particular tuple or tuples in the relation where the desired information can be located. This requirement is fulfilled in the above algorithm by using the project and entity names as qualifiers in the retrieval from the Module Relation. The control portion of the algorithm simply consists of a sequential call to every relation in the database which might contain a portion of the entity definition.

This concludes the discussion of the algorithms for the Retrieve Entity Definition functional modules. The next category of algorithms to be discussed will be those for the Print Entity Definition modules.

Print Entity Definition Algorithms

The Print Entity Definition module write the entity definition for all action or data entities associated with a particular software project to file. The tool user can then utilize the procedures associated with the particular operating system supporting the Data Dictionary Generation Tool to obtain a printed copy of the entity definitions. There are, as in the previous categories of functional modules discussed, eight sets of print modules associated with the Data Dictionary Generation Tool. Each of these sets of functional modules support the printing of the data or action entities for one of the four software representations supported by the dictionary.

The algorithm for writing the entity definitions to a system file are similar to the algorithms used to retrieve entity definitions from the database. The only difference, is that instead of writing the entity definition to the terminal screen, these functional modules write the definition to a system file. The Print Entity Definition modules obtain the name of the project of interest from the tool user. These modules then, using project name as a search key, obtain from the dictionary database the name of every entity associated with this project which is of the entity type and software representation supported by this group of functional print modules. These entity names are found by searching the main relation for each combination of entity type and representation. These main relations are those relation which serve as the storage relation for identification of entity definitions supported. The eight main relations are displayed in figure 68 and their corresponding entity type and software representation are also presented in figure 68.

The entity names obtained from the search of the main relations are written to a system file as they are retrieved When the retrieval of entity names is complete, the modules open the file where these entity names are stored and one at a time send the entity name to the functional modules which retrieve the entity definition from the database relations and write the definition to a system file. This process continues until all entity definition have been retrieved and written to the system file.

Main Relation Name	Entity Type	Software Representation
Activity	Action	SADT
Data Item	Data	SADT
Bubble	Action	Data Flow Diagram
Data Flow	Data	Data Flow Diagram
Process	Action	Structure Chart
Parameter	Data	Structure Chart
Module	Action	Code
Variable	Data	Code

Figure 68. Main Relation For Printing Entity Definitions

The following two algorithms will display one of the control module for The Print Entity Definition Functional modules and one of the actual functional modules involved in the operation. The algorithms displayed are for the Control Print Of SC Parameter Definition (1.6.6.1) , which controls the printing of structure chart parameter definitions, and for the Print P_Alias Relation module (1.6.6.5) which prints to the system file any alias names associated with the parameters whose definition are being printed.

Control Print Of SC Parameter Definition Module 1.6.6.1

```
PROMPT User For Project Name
GET Project Name
OPEN File A For Writing
Retrieve   The  entity  name  for  all  tuples  in  the
Parameter  Relation  Whose Project  Attribute  Value  =
  Project Name
Write entity names to File A
OPEN File A For Reading
WHILE NOT End Of File On File A
   READ Next Entity Name In File A
   CALL Print Parameter Relation(Project, Entity)
   CALL Print P_Desc Relation(Project, Entity)
   CALL Print P_Hierarchy Relation(Project, Entity)
   CALL Print P_Reference Relation(Project, Entity)
   CAll Print Process_IO Relation(Project, Entity)
   CALL Print Pr_Passed Relation(Project, Entity)
   CALL Print P_Alias Relation(Project,Entity)
   CALL Print P_Value_Set Relation(Project,Entity)
   CALL Print P_History Relation(Project, Entity)
```

Print Alias Relation Module 1.6.6.5

(Values for Project and Entity Name are passed to this module from the control functional module displayed above.)

Retrieve From P_Alias Relation The value for the Name1

```
    Attribute  Where The Project Attribute = Project Name
    And the Name2 Attribute = Entity Name
  WRITE TO FILE B Alias Names:
  WRITE TO File B Name1 attribute value
```

This concludes the discussion of the algorithms for the Print Entity Definition Functional modules. The next category of algorithms to be discussed are those associated with the Delete Entity Definition Functional Modules.

Delete Entity Definition Algorithms

The Delete Entity Definition module remove an entity definition from the dictionary database. There are eight different sets of Delete Definition functional modules contained in the Data Dictionary Generation Tool. Each set of modules handles the delete definition function for a data or action entity for each of the four software representations supported by the dictionary. The individual functional modules will delete the appropriate tuples in an individual relation in the database. As in the previous categories of modules, one module will act as the control module for the definition deletion. A control module and a functional module will be used to display the algorithms associated with the deletion of an entity definition from the dictionary database.

The modules described in the following algorithms are the Control SC Process Definition Deletion module (1.7.5.1), which control the deletion of a action entity definition in the structure chart representation, and the

Delete From Process Relation module (1.7.5.2) which deletes the tuple from the Process Relation which supports the definition of a particular entity definition. The algorithms for these two modules are displayed below :

```
PROMPT User For Project Name
GET Project Name
PROMPT User For Entity Name
GET Entity Name
CALL Delete From Process Relation(Project, Entity)
CALL Delete From Process_IO Relation(Project, Entity)
CALL Delete From Pr_Call Relation(Project, Entity)
CALL Delete From Pr_Passed Relation(Project,Entity)
CALL Delete From Pr_Hierarchy Relation(Project, Entity)
CALL Delete From Pr_Alias Relation (Project, Entity)
CALL Delete From Pr_Reference Relation(Project, Entity)
CALL Delete From Pr_Desc Relation(Project, Entity)
CALL Delete From Pr_Alg Relation(Project, Entity)
CALL Delete From Pr_History Relation(Project, Entity)
```

<u>Delete From Process Relation (1.7.5.2)0</u>

(Values for Project Name and Entity Name are passed from the Control module displayed above.)

```
DELETE From the Process Relation The Tuple
WHERE Project Attribute = Project Name and Name
Attribute = Entity Name
```

This concludes the discussion of the algorithms which support the deletion of entity definitions from the dictionary database. The next section discusses the algorithms associated with the modifiy entity definition cataegory of functional modules.

Modify Entity Definition Algorithms

The Modify Entity Definition Functional modules allow the tool user to modify or change the contents of an entity definition which is maintained in the dictionary database. There are eight sets of modify definition functional modules contained in the Data Dictionary Generation Tool. Each set of functional modules supports either the action or data entites of one of the four representations supported by the dictionary.

The modification operation can take three different forms: addition of new information to the entity definition, deletion of a portion of the information for an entity definition, and the changing of information in the definition from one value to another. For example, the tool user might wish to add to the definition of an action entity the name of another action entity which the action entity defined calls or uses. A long the same line the user might want to delete information about a file which an action entity uses. The user may wish to change the minimum value which a parameter defined in the dictionary can assume.

As in the other categories of modules discussed, the modification modules for a particular entity type and representation are managed or controlled by a module which calls the appropriate functional module. In the case of the Modify Entity Definition modules, these control modules will present the user with a menu selection of the components of the entity definition. The user will select the information component that he wishes to modify. Once the user's selection is obtained the control module will call the appropriate functional module which will perform the modification operation on the particular database relation which maintain the portion of the entity definition which the user desires to modify.

The exact nature of the algorithm will depend upon the particular information element the user wishes to modify and how that particular portion of the entity definition is represented in the database relations. The functional modules will have to obtain additional information from the tool user in order to ascertain exactly what type of modification the user wishes to perform (delete, add, change)

The following algorithm is for the Modify P_Value Set Relation (1.4.6.4). This module allows the user to modify ie, delete, add , or change, a value which a parameter can assume in the definiton of a structure chart data entity It is presented here because it provides an example of the use of all three possible modification operations. The values for the project name and the entity name are passed to this module from the control module for modification of structure chart parameter definitions.

```
DISPLAY Menu of Modification Operations
   1. Add new value parameter can assume
   2. Delete a value which the parameter can assume
   3. Change the value
GET User Response
IF Response = 1 THEN
     CALL Append To P_Value Set Relation
IF Response = 2 THEN
     PROMPT User For The Value To Be Deleted
     GET Value To Be Deleted
     DELETE Tuple From Relation P_Value Set
     WHERE Project Attribute =Project Name AND
     Name Attribute = Entity Name AND Value Attribute
     = Value To Be Deleted
IF Response = 3
     PROMPT User For Value To Be Changed
     GET Value To Be Changed
     PROMPT User For New Value
     GET New Value
     REPLACE The Value Attribute In Relation
      P_Value_Set New Value In The Tuple
      WHERE Project Attribute = Project Name
      AND Name Attribute = Entity Name AND
      Value Attribute = Value To Be Changed
```

This concludes the discussion of the algorithms which are used in the Modify Entity Definition Functional modules. The next category of algorithms to be discussed are those associated with the List Entity Names Functional Modules.

List Entity Names Algorithms

The List Entity Names functional modules present to the tool users a list of all entity names associated with a particular project within a specific entity type and software representation. For example, these functional modules allow the user to view all the structure chart parameter definitions associated with a particular project designation. There are eight different functional modules which perform the list entity names function for either a data or action entity associated with one of the four software representation supported by the data dictionary.

In the Print Defintion section, the use of main relations to retrieve all entity names for the print functions was discussed. The main relations for each of the data and entity definitions for each of the four software representations were displayed in figure 68. The list operations use these same main relations to obtain the name of all entities which are defined for a particular entity type, software representation, and project designation. The only basic difference between the algorithm for obtaining entity names in the Print Definition section and here is that instead of writing the names to a file, as was done in the Print Definition section ,the entity names are written directly to the terminal screen.

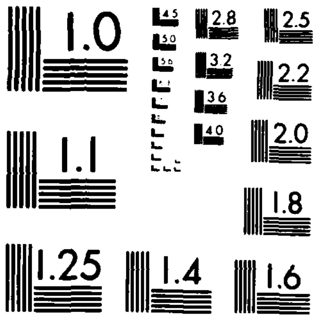
The following is the algorithm for List Names For SC Parameters (1.5.6). Which will print to the terminal screen the names of all parameter defined under a specific project name.

```
PROMPT User For Project Name
GET Project Name
RETRIEVE From Relation Parameter The Name Attribute For
All Tuples WHERE The Project Attribute = Project Name
WRITE TO TERMINAL Name Attribute
```

This concludes the discussion of the List Entity Names Functional modules and their associated algorithms. This also concludes the discussion of the basic algorithms used

AD-A152 215 AN AUTOMATED/INTERACTIVE SOFTWARE ENGINEERING TOOL TO GENERATE DATA DICTIONARIES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. C W THOMAS 3/4

UNCLASSIFIED DEC 82 AFIT/GCS/ENG/84D-29 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

by the functional modules which make up the Data Dictionary Generation Tool.

V. Implementation

Introduction

The implementation stage of the software life cycle involves the conversion of the detailed design specification into an appropriate programming language. The specific objectives of the implementation stage for the Data Dictionary Generation Tool is to fully code and test the functional modules developed during the preliminary and detailed design stages.

In the following sections, the selection of the database management system (DBMS) to support the dictionary database will be discussed. An appropriate programming language for coding the functional modules of the Data Dictionary Generation Tool will also be selected and discussed. The implementation of the dictionary database design and the implementation of the Data Dictionary Generation Tool's functional modules will be presented. This chapter will conclude with a discussion of the use of the DBMS to perform dictionary maintenance and administrative functions and a discussion of the testing of the Data Dictionary Generation Tool.

Selection of Database Management System

The operation of the Data Dictionary Generation Tool is dependent upon the existence of a database management system

(DBMS) to maintain the dictionary relations designed and discussed in the Data Dictionary Database section. The DBMS selected must also provide support to the Tool's application software in interacting with the dictionary database relations to delete, modify, input, and retrieve data dictionary information.

During the initial development of this project, the target environment for the tool was as a component tool for the Software Development Workbench (SDW) which resides on the VAX 11/780 computer utilizing the VMS operating system. This system is primarily used to support research and upper level graduate courses. As the initial development continued, it was realized that the Data Dictionary Generation Tool could be developed as an independent tool to support graduate classes in Software Engineering and Real Time Programming Laboratories. For this reason, the tool was initially developed on the VAX 11/780 computer utilizing the UNIX operating system. This system supports the majority of the graduate students and graduate courses at the Air Force Institute of Technology. When this project began, the VAX/VMS supported a network database system known as TOTAL. However, the acquistion of an INGRES Relational Database system was planned and subsequently completed for the VAX/VMS system. The VAX/UNIX system already supported an INGRES DBMS. Therefore, the implementation plan for the Data Dictionary Generation Tool was to develope the tool as an independent tool on the VAX/UNIX system and eventually transfer the tool to the VAX/VMS system where it would be a component part of the Software Development Workbench (SDW).

Because the INGRES DBMS existed on the VAX/UNIX system and was in the process of being acquired for the VAX/VMS system, the INGRES Relational DBMS was selected to support the Data Dictionary Generation Tool. In order to maintain compatability between the two versions of the tool residing on both the VAX/VMS and VAX/UNIX systems, it was essential that the same DBMS be used by both versions of the Data Dictionary Generation Tool.

Although the selection of INGRES as the DBMS for the tool was driven mainly by the need to maintain compatabilty between the two versions of the tool on both target system, INGRES is an excellent DBMS which fulfilled all the tool's requirements for DBMS support.

INGRES is a relational database mangement system which meets all the requirements as a DBMS for the dictionary tool. "INGRES is designed for both powerful functions and ease of use. As such, INGRES offers users a range of useful commands in QUEL (QUEry Language), the system's data manipulation language. A database is a shared resource containing information about some subject. As a data source accessed by many users, a database must be managed so that the different users' needs are met. The database must also be managed so that the computer system is used efficiently as possible. Balancing these two management tasks is the job of a DBMS, and INGRS is designed to handle the two tasks without compromising the utility of the data " (24:1-1).

The Data Dicitonary Generation Tool has three general requirements which the INGRES DBMS must support:

1. Maintain the Dictionary Database Relations.
2. Support the interaction of application software the database relations in the retrieval, input, deletion, and modification of data dictionary information.
3. Provide a capability for performing dictionary maintenance and administrative functions (I.E create new relations, modify format of relation, etc.).

The INGRES DBMS is a relational DBMS. As such, it is able to support the dictionary relations designed in Data Dictionary Database section in the format required by the Data Dictionary Generation Tool. INGRES also possesses facilities which support the designation of relation attributes to serve as primary keys for the subject relation. The actual maintenance of relations in an INGRES DBMS will be discussed more in the Dictionary Implementation section when the implementation of the dictionary database is discussed.

INGRES, through QUEL (que.y language), provides a means by which the information contained in the database relations can be accessed by an INGRES user. The QUEL commands allow information to be added to relations and retrieved from relations. QUEL also provides a means of modification of attribute values within particular tuples in a relation and for the deletion of tuples from a relation. The application software of the Data Dictionary Generation Tool allows the user to view information in the format of an entity definition rather than as a series of related but separate database relations. In order to interact with the database relations, the application software utilizes EQUEL (Embedded Query Language) commands which the INGRES DBMS supports.

"EQUEL (Embedded QUEL) is an embedding of the INGRES query language into a procedural programming language (25:1-1)". EQUEL is provided as the programming language interface to INGRES because it offers significant advantages for the programmers. It is nearly identical to QUEL, the INGRES query language. Equel allows the programmer to utilize the control constructs of a programming language for looping and condition checking while interacting with database. "EQUEL is essentially the same in all languages, statements used in different languages are interchangeable (25:1-2)". The use of EQUEL statements in the application software of the Data Dictionary Generation Tool will be further discussed in Data Dictionary Generation Tool Implementation section when the actual coding of the Data Dictionary Generation Tool's functional modules are discussed.

The INGRES DBMS provides facilities which can be used to perform the maintenance and administrative functions required to support the dictionary database. These functions allow the creation and deletion of relations in the database. INGRES also provides facilities for database security and the selection of the most appropriate storage structures for database relations. The dictionary maintenance and administrative functions will be discussed further in the Dictionary Database Implementation section and the Use of DBMS To Perform Dictionary Maintenance and Administrative Functions section.

Choice Of Implementation Language

The programming language selected for the Data Dictionary Generation Tool must be compatable with EQUEL, the embedded query language supported by INGRES. The language selected must also be available on the two target machines, VAX/VMS and VAX/UNIX. In order to support communication between the user and the tool, the language must provide input and output facilities for data as well as other information handling facilities. The language must also have facilities for conditional branching and support modular design.

As in the selection of the support DBMS, compatability between the two target systems is the reason for the particular language selected for the implementation of the Data Dictionary Generation Tool. The VAX/UNIX system supports the embedding of EQUEL statements in only the C programming language. The VAX/VMS system supports the embedding of EQUEL statements in several programming languages: Pascal, Fortran, C, Cobol, and Basic(25:1-2). In order to maintain compatability between the version of the tool on both the VAX/UNIX and VAX/VMS systems, the C programming language was selected as the implementation language for the Data Dictionary Generation Tool.

Although compatability was the main reason for the selection of the C language, it does adequately meet all the requirements of the tool for an implementation language.

"C is a general purpose programming language which features economy of expression, modern control flows and data structures, and a rich set of operators" (27:5). " C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie.... C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C" (27:5). The C language does have some disadvantages over other languages such as Pascal and Fortran. C itself provides no input or output facilities and no wired in file access methods. All of these higher level mechanisms are provided by explicitly called functions. These functions, however, are provided by a standard C I/O library which is supported on all machines which support C. This standard library allows C programs which require input, output, and other system functions to be moved from one system to another essentially without change (27:4). The actual use of the C language in coding the functional modules of the Data Dictionary Generation Tool are discussed further in the Implementation of The Tool's Functional Modules section when the actual implementation of the application software for the Data Dictionary Generation Tool is discussed.

Implementation of Dictionary Database

The implementation of the dictionary database involves the installation of the database relations discussed in the Data Dictionary Database section in the INGRES DBMS. The initial step in installing the dictionary database is to create an INGRES database to support the data dictionary. This is accomplished be the execution of the INGRES createdb command. The createdb command creates a new database under the INGRES DBMS (26:1). The name given to the database which supports the Data Dictionary Generation Tool is swtools. Therefore the following command creates the database:

```
     $ createdb swtools     ($ operating system prompt)
```

The database is initially created without containing any relations or data. The next step is to create within the swtools database the individual relations which will maintain the data dictionary information.

The relation or table is the basic storage unit in INGRES (24:6-1). It is easy to create relations, as long as the following three pieces of information are known: attributes or columns of the relation, the type of data that will be placed under each attribute, and the amount of space or the allowable size of each attribute.

INGRES supports three types of data: character string, integer, and floating point (24:6-2). The character string data type is appropriate for non-numeric data such as names, dates, addresses,etc. The integer data type is appropriate for numeric data that have no decimal points, integers. The floating point data type is appropriate for numeric data with decimal points, real numbers.

In addition to the data type of each attribute, the size of each attribute must be designated when a relation is installed into an INGRES supported database. Size and data types are designated by the use of a character immediately followed by a number. Characters are designated by the character c. A numeric value following this letter designates the size of a charcter type attribute. For example, c12 indicates an attribute which contains characters whose maximum size is 12 characters.

For integers, the number following the letter i (designation for the integer data type) indicates the byte size supported by the relation. The byte size determines the range of numbers which can be stored under the subject attribute. For example, the designation i1 indicates a byte size of one which allows the attribute to accomodate any integer number greater than -128 and less than +128. The range for an i2 designation (2 byte size) would be any integer number between -32,768 and +32,768.

"Floating point numbers can be specified as either single precision (4 bytes) or double precision (8 bytes). Both types designations support a range of from -10 ** 38 to +10 ** 38 (** indicates exponentiation). The precision choosen effects how many decimal places are retained in the number"(24:6-2). "Single precision (f4) supports seven decimal digit precision and double precision (f8) supports 17 decimal digit precision" (24:6-3).

The actual implementation of a relation in an INGRES supported database is done by the execution of the create command. The create command creates tables or relations in a database. In order to use the create command the subject database must be accessed from the INGRES DBMS. This is accomplish by entering the following command from the operating system:

```
   $ ingres swtools     ($ operating system prompt)
```

This commands allows the swtools database to by accessed through the INGRES DBMS.

The syntax for the INGRES create command is as follows.

```
*  create relation name(domain name 1=format, domain name
   2=format)
```

(* INGRES DBMS prompt)

Relation name is the name of the relation being created. Domain name is the name of the individual relations which make up the relation being created. Format is the designation which indicates the data type and size associated with each attribute. For example, the following command:

```
* create activity(project=c12, name=c25, number=c15)
  (*INGRES DBMS prompt)
```

would create the activity relation described in the Data Dictionary Database section. Figure 69 presents a visual display of the relation created.

Activity Relation

project 12 characters max	Name 25 character max	Number 15 characters max

Figure 69. Create Relation Example

The INGRES create command is used to implement all the dictionary relations described and discussed in the Data Dictionary Database section. There are three other procedures which are applied to newly created relations: designation of storage structure and primary keys, setting permissions on relations, and designating and time limitation on a relations existence in the database. Since these procedures are useful at times other than the implementation of new relation, they will be discussed in the section on the Use of the DBMS To Perform Dictionary Maintenance and Adminstration Functions.

Implementation of Data Dictionary Generation Tool's Functional Modules

The actual implementation of the application software associated with the Data Dictionary Generation Tool consists of the generation of C language code which contains embedded EQUEL statements. The C program statements perform the necessary communications tasks between the user and the tool while the embedded EQUEL statements handle the necessary interactions with the dictionary database. The program consisting of EQUEL statements embedded in C programming language statements is processed through the EQUEL/C preprocessor. "Statements beginning with two number signs (##) are recognized by the EQUEL/C preprocessor. All other statements must be standard C or statements acceptable to another preprocessor.... The EQUEL extension allow table names, column names, target-list elements, domain values and qualifier clauses to be contained in C variables" (25:1-1).

Each line in the source code which contains an EQUEL statement must begin with two number signs (##) in the first column. In order to transfer information between the dictionary database and the tool user, it is necessary for C language variables to be referenced to the relation names, and attributes names. In order for this to occur, the subject C variables must be made known to the EQUEL/C preprocessor. This is accomplished by beginning the line on which the C variables are declared with the ## signal.

In order for the application software to interact with the swtools database, which contains the data dictionary information, it must initiate access with the database. This is accomplished by including the following statement in the source code: ##ingres swtools. In order to end database access from the application software, the following statement must be included in the source code: ##exit.

In the following sections, examples of the C code with embedded EQUEL statements which perform the four major functions of the tool (input defintion, retrieve definition, modify defintion, and delete definition) will be presented and discussed.

Input Definition.

The algorithm for inputting an entity defintion was discussed in the Detailed Design chapter. The functional modules that perform this task prompt the tool user for data dictionary information and append this information to the proper relation in the dictionary database. These modules utilize calls to the standard C I/O library to accomplish the input and output of information between the tool and the user. The standard C library function "printf" is used to display formatted output to the terminal screen. The standard C library funcition "fgets" is used to obtain user input. The EQUEL command "append" is used to add a tuple to the appropriate relation in the database. The following is an example of the source code required to input the information for an entity definition. This example only demonstrates the addition of information to a single relation. However the same basic source code is used in all functional modules involved in the input of an entity definition. The text displayed between two * symbols represents an explanation or comment and is not part of the actual source code.

```
Inputactivity(proj,actname)
*  module  name and input variables proj(project  name)  and
   actname (entity name) *
##char proj[15], actname[27];
*    C  declaration  of input variables,  ##  indicates  that
     variables are known to the EQUEL/C preprocessor and that
     variables can be used in EQUEL statements*

{* Begin symool in C*

##char  actno[15];  *  Declaration of a  local  variable  in
     module which is known to EQUEL/C preprocessor*

printf("±nEnter activity number.±n"); *User Prompt*
fgets(actno,14,stdin);  *  Input  of  user response  into  C
  variable actno*
##append to activity(project=proj,name=actname,number=actno)
*  EQUEL  Statement  which add  information  to  a  database
   relation*
}*End symbol in C*
```

The EQUEL append command adds a tuple to the activity relation. It also obtains values for the attributes in the tuple (project, name, number) from the referenced C variables(proj, actname, actno).

Retrieve Definition

The definition retrieval algorithm is presented in the Detailed Design Chapter. The modules which perform the retrieval function use the tool user's imput as a key or guide in searching the database for the requested information. This guide or search information forms the qualifier for an EQUEL retrieve statement which finds and retrieves information from the database. The following is a sample of source code required to retrieve information from a single relation in the database

```
getactivity(proj,actname)
##char proj[15], actname[27];
{
##char actno[15];
##range of e is activity
* Establishes the relation of interest, allows variable e to
  represent the activity relation in the retrieve statement*
##retrieve(actno=e.number) *Retrieves the value of the
  number attribute and sets the C variable actno equal to
  this attribute value*
##where e.project=proj and e.name=actname
*  Qualifier for the retrieve statement *
##{
  printf("±nACTIVITY NUMBER:±n",actno);
##}
{
```

The EQUEL range statement allows the e variable to represent the activity relation. The retrieve statement obtains the value of the number attribute from the database relation and places the value in the C variable actno. The where statement is the qualifier for the retrieve statement. It identifies the proper tuple of the relation from which the retrieve statement is to retrieve the required attribute. In example, the desired tuple is the one where the project attribute is equel to the value in the C variable proj and the name attribute is equal to the value in the C variable actname. The ## and ## symbols provide a useful convention. This statement causes the C code in between the two symbols to be executed once for each tuple retrieved. For example, if two tuples in the activity relation qualified for the retrieval(two tuples which both had the same project and name attribute values) the C printf function would display number attribute value associated with both tuples.

<u>Delete Definition</u>

The algorithm for the Delete Entity Definition functional modules is presented in the Detailed Design Chapter. These modules, based upon user input, delete data dictionary information from relations in the database. Like the EQUEL retrieve stat ment, the delete statement use a where statement as a qualifier for selecting the proper tuple(s) to be deleted from a relation. The range statement is also used to set a variable to represent the relation of interest. The following is an example of the code used to delete information from a single relation in the database.

```
deleteactivity(proj,actname)
##char proj[15],actname[27];
{
##range of e is activity
##delete e where e.project=proj and e.name=actname
```

This code will delete every tuple in the activity relation where the project attribute and the name attribute are equal to the value for the C variable proj and actname respectively.

Modify Definition

The algorithm for the functional modulules which modify an entity definition are presented in the Detailed Design Chapter. The EQUEL replace statement is used to change the value of an attribute in a particular tuple of a relation. This statement uses the range statement to set a variable to the relation of interest and a where statement to identify the proper tuple in the relations for modification. The following is an example of source code used to change a value of an information element which makes up an entity defintion.

```
modifyactivity(proj,actname)
##char proj[15],actname[27];
{
##char actno[15];
printf("±nEnter new number for activity.±n");
fgets(actno,15,stdin);
##range of e is activity
##replace e(number=actno)where e.project = proj and e.name =
actname
}
```

The EQUEL replace statement places the new value for the number attribute in the relation tuple where the project and name attributes are equal to the corresponding values for the proj and actname C variables.

Use of The DBMS to Perform Dictionary Maintenance and Administrative Functions

The INGRES DBMS system provides facilities which can be used to perform the maintenance and administrative function for the Data Dictionary Generation Tool. These functions consist of the creation of new database relations, the identification of storage structures and primary attribute keys for database relations, the deletion of relation from the database, setting of access permisssions for relations, and setting time limitation on the existence of relations in the database.

The creation of database relation was discussed in the Dictionary Database Implementation section. The other dictionary maintenance functions will be discussed in the following sections.

Selecting Storage Structure and Identifying Primary Keys For the Dictionary Database Relations

When a relation is created in an INGRES supported database, it is automatically stored as a heap. A heap storage structure has two main characteristics: nothing is known about the location of the tuples in the relation and duplicate rows are not removed from the relation. New tuples are added at the bottom of the relation regardless of what attribute values are contained. In order for INGRES to perform a query on a heap storage structure it must scan the entire relation to be sure of retrieving the correct data (24:17-1).

INGRES supports two other storage structures (other than heap) that locate tuples without having to scan the entire relation. These two structures, hash and ISAM can greatly accelerate queries run on relations which contain a large number of rows (24:17-3).

The hash storage structure stores each tuple in a relation at an address determined by the value of a attribute or combination of attributes contained in the tuple. These attributes form the key for the relation. These keys allow the access to a specific tuple in a relation to be speeded up. The hash structure is especially useful in queries which involve the exact matching of key attribute values (24:17-4).

The ISAM storage structure is useful for retrieval based upon a range of values. ISAM is a structure that supports retrieval based upon both an exact match and and ranges of values. "ISAM stands for indexed sequential access method (23:-17-4)

As stated earlier, a newly created relation is always stored as a heap. The INGRES modify command is used to change the storage structure associated with a particular relation. The following examples demonstrate the use of the modidfy command to change a heap structure to both a hash and ISAM storage structure.

The command modify employee to hash on name would change the storage structure of relation employee to hash and cause the attribute name to be used as a key to the relation. In a similar fashion, the command : modify employee to isam on salary would cause the storage structure of relation employee to be converted to ISAM and the salary attribute to be used as a key for the relation.

Setting Time Limitations on the Existence of Relations in the Database

The QUEL command save is used to preserve a given relation in the database until a given expiration date. Only the owner of the relation (the person who created the relation can save a relation (28:2-30). For example, the following command will save relation activity until February 28, 1987:

```
save activity until feb 28 1987.
```

Deletion of Relations From The Database

The QUEL destroy command is used to remove a relation from the database. Only the owner of a relation (the person who created the relation can destroy it). This command is very different from the delete command used in the application software for the tool. The delete command only removes information from a relation. The destroy command totally removes the relation from the database. If the relation activity was removed removed from the database by means of the destroy command, an attempt to retrieve from or append to the activity relation would cause the DBMS to issue an error message. The following is an example of the destroy command being used to remove a relation named department: destroy department.

Obtaining Information About The Structure Of a Database Relation

The QUEL help command is used to obtain information about the structure of any relation in the database. By entering the command: help relation name, a list of the all the attribute fields which are contained in a relation, the storage structure supporting the relation, and the key attributes of a relation can be found. The help command also provides useage information by displaying the number of tuples currently stored in a relation. The help command also displays the type of information contained in each attribute in a relation and the allowable size of a value under that attribute.

Testing

Testing the functional modules of the Data Dictionary Generation Tool was greatly enhanced by the use of the INGRES DBMS. Since the EQUEL commands embedded in the source code are identical to the QUEL commands used in INGRES, the result of EQUEL statements could be tested without having to actually run the application program. Each functional module of the Data Dictionary Generation Tool was designed to interact with only one individual database relation. This modular design made it much easier to trace errors which occurred during the testing of the tool.

The implementation testing of the Data Dictionary Generation Tool was conducted in four phases. During phase one, the INGRES QUEL commands were used to test the EQUEL statements embedded in the source code. This ensured that the interactions with the database would obtain the desired result during the execution of the dictionary operations.

The next phase was the testing of the individual functional modules. Since each functional module was responsible for both obtaining the necessary information from the user and using this information to perform a specific operation on an individual database relation, these modules formed a natural building block for the system.

Phase three of implementation testing involved the testing of the functional and mangerial modules involved in performing operations on the basic informational unit of the dictionary, the entity definition.

Phase four consisted of placing all functional and lower level managerial modules under the executive modules for the tool and testing the system as a whole.

During each of these phase, the IGRES DBMS was used to validate that the desired operations on the database were performed as a result of the execution to the functional modules of the Tool.

VI. Conclusions And Recommendations

Introduction

The purpose of this thesis investigation was to design and implement an automated interactive software engineering tool which would generate data dictionaries from the information contained in software representations such as SADTs, Data Flow Diagrams, Structure Charts, and Code. This tool was to provide its users with an interactive data dictionary to support the development of software throught all phases of the software life cylce.

Design Summary

The software life cycle approach was utilized in developing the Data Dictionary Generation Tool. The tool's requirement's definition phase was directed toward identifying the goals and objectives of the system. A list of objectives and concerns for the tool was developed and discussed. A model describing and defining system requirements was also developed.

The preliminary design phase of development identified four software representations which would be supported in the initial version of the Data Dictionary Generation Tool: SADTs, structure charts, data flow diagrams, and code. The information required to formulate the data dictionaries for each of these representations was identified and the database to maintain this information was designed. The structural framework of the application software required to perform the basic dictionary operations (addition of information, deletion of information, information modification, etc) was developed.

During the detailed design phase of development a functional algorithm was formulated to support each functional and managerial module identified in the structural model developed during the preliminary design phase.

The implementation phase consisted of the selection of both a DBMS, INGRES, to support the dictionary database and an implementation language, C, to code the application software. The database to support the four software representation was implemented and a portion of the application software was coded and tested.

Implementation/Testing Results

The Data Dictionary Generation Tool was utilized by the EE 690, Real Time Programming Laboratory, class to generate data dictionaries in support of the class project. Approximately 24 students utilized the tool and provided feedback on the tool's performance. The EE 690 project supported by the tool utilized only the portion of the tool which supported the structure chart and code software representation. The application software for the Data Dictionary Generation Tool was improved greatly as a result of feedback from the students who used the tool for this project as shown in the following paragraph.

During the initial of use of the tool, several errors were detected in the execution of the application software. These errors, for the most part, were due to the failure of the system to sufficiently check user inputs for errors. Another problem encounter was system responsiveness. The user would sometimes experience delays while waiting for system prompts to signal the user to enter information into the dictionary database. As a result of the user's initial experience with the tool, it was realized that the user required a small amount of hands on experience with the tool before they were comfortable with its operation.

As a result of feedback obtained, the application software for the tool was modified to include additonal error checking routines to verify user inputs. The Tool was being supported by the VAX 11/780 computer utilizing the UNIX operating system. This system was experiencing an extremely heavy workload during time of the tool's initial use. The tool was checked during times of normal useage on the VAX/UNIX system and its response time was found to be adequate for interactive use. However, during peak workload periods for the system, the tool's response time was slow. The response time problem was due to the workload on the host system and was not due to problems in the Data Dictionary Generation Tool.

With the inclusion of error checking procedures in the application software and an increased level of user experience with the tool, the problems experienced by the EE690 class in using the tool were greatly reduced and the support provided by the tool was improved.

Recommendation For Further Development

This effort resulted in the tool only being implemented on the VAX/UNIX system. The tool should be rehosted on the VAX/VMS system and be integrated unto the Software Development Work Bench (SDW).

This tool constitutes an excellent first step towards the development of a tool which will automatically generate data dictionaries from the actual graphical representations or diagrams used in the SADT, Structure Chart, and Data Flow Diagram methods of software representation. The interfacing of this tool with the graphical tools supported hosted on the Software Development Workbench would constitute an excellent start in this development effort.

During the design of the database for the tool, several areas of commonality among the relations supporting the various representations was pointed out. Howver, this commonality was not taken advantage of in either the design of the database nor the development of the application software. Investigation into these areas could lead to improvement in the tool's speed of operation and reduction in the memory space required to support the tool's operation.

Appendix A: Requirement's Model Data Dictionary Generation Tool

The Data Dictionary Generation Tool is a software engineering tool which generates provides data dictionary support for various methods of software representation such as structure charts, data flow diagrams, etc. In order to develope this tool, a thorough understanding of the goals and objectives of the tool must be formulated. This model, utilizing the data flow diagram method of representation, provides a means of defining and describing the requirements for this tool. The upper level of the model is presented and explained in chapter two, Requirement's Definition.

Requirement's Model Data Dictionary Generation Tool

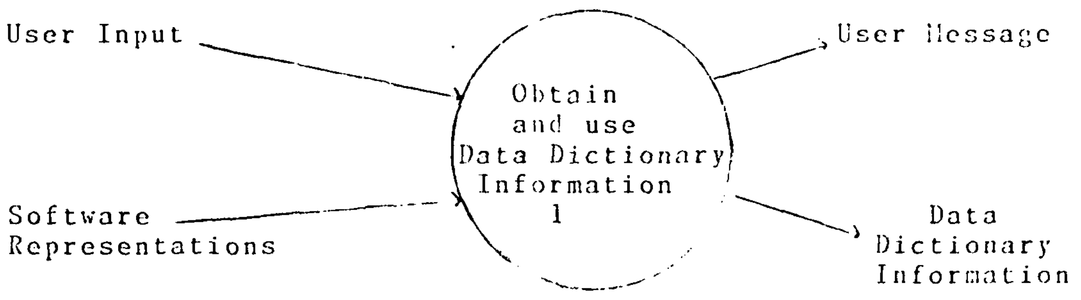

User Input
Software Representations
Obtain and use Data Dictionary Information 1
User Message
Data Dictionary Information

Figure 1-1. Top Level Data Dictionary Generation Tool.

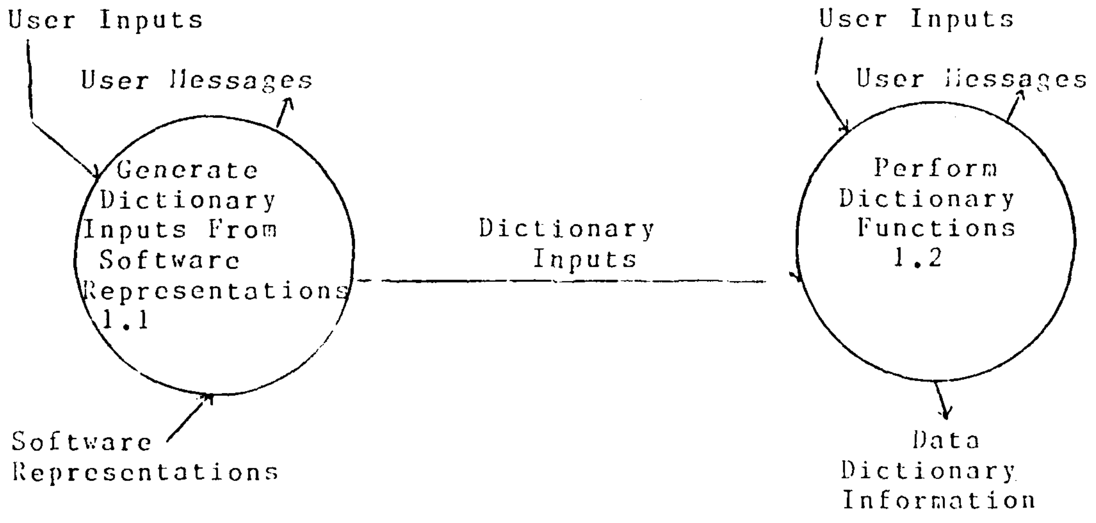
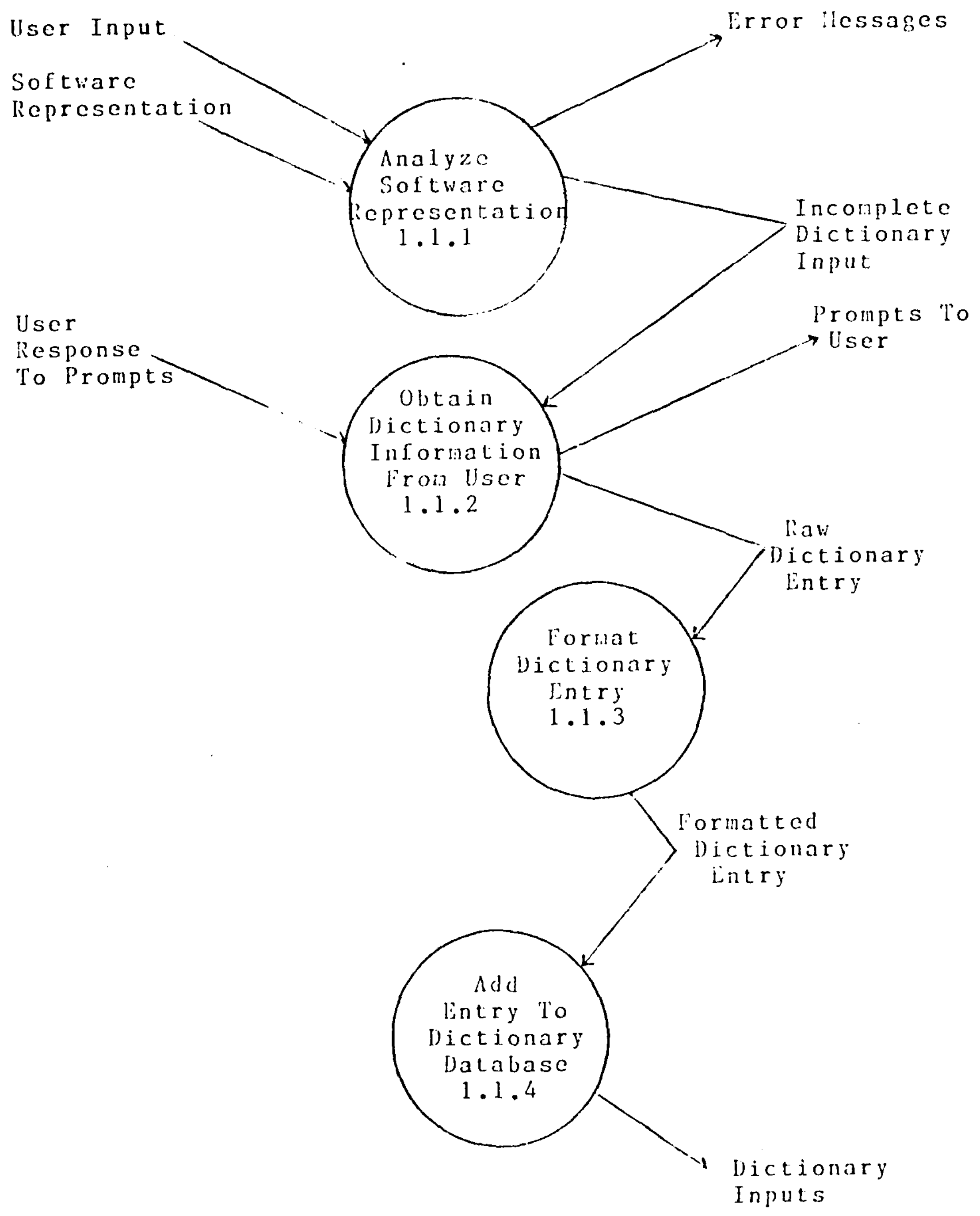

User Inputs
User Messages
Generate Dictionary Inputs From Software Representations 1.1
Software Representations
Dictionary Inputs
User Inputs
User Messages
Perform Dictionary Functions 1.2
Data Dictionary Information

Figure 1-2. Obtain and Use Data Dictionary Information (1).



User Input
Software Representation
Error Messages
Analyze Software Representation 1.1.1
Incomplete Dictionary Input
User Response To Prompts
Prompts To User
Obtain Dictionary Information From User 1.1.2
Raw Dictionary Entry
Format Dictionary Entry 1.1.3
Formatted Dictionary Entry
Add Entry To Dictionary Database 1.1.4
Dictionary Inputs

Figure 1-3. Generate Dictionary Inputs From Software Representations (1.1).

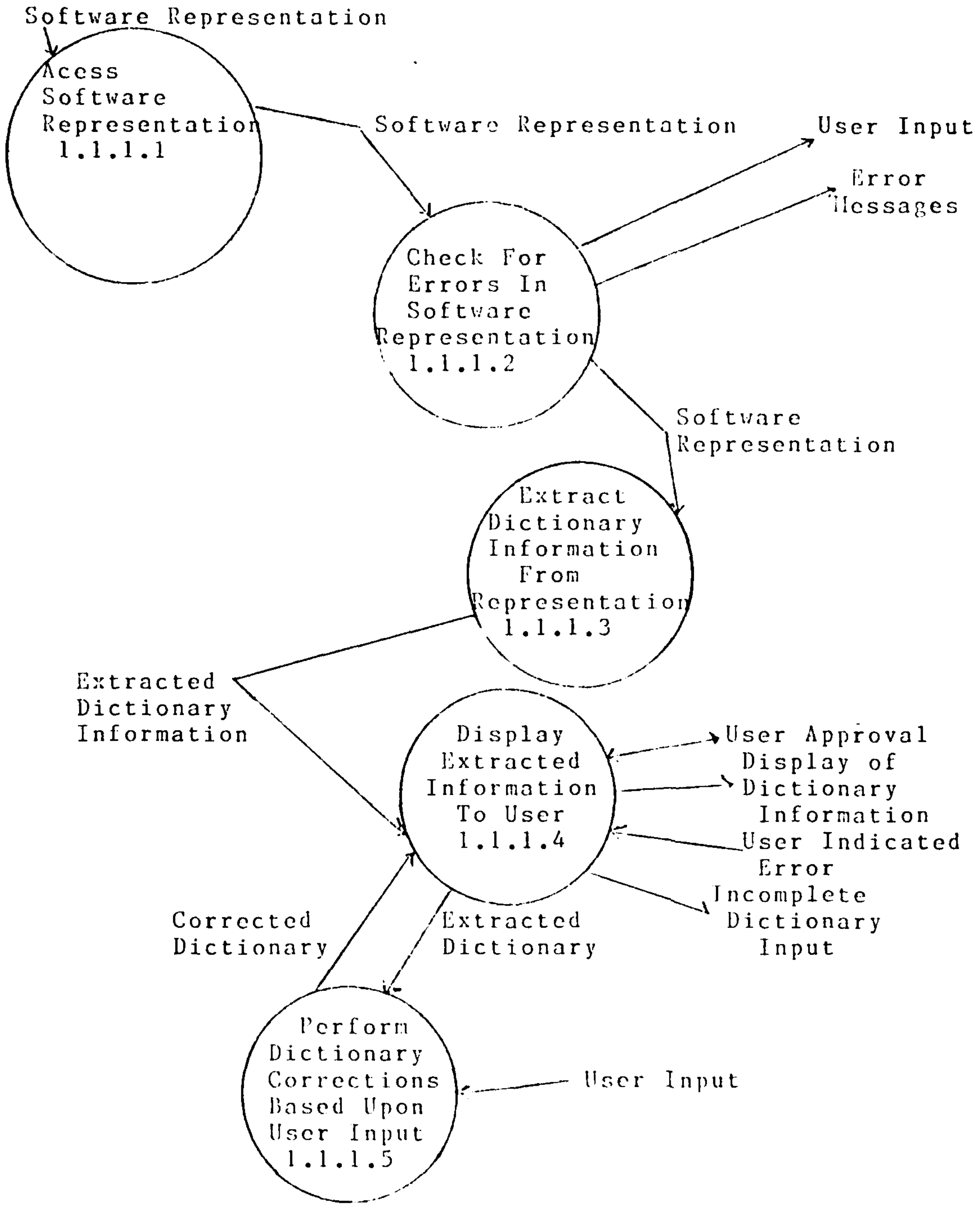

Software Representation
Acess Software Representation 1.1.1.1
Software Representation
User Input
Error Messages
Check For Errors In Software Representation 1.1.1.2
Software Representation
Extract Dictionary Information From Representation 1.1.1.3
Extracted Dictionary Information
Display Extracted Information To User 1.1.1.4
User Approval
Display of Dictionary Information
User Indicated Error
Incomplete Dictionary Input
Corrected Dictionary
Extracted Dictionary
Perform Dictionary Corrections Based Upon User Input 1.1.1.5
User Input

Figure 1-4. Analyze Software Representation (1.1.1).

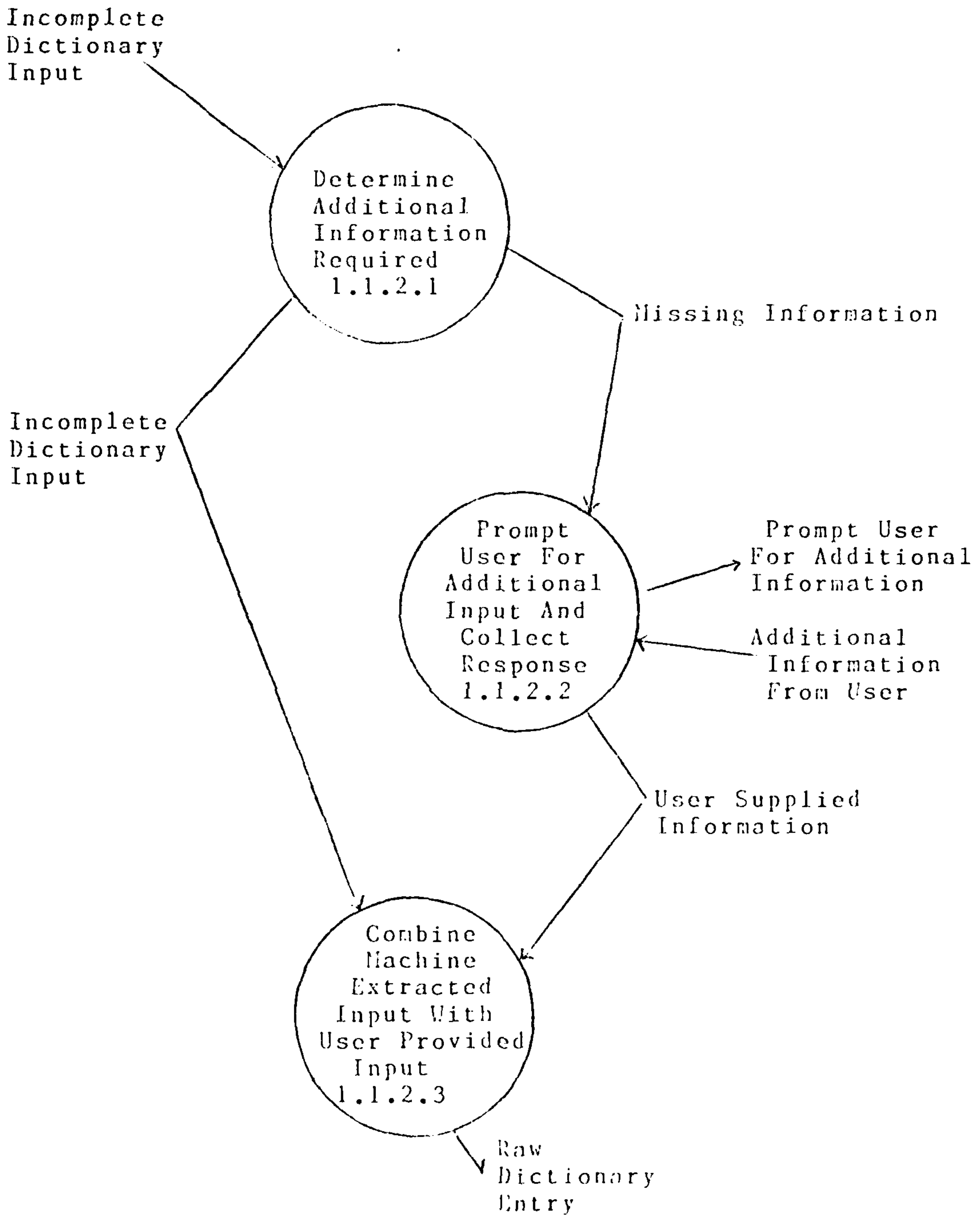

Incomplete Dictionary Input
Determine Additional Information Required 1.1.2.1
Missing Information
Incomplete Dictionary Input
Prompt User For Additional Input And Collect Response 1.1.2.2
Prompt User For Additional Information
Additional Information From User
User Supplied Information
Combine Machine Extracted Input With User Provided Input 1.1.2.3
Raw Dictionary Entry

Figure 1-5. Obtain Additional Dictionary Information From User (1.1.2).

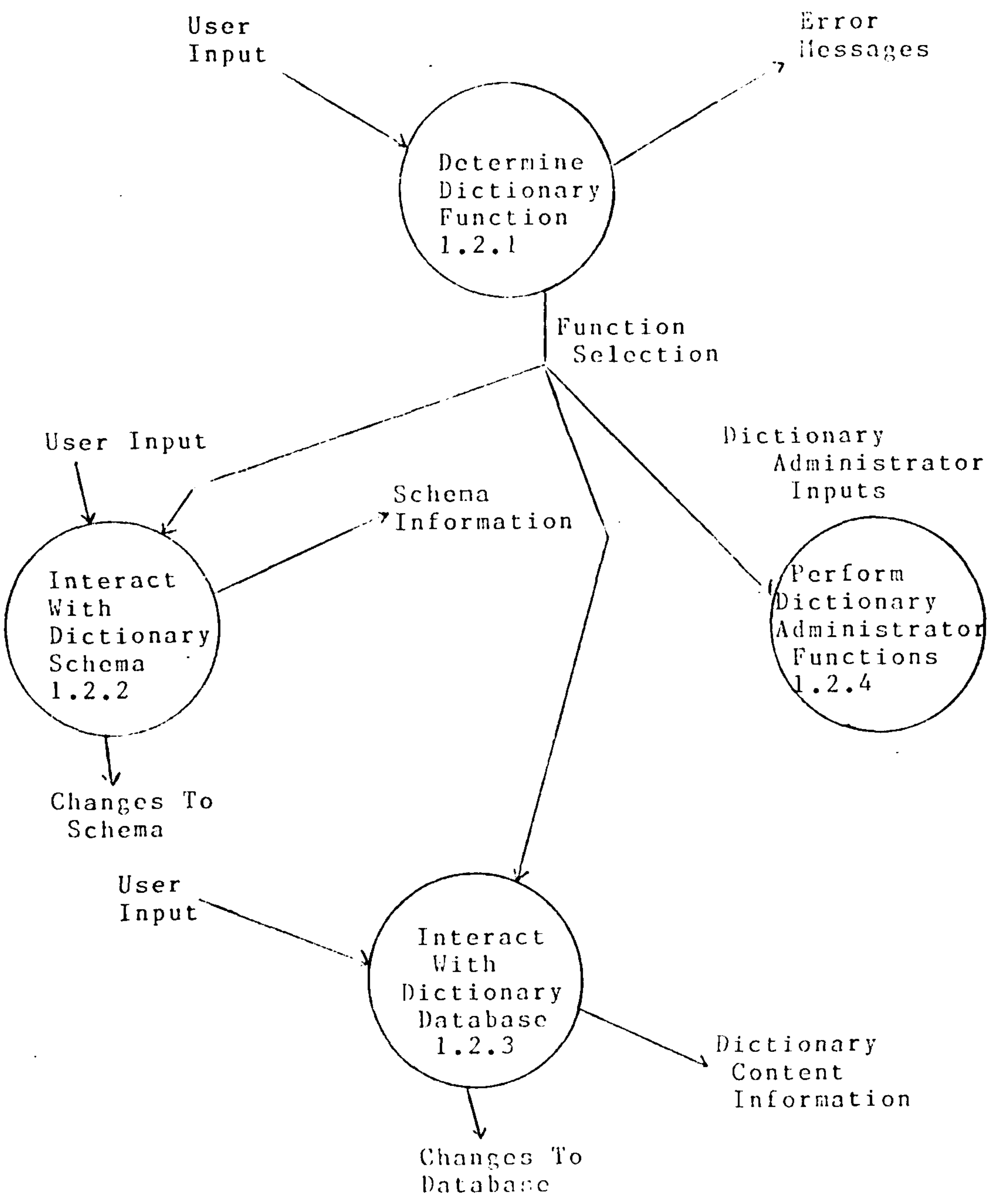

User Input
Error Messages
Determine Dictionary Function 1.2.1
Function Selection
User Input
Schema Information
Dictionary Administrator Inputs
Interact With Dictionary Schema 1.2.2
Perform Dictionary Administrator Functions 1.2.4
Changes To Schema
User Input
Interact With Dictionary Database 1.2.3
Dictionary Content Information
Changes To Database

Figure 1-6. Perform Dictionary Functions (1.2).

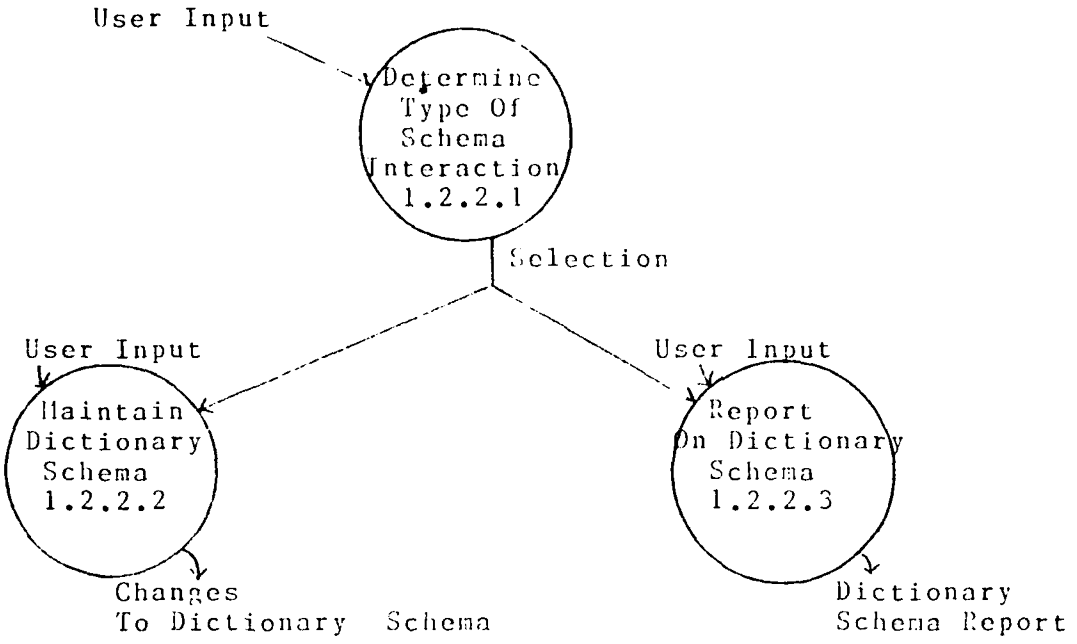

User Input
Determine Type Of Schema Interaction 1.2.2.1
Selection
User Input
Maintain Dictionary Schema 1.2.2.2
User Input
Report On Dictionary Schema 1.2.2.3
Changes To Dictionary Schema
Dictionary Schema Report

Figure 1-7. Interact With Dictionary Schema (1.2.2)

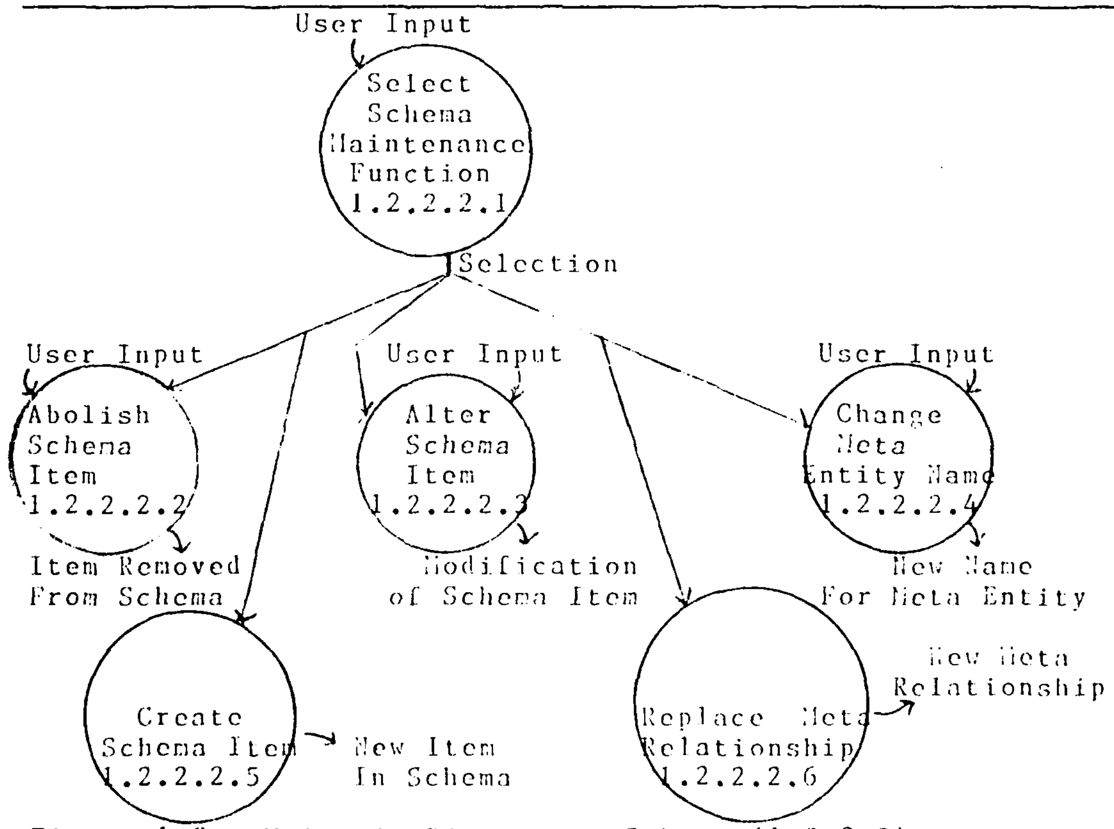

User Input
Select Schema Maintenance Function 1.2.2.2.1
Selection
User Input
Abolish Schema Item 1.2.2.2.2
User Input
Alter Schema Item 1.2.2.2.3
User Input
Change Meta Entity Name 1.2.2.2.4
Item Removed From Schema
Modification of Schema Item
New Name For Meta Entity
Create Schema Item 1.2.2.2.5
New Item In Schema
Replace Meta Relationship 1.2.2.2.6
New Meta Relationship

Figure 1-8. Maintain Dictionary Schema (1.2.2.2).

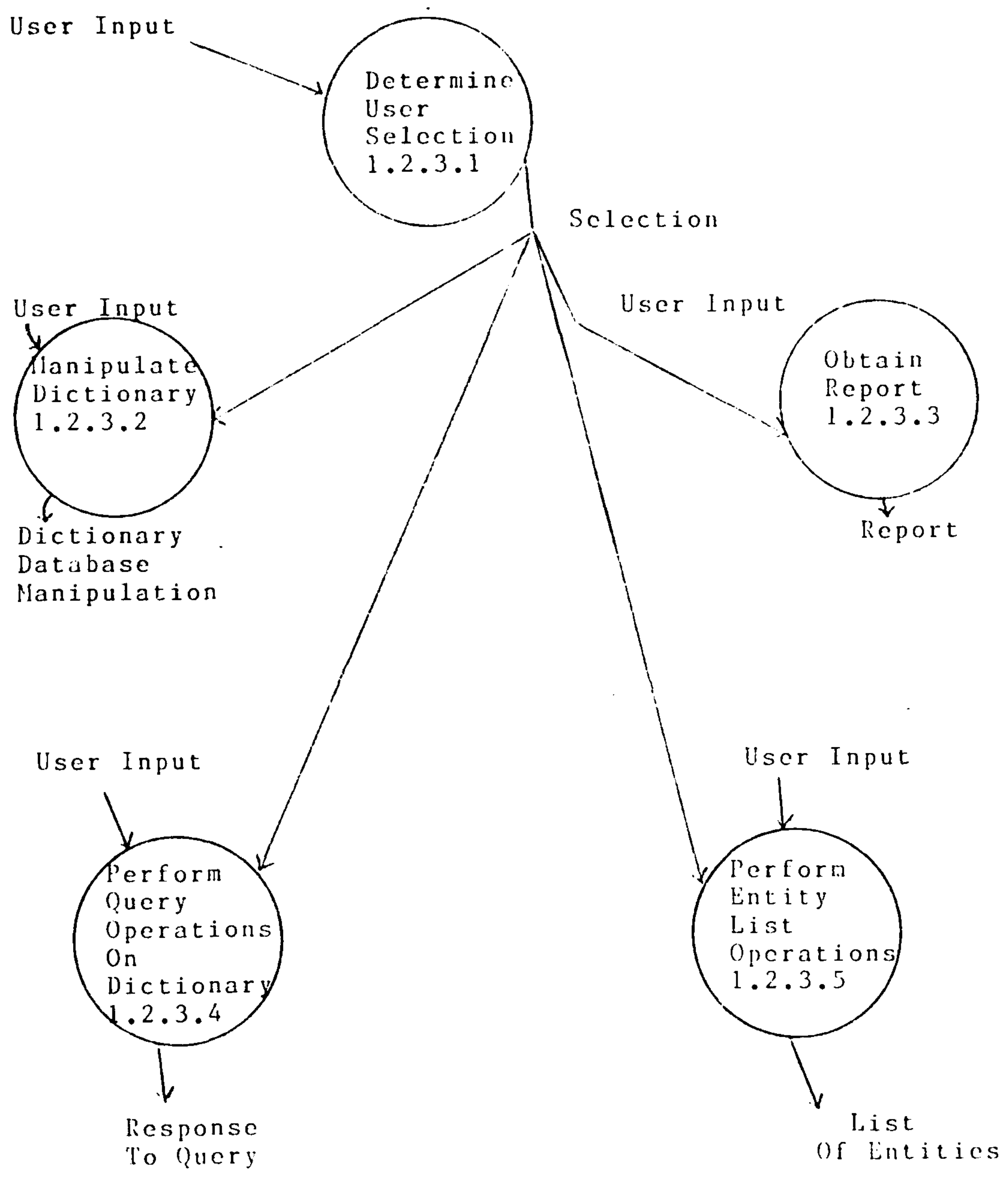

User Input
Determine User Selection 1.2.3.1
Selection
User Input
Manipulate Dictionary 1.2.3.2
Dictionary Database Manipulation
User Input
Obtain Report 1.2.3.3
Report
User Input
Perform Query Operations On Dictionary 1.2.3.4
Response To Query
User Input
Perform Entity List Operations 1.2.3.5
List Of Entities

Figure 1-9. Interact With Dictionary Database (1.2.3)

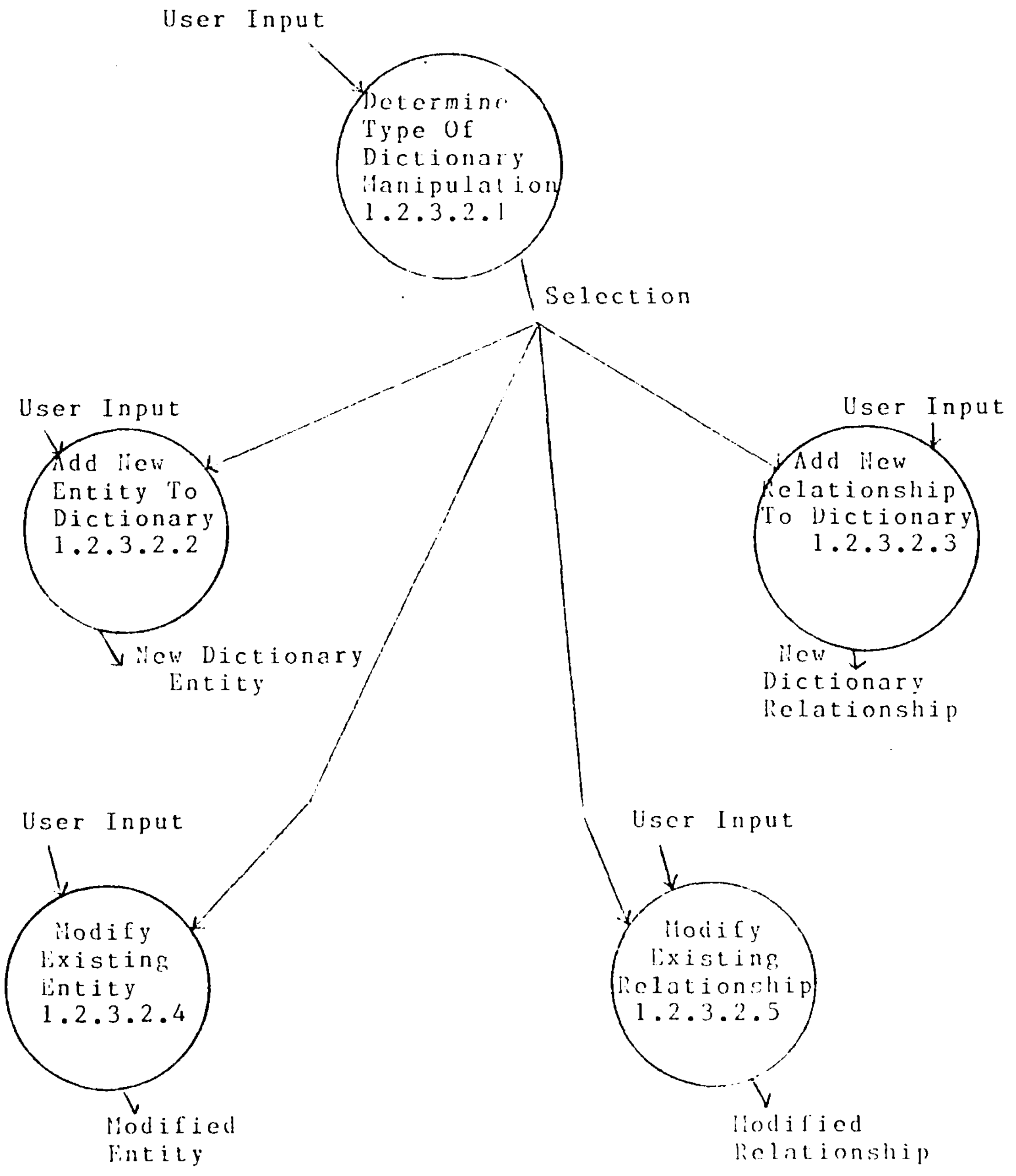
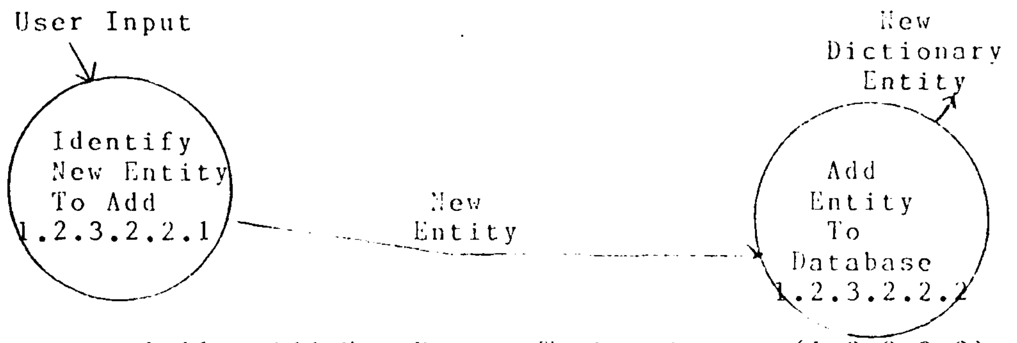
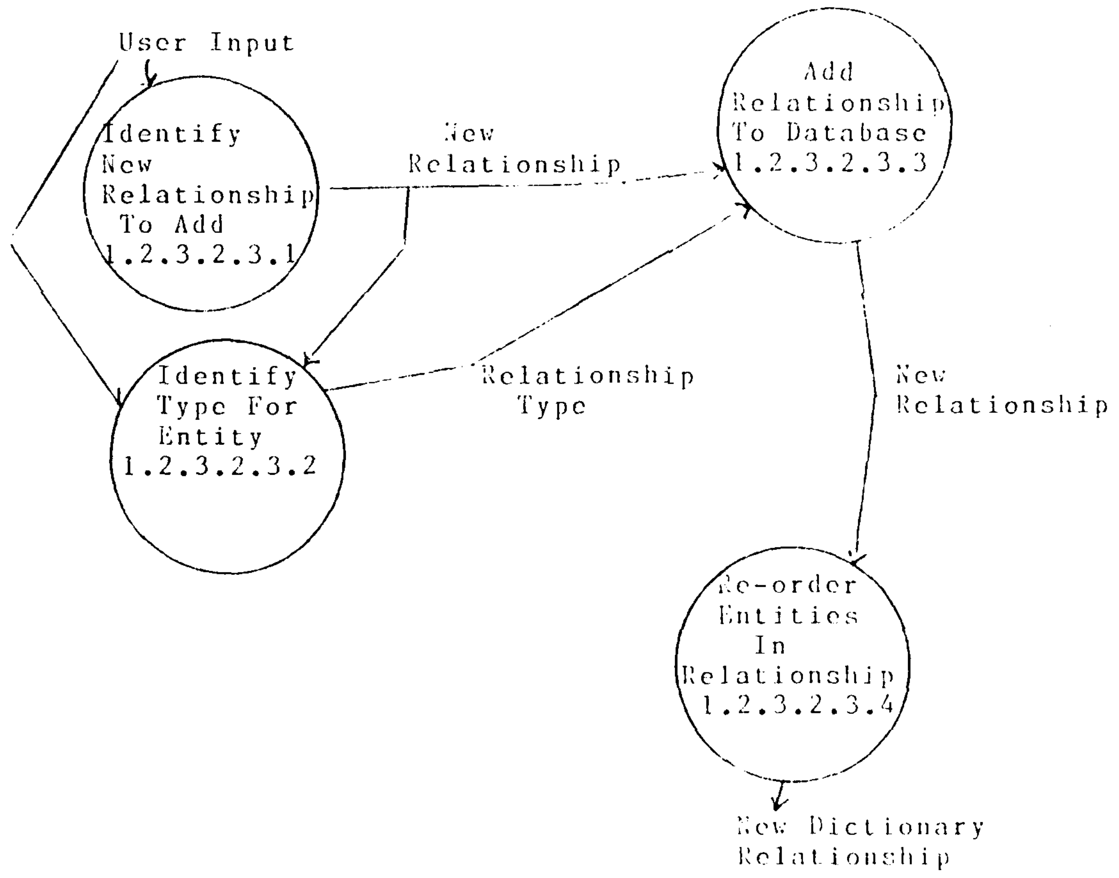

User Input
Determine Type Of Dictionary Manipulation 1.2.3.2.1
Selection
User Input
Add New Entity To Dictionary 1.2.3.2.2
New Dictionary Entity
User Input
Add New Relationship To Dictionary 1.2.3.2.3
New Dictionary Relationship
User Input
Modify Existing Entity 1.2.3.2.4
Modified Entity
User Input
Modify Existing Relationship 1.2.3.2.5
Modified Relationship

Figure 1-10. Manipulate Dictionary (1.2.3.2).



User Input
Identify New Entity To Add 1.2.3.2.2.1
New Entity
Add Entity To Database 1.2.3.2.2.2
New Dictionary Entity

Figure 1-11. Add New Entity To Dictionary (1.2.3.2.2).



User Input
Identify New Relationship To Add 1.2.3.2.3.1
New Relationship
Add Relationship To Database 1.2.3.2.3.3
Identify Type For Entity 1.2.3.2.3.2
Relationship Type
New Relationship
Re-order Entities In Relationship 1.2.3.2.3.4
New Dictionary Relationship

Figure 1-12. Add New Relationship To Dictionary (1.2.3.2.3).

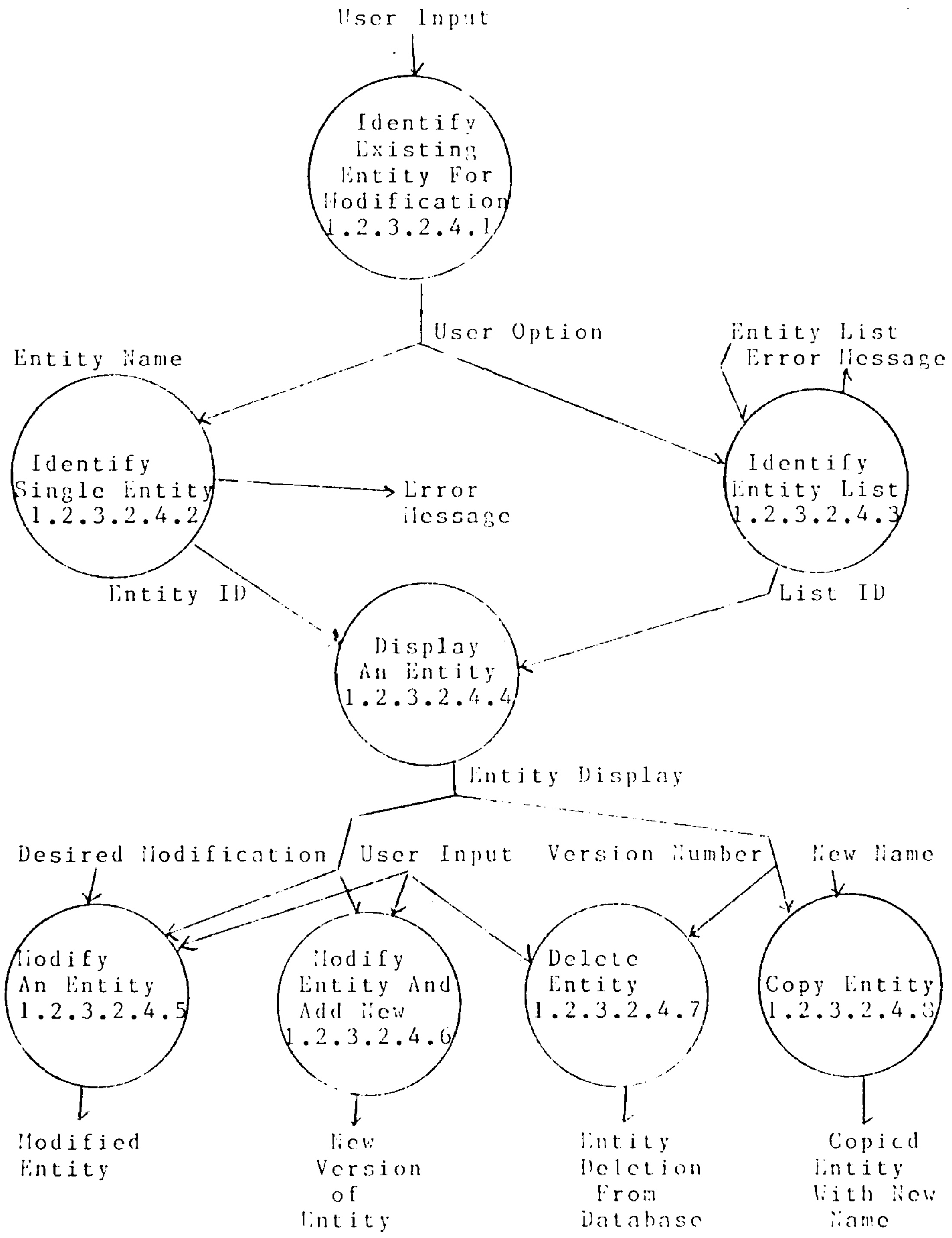

User Input
Identify Existing Entity For Modification 1.2.3.2.4.1
User Option
Entity Name
Entity List Error Message
Identify Single Entity 1.2.3.2.4.2
Error Message
Identify Entity List 1.2.3.2.4.3
Entity ID
List ID
Display An Entity 1.2.3.2.4.4
Entity Display
Desired Modification
User Input
Version Number
New Name
Modify An Entity 1.2.3.2.4.5
Modify Entity And Add New 1.2.3.2.4.6
Delete Entity 1.2.3.2.4.7
Copy Entity 1.2.3.2.4.8
Modified Entity
New Version of Entity
Entity Deletion From Database
Copied Entity With New Name

Figure 1-13. Modify Existing Entity (1.2.3.2.4).

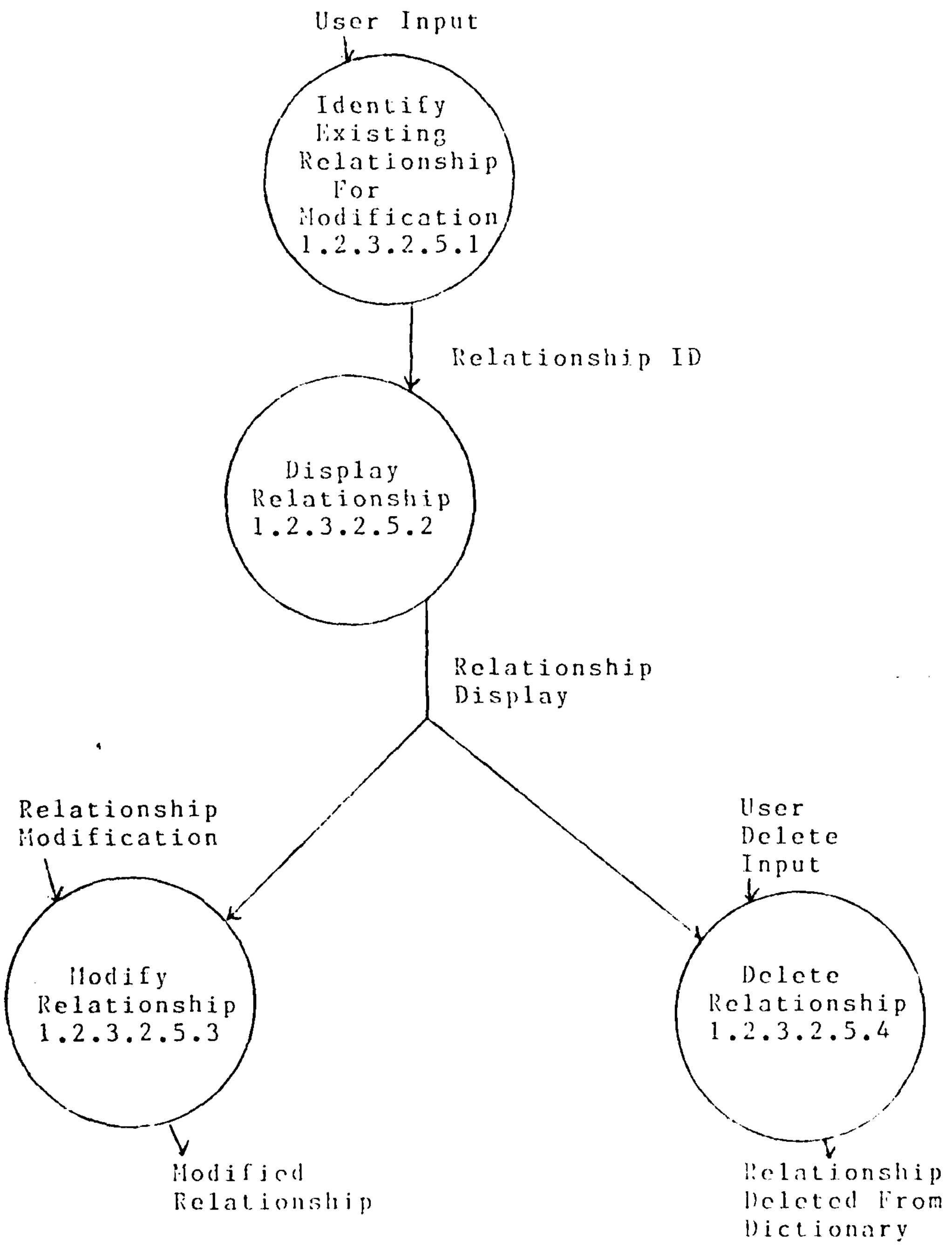

User Input
Identify Existing Relationship For Modification 1.2.3.2.5.1
Relationship ID
Display Relationship 1.2.3.2.5.2
Relationship Display
Relationship Modification
User Delete Input
Modify Relationship 1.2.3.2.5.3
Delete Relationship 1.2.3.2.5.4
Modified Relationship
Relationship Deleted From Dictionary

Figure 1-14. Modify Existing Relationship (1.2.3.2.5).

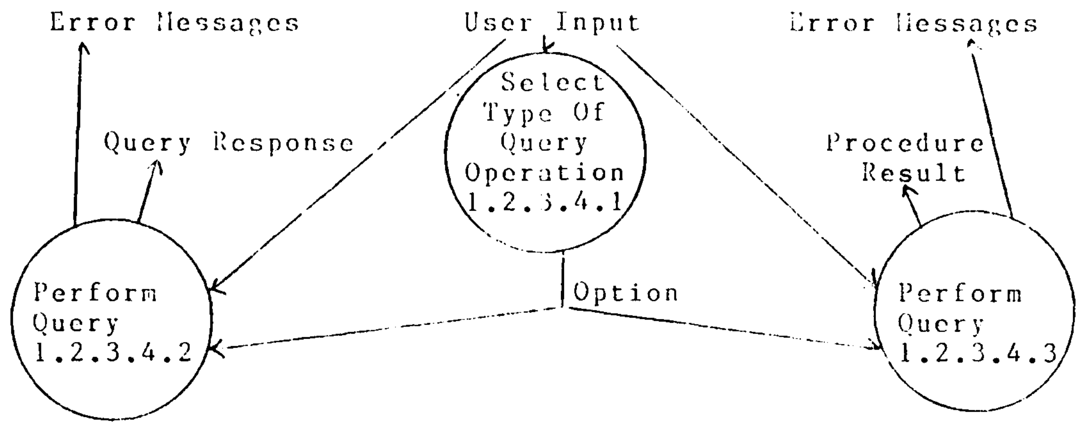


Figure 1-15. Perform Query Operations On Dictionary (1.2.3.4).

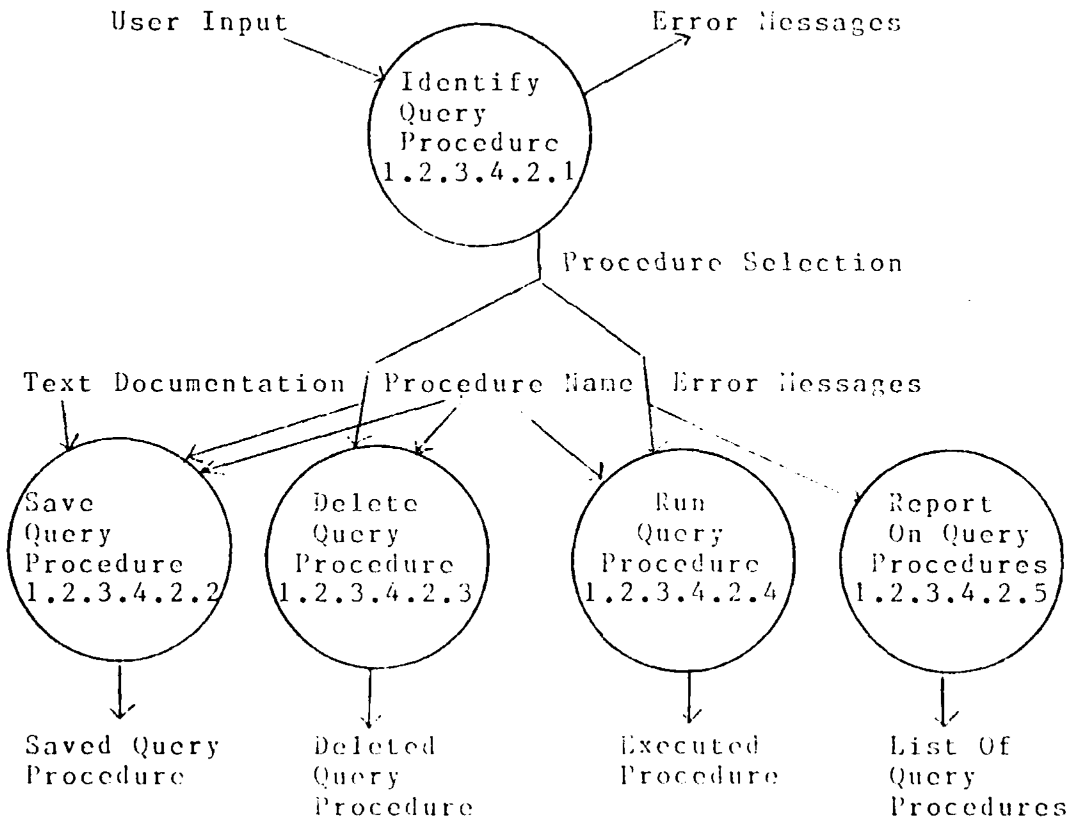


Figure 1-16. Perform Query Procedures (1.2.3.4.2).

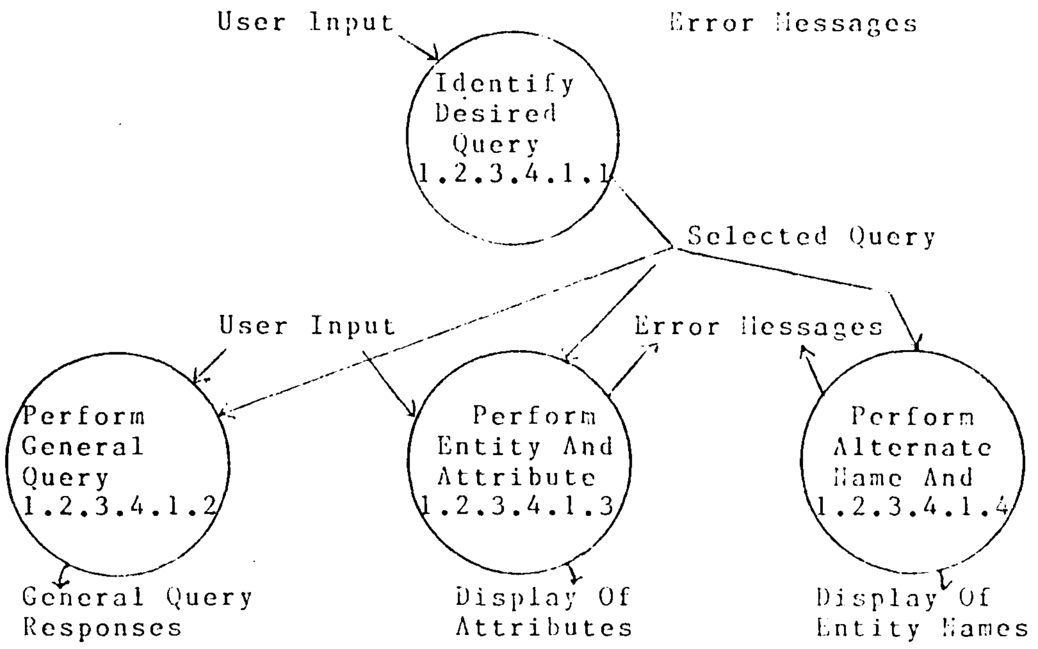

User Input
Error Messages
Identify Desired Query 1.2.3.4.1.1
Selected Query
User Input
Error Messages
Perform General Query 1.2.3.4.1.2
Perform Entity And Attribute 1.2.3.4.1.3
Perform Alternate Name And 1.2.3.4.1.4
General Query Responses
Display Of Attributes
Display Of Entity Names

Figure 1-17. Perform Query (1.2.3.4.1).

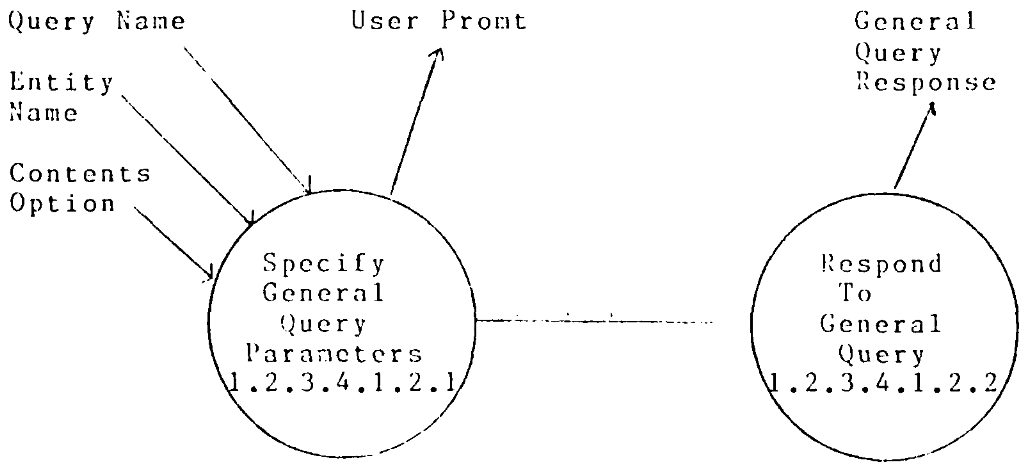

Query Name
User Promt
General Query Response
Entity Name
Contents Option
Specify General Query Parameters 1.2.3.4.1.2.1
Respond To General Query 1.2.3.4.1.2.2

Figure 1-18. Perform General Query (1.2.3.4.1.2).

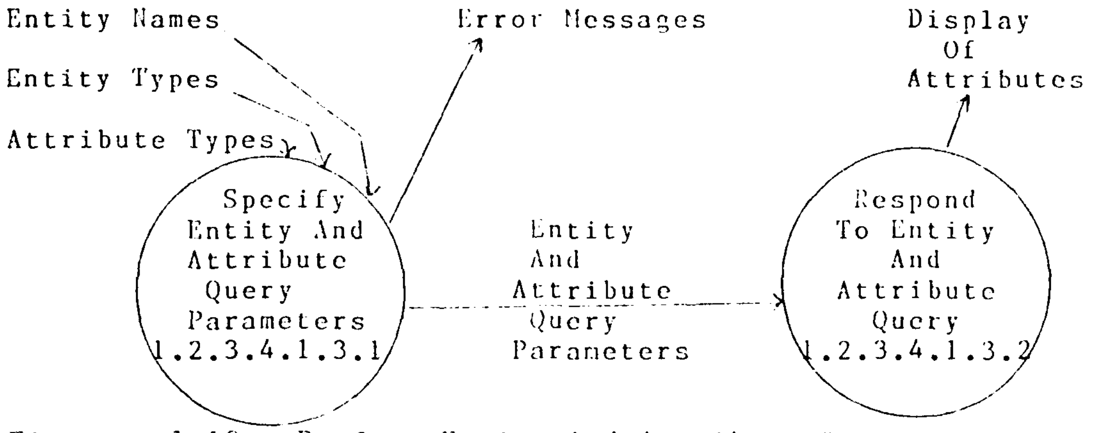

Entity Names
Entity Types
Attribute Types
Error Messages
Display Of Attributes
Specify Entity And Attribute Query Parameters 1.2.3.4.1.3.1
Entity And Attribute Query Parameters
Respond To Entity And Attribute Query 1.2.3.4.1.3.2

Figure 1-19. Perform Entity And Attribute Query (1.2.3.4.1.3).

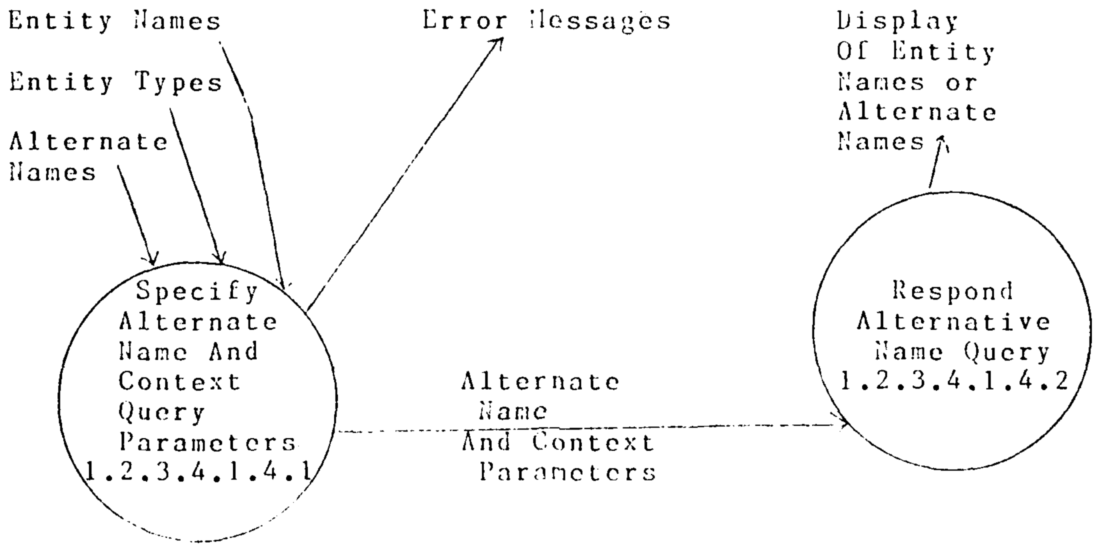

Entity Names
Entity Types
Alternate Names
Error Messages
Display Of Entity Names or Alternate Names
Specify Alternate Name And Context Query Parameters 1.2.3.4.1.4.1
Alternate Name And Context Parameters
Respond Alternative Name Query 1.2.3.4.1.4.2

Figure 1-20. Perform Alternate Name And Context Query (1.2.3.4.1.4).

Appendix B: Structural Model Data Dictionary Generation Tool

The Preliminary Design for the Data Dictionary Generation Tool establishes a structural framework for the application software which forms the interface between the tool user and the data dictionary database. The model presented in this appendix depicts the structural framework of the data dictionary generation tool. This model uses the Structure Chart method to depict the structural framework. The model is introduced and discussed in chapter 3.

Structural Model For the Data Dictionary Generation Tool

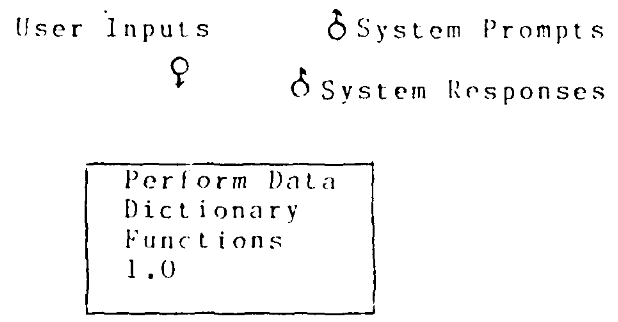


Figure 2-1. Top Level Structural Model.

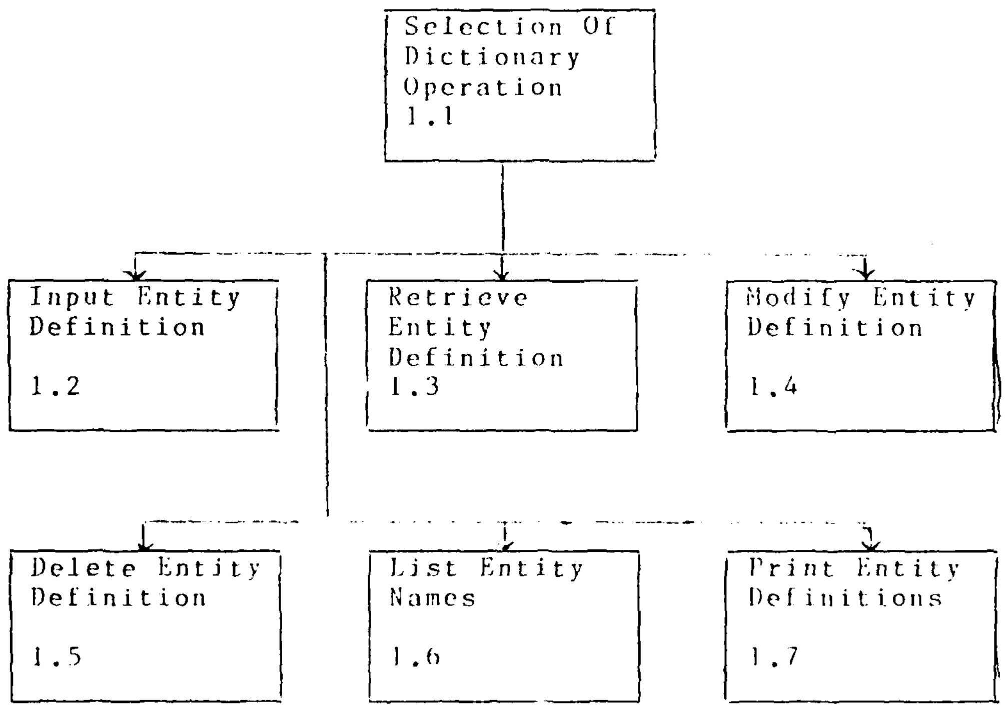

Selection Of Dictionary Operation 1.1
Input Entity Definition 1.2
Retrieve Entity Definition 1.3
Modify Entity Definition 1.4
Delete Entity Definition 1.5
List Entity Names 1.6
Print Entity Definitions 1.7

Figure 2-2. Perform Data Dictionary Functions (1.0).

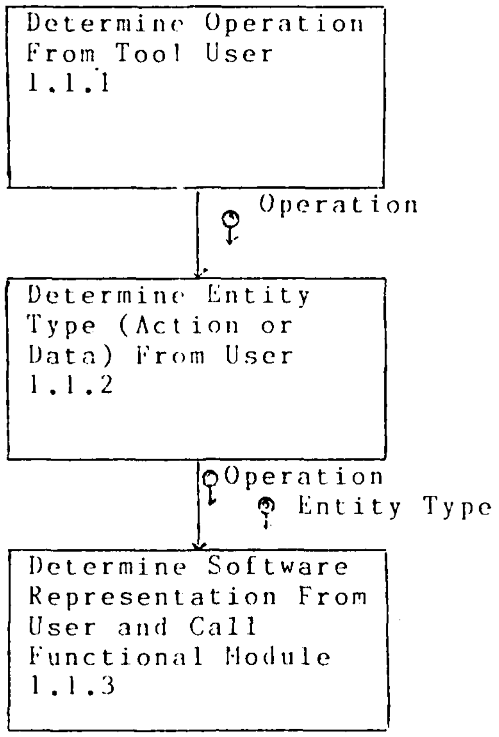

Determine Operation From Tool User 1.1.1
Operation
Determine Entity Type (Action or Data) From User 1.1.2
Operation
Entity Type
Determine Software Representation From User and Call Functional Module 1.1.3

Figure 2-3. Selection of Dictionary Operation (1.1).

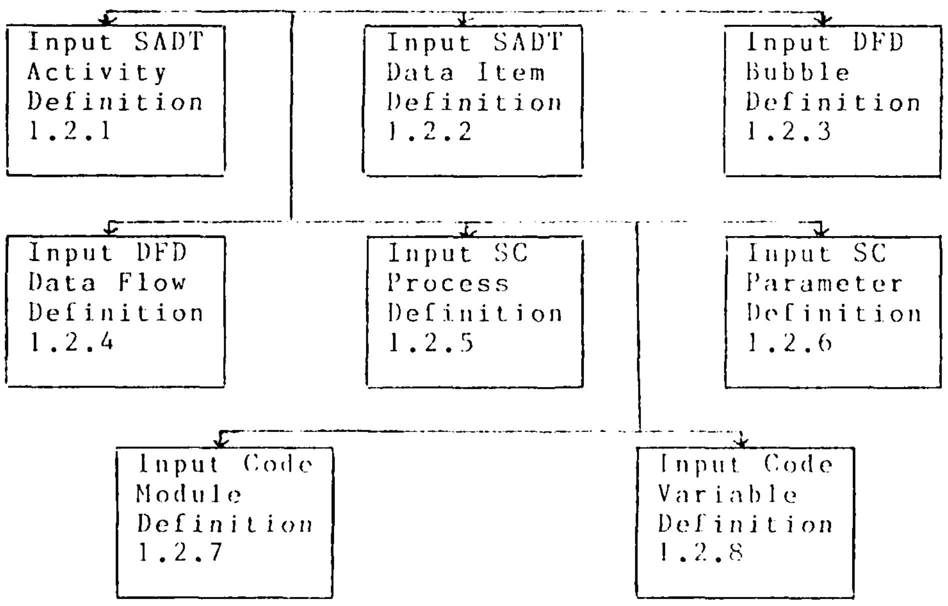
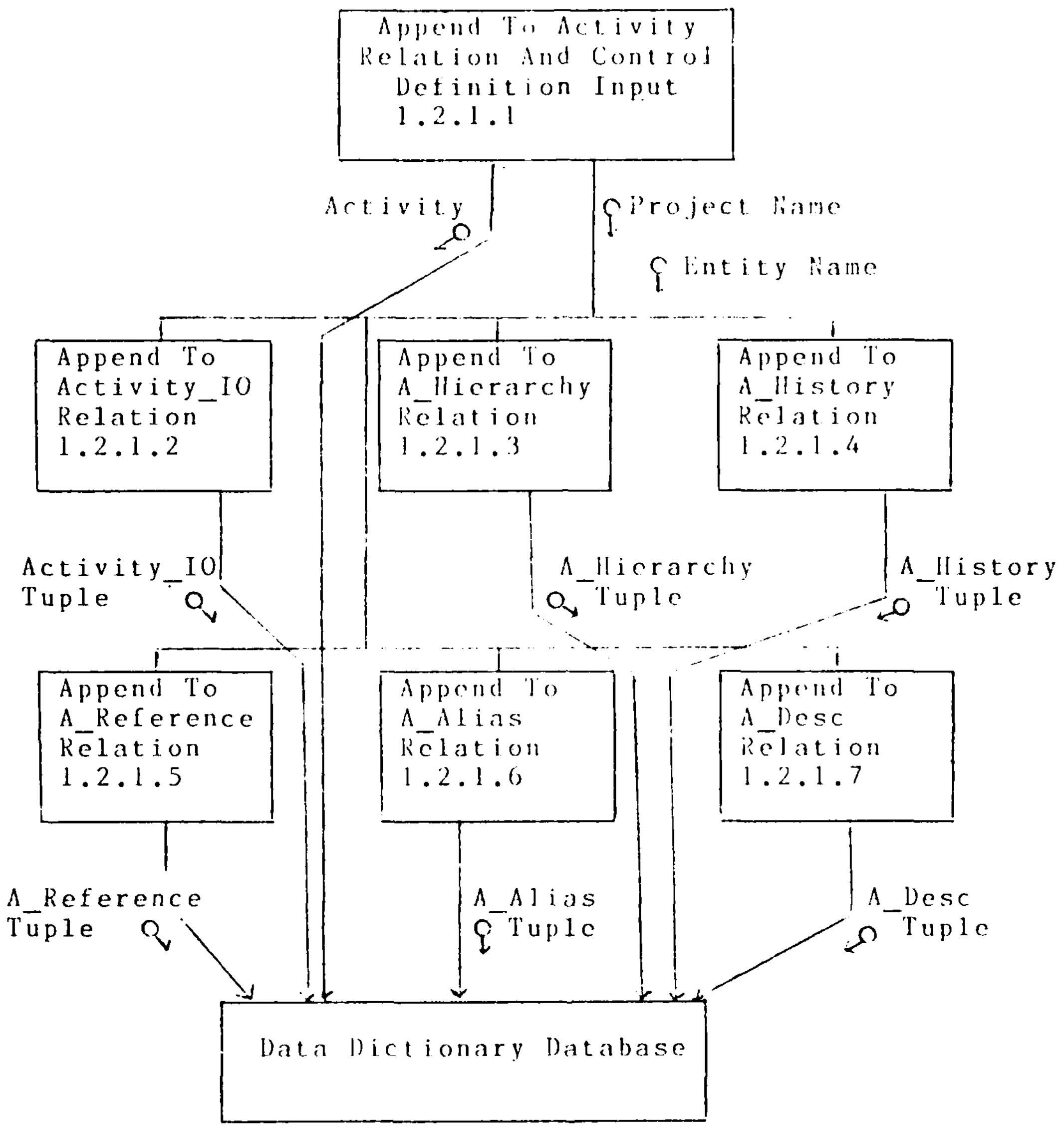

Input SADT Activity Definition 1.2.1
Input SADT Data Item Definition 1.2.2
Input DFD Bubble Definition 1.2.3
Input DFD Data Flow Definition 1.2.4
Input SC Process Definition 1.2.5
Input SC Parameter Definition 1.2.6
Input Code Module Definition 1.2.7
Input Code Variable Definition 1.2.8

Figure 2-4. Input Entity Definition (1.2).



Append To Activity Relation And Control Definition Input 1.2.1.1
Activity
Project Name
Entity Name
Append To Activity_IO Relation 1.2.1.2
Append To A_Hierarchy Relation 1.2.1.3
Append To A_History Relation 1.2.1.4
Activity_IO Tuple
A_Hierarchy Tuple
A_History Tuple
Append To A_Reference Relation 1.2.1.5
Append To A_Alias Relation 1.2.1.6
Append To A_Desc Relation 1.2.1.7
A_Reference Tuple
A_Alias Tuple
A_Desc Tuple
Data Dictionary Database

Figure 2-5. Input SADT Activity Definition (1.2.1).

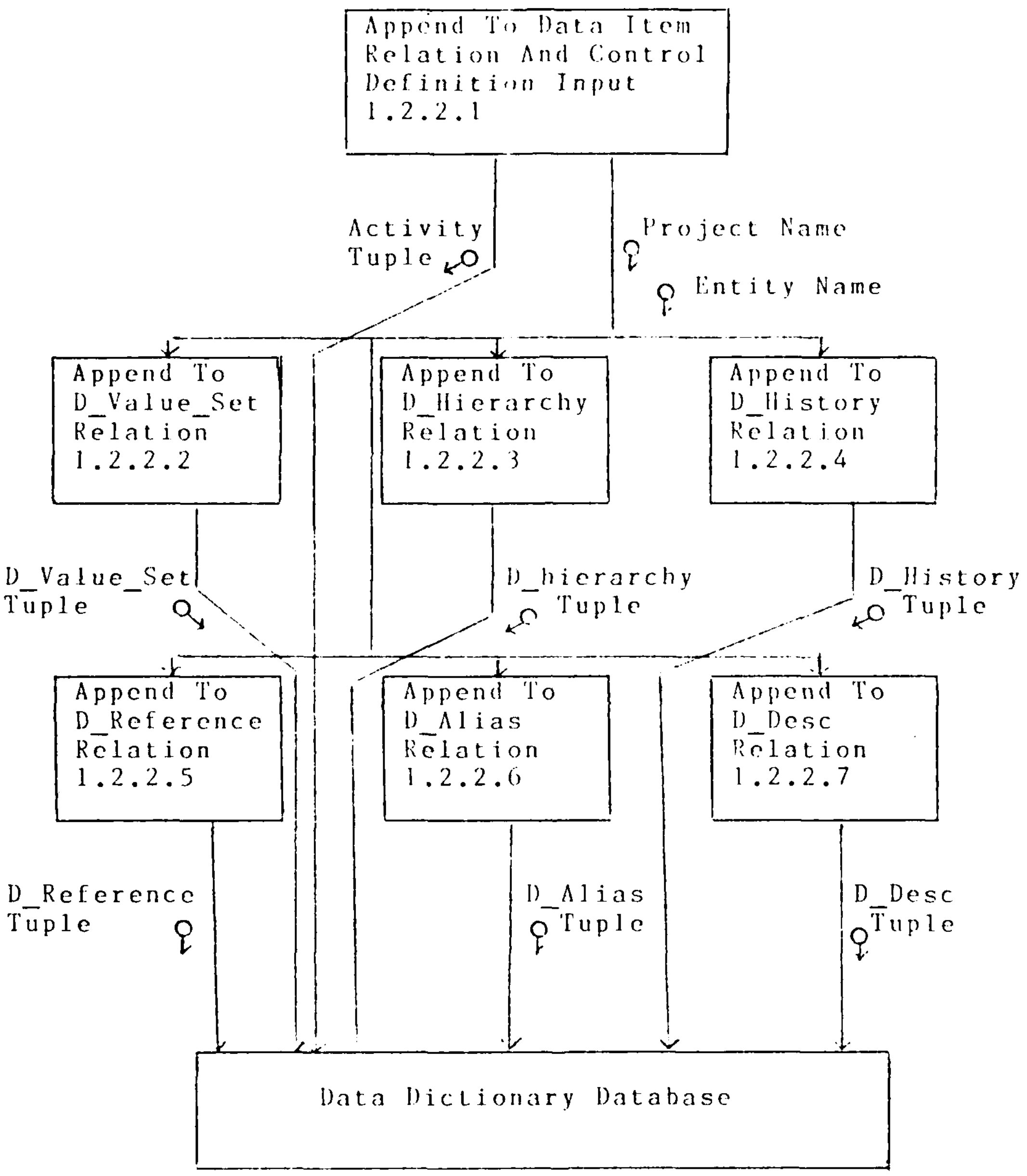

Append To Data Item Relation And Control Definition Input 1.2.2.1
Activity Tuple
Project Name
Entity Name
Append To D_Value_Set Relation 1.2.2.2
Append To D_Hierarchy Relation 1.2.2.3
Append To D_History Relation 1.2.2.4
D_Value_Set Tuple
D_hierarchy Tuple
D_History Tuple
Append To D_Reference Relation 1.2.2.5
Append To D_Alias Relation 1.2.2.6
Append To D_Desc Relation 1.2.2.7
D_Reference Tuple
D_Alias Tuple
D_Desc Tuple
Data Dictionary Database

Figure 2-6. Input SADT Data Item Definition (1.2.2).

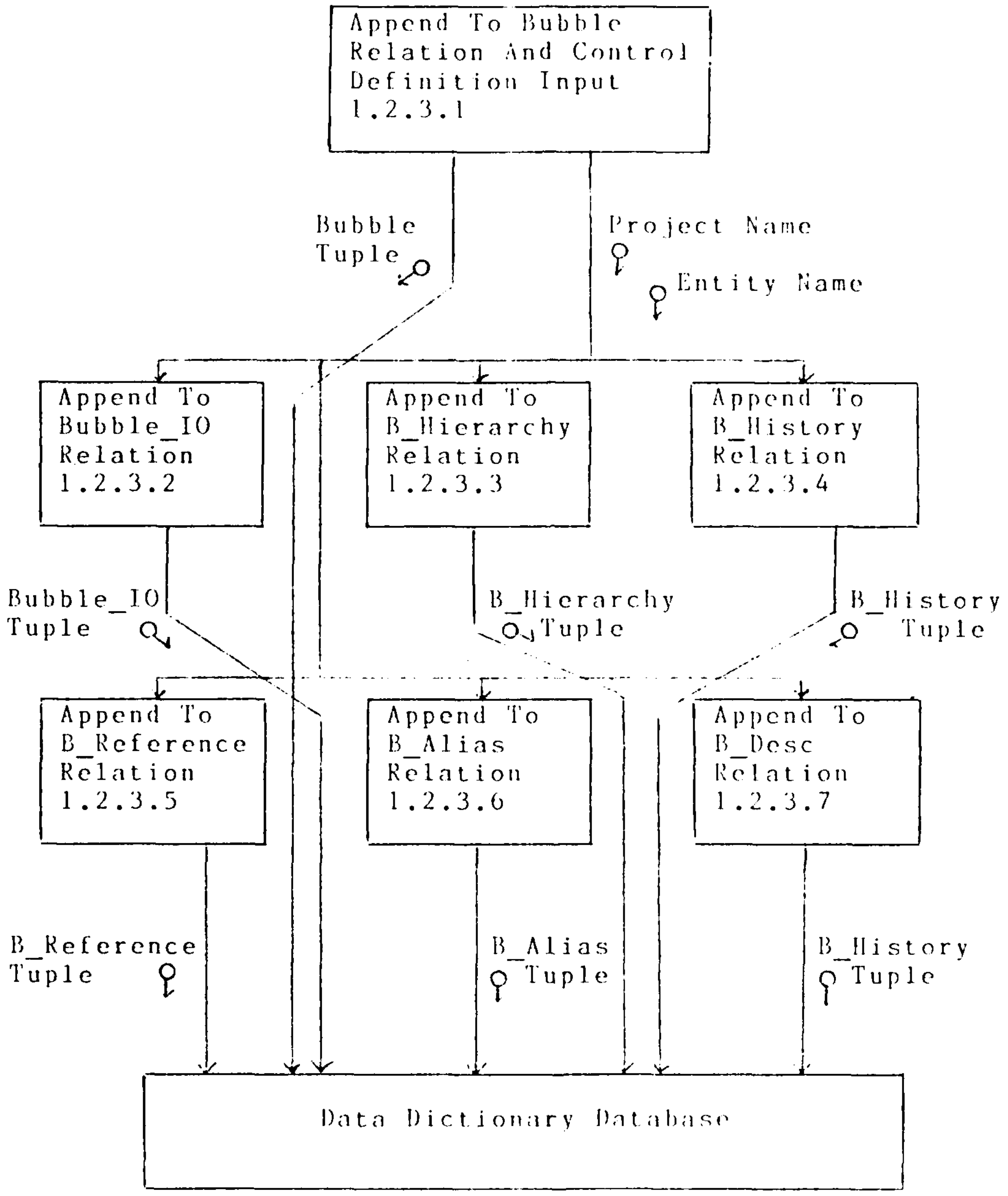

Append To Bubble Relation And Control Definition Input 1.2.3.1
Bubble Tuple
Project Name
Entity Name
Append To Bubble_IO Relation 1.2.3.2
Append To B_Hierarchy Relation 1.2.3.3
Append To B_History Relation 1.2.3.4
Bubble_IO Tuple
B_Hierarchy Tuple
B_History Tuple
Append To B_Reference Relation 1.2.3.5
Append To B_Alias Relation 1.2.3.6
Append To B_Desc Relation 1.2.3.7
B_Reference Tuple
B_Alias Tuple
B_History Tuple
Data Dictionary Database

Figure 2-7. Input DFD Bubble Definition (1.2.3).

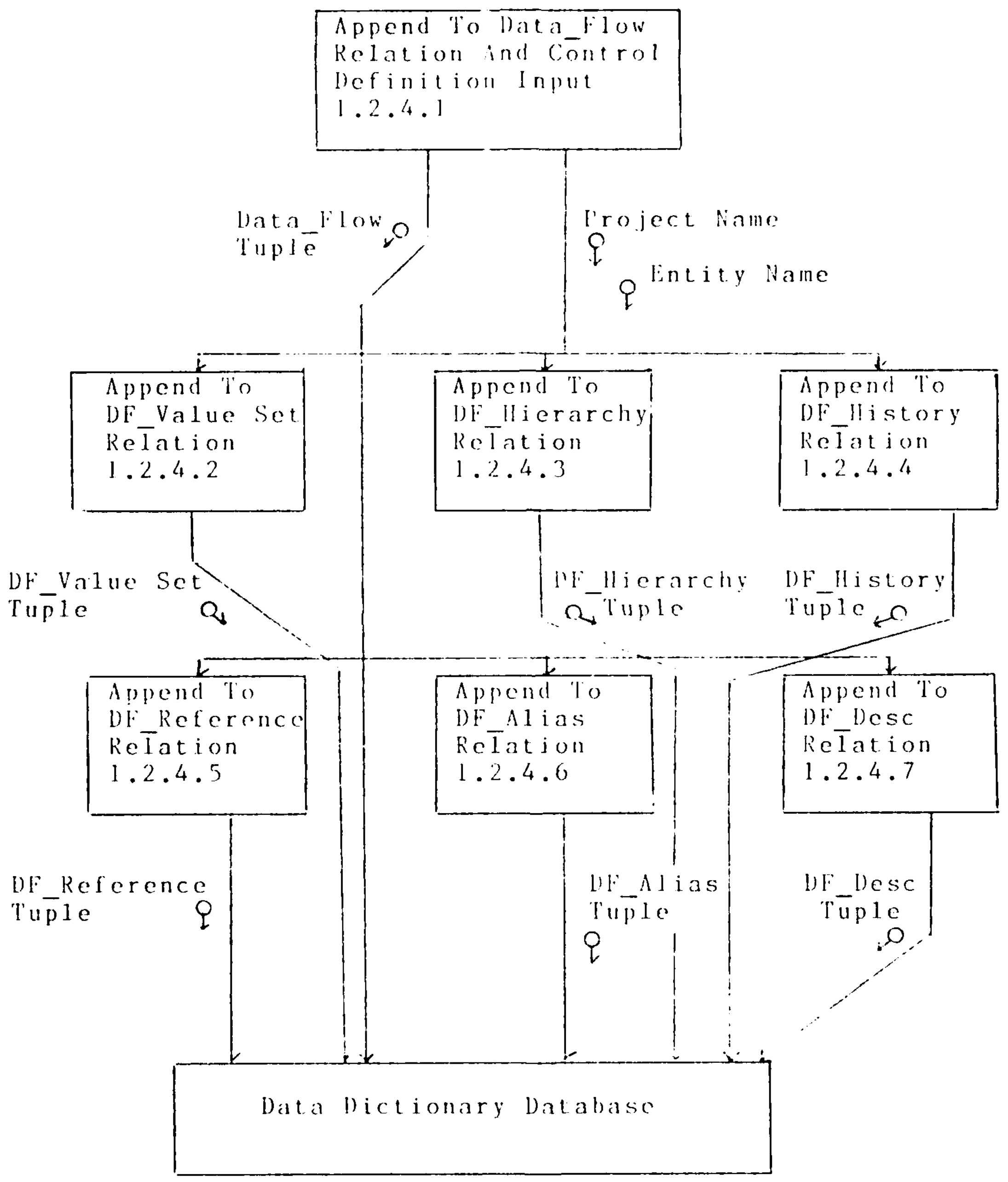

Append To Data_Flow Relation And Control Definition Input 1.2.4.1
Data_Flow Tuple
Project Name
Entity Name
Append To DF_Value Set Relation 1.2.4.2
Append To DF_Hierarchy Relation 1.2.4.3
Append To DF_History Relation 1.2.4.4
DF_Value Set Tuple
DF_Hierarchy Tuple
DF_History Tuple
Append To DF_Reference Relation 1.2.4.5
Append To DF_Alias Relation 1.2.4.6
Append To DF_Desc Relation 1.2.4.7
DF_Reference Tuple
DF_Alias Tuple
DF_Desc Tuple
Data Dictionary Database

Figure 2-8. Input DFD Data Flow Definition (1.2.4).

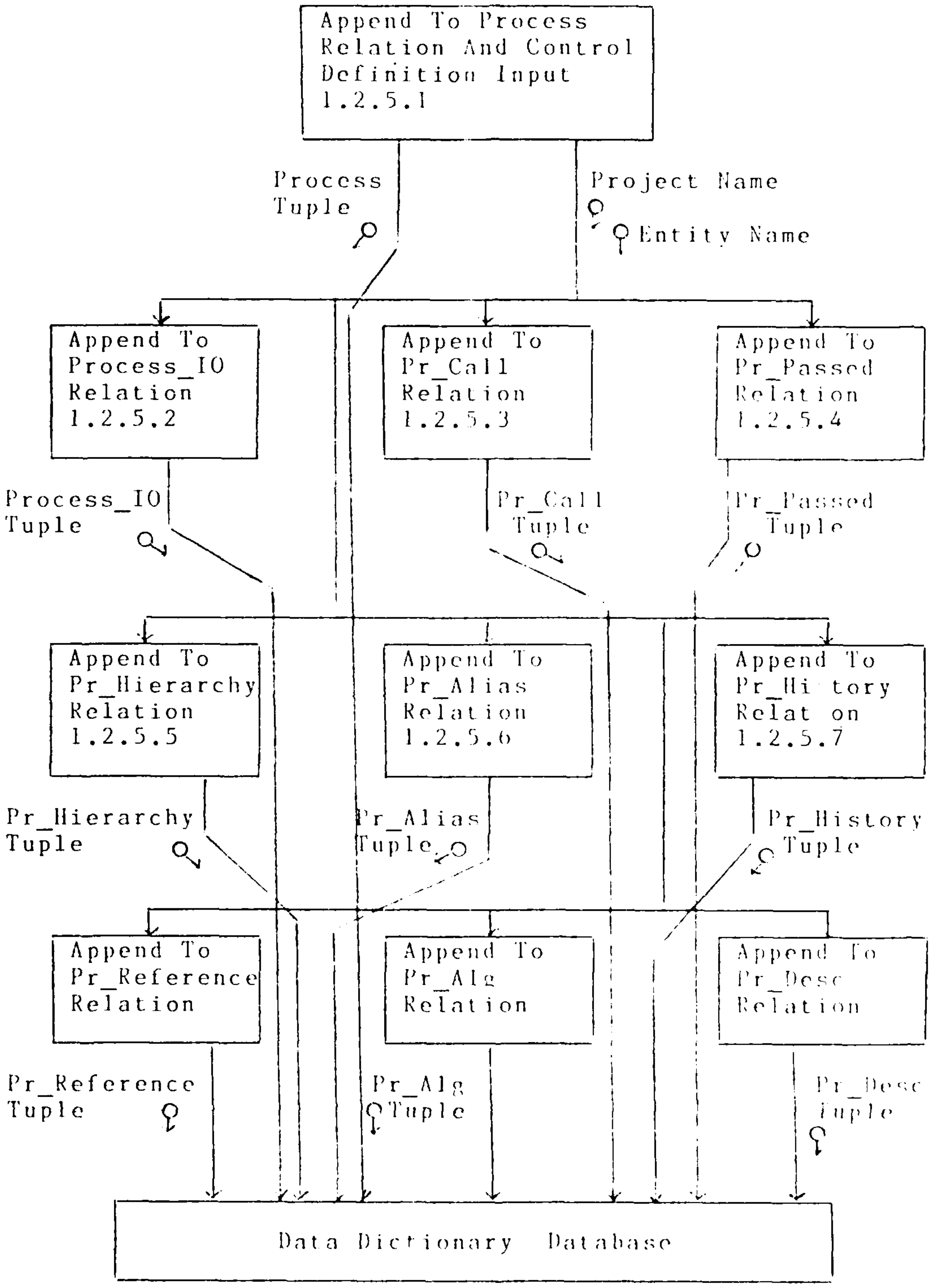

Append To Process Relation And Control Definition Input 1.2.5.1
Process Tuple
Project Name
Entity Name
Append To Process_IO Relation 1.2.5.2
Append To Pr_Call Relation 1.2.5.3
Append To Pr_Passed Relation 1.2.5.4
Process_IO Tuple
Pr_Call Tuple
Pr_Passed Tuple
Append To Pr_Hierarchy Relation 1.2.5.5
Append To Pr_Alias Relation 1.2.5.6
Append To Pr_History Relation 1.2.5.7
Pr_Hierarchy Tuple
Pr_Alias Tuple
Pr_History Tuple
Append To Pr_Reference Relation
Append To Pr_Alg Relation
Append To Pr_Desc Relation
Pr_Reference Tuple
Pr_Alg Tuple
Pr_Desc Tuple
Data Dictionary Database

Figure 2-9. Input SC Process Defintion (1.2.5).

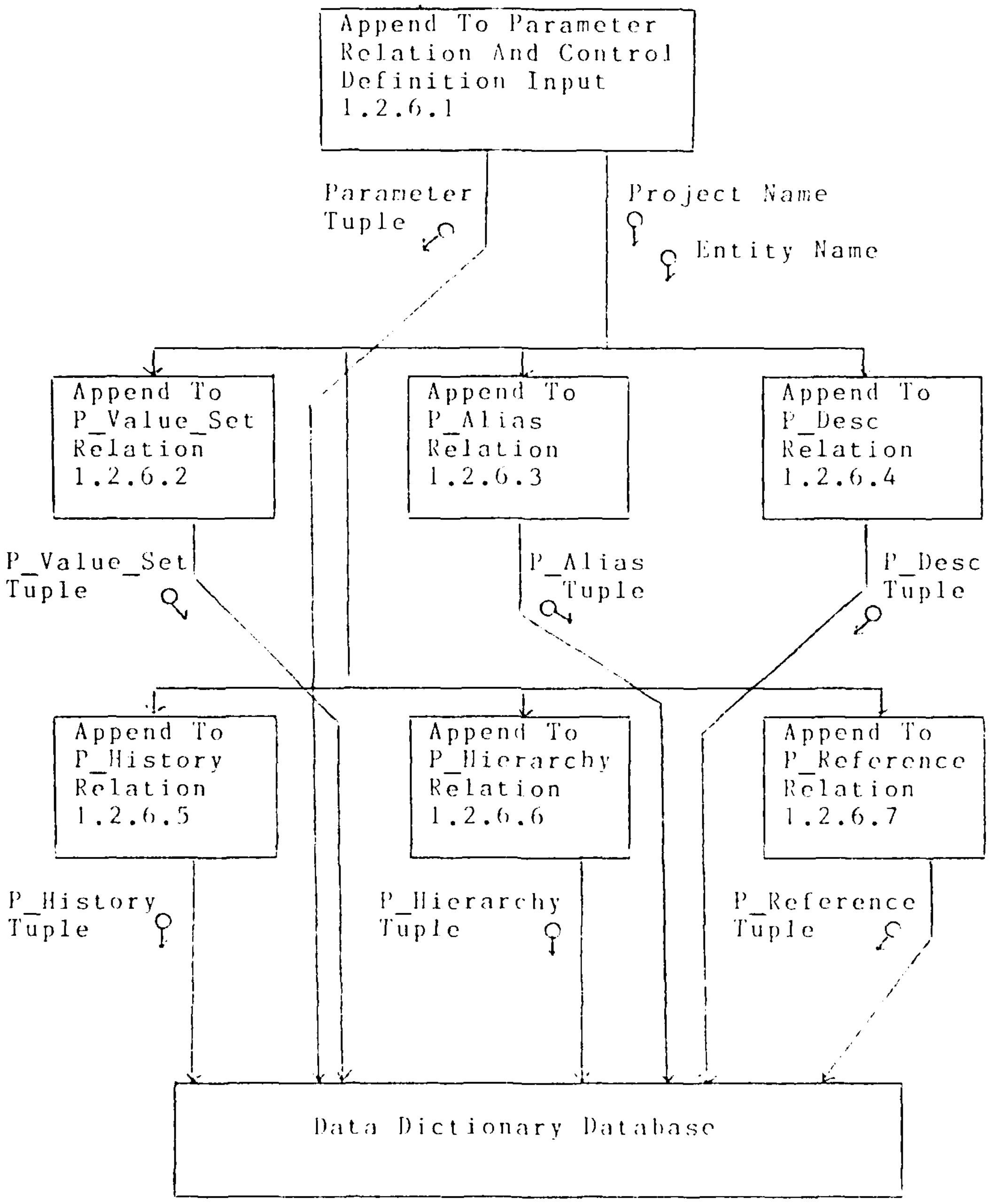

Append To Parameter Relation And Control Definition Input 1.2.6.1
Parameter Tuple
Project Name
Entity Name
Append To P_Value_Set Relation 1.2.6.2
Append To P_Alias Relation 1.2.6.3
Append To P_Desc Relation 1.2.6.4
P_Value_Set Tuple
P_Alias Tuple
P_Desc Tuple
Append To P_History Relation 1.2.6.5
Append To P_Hierarchy Relation 1.2.6.6
Append To P_Reference Relation 1.2.6.7
P_History Tuple
P_Hierarchy Tuple
P_Reference Tuple
Data Dictionary Database

Figure 2-10. Input SC Parameter Definition (1.2.6).

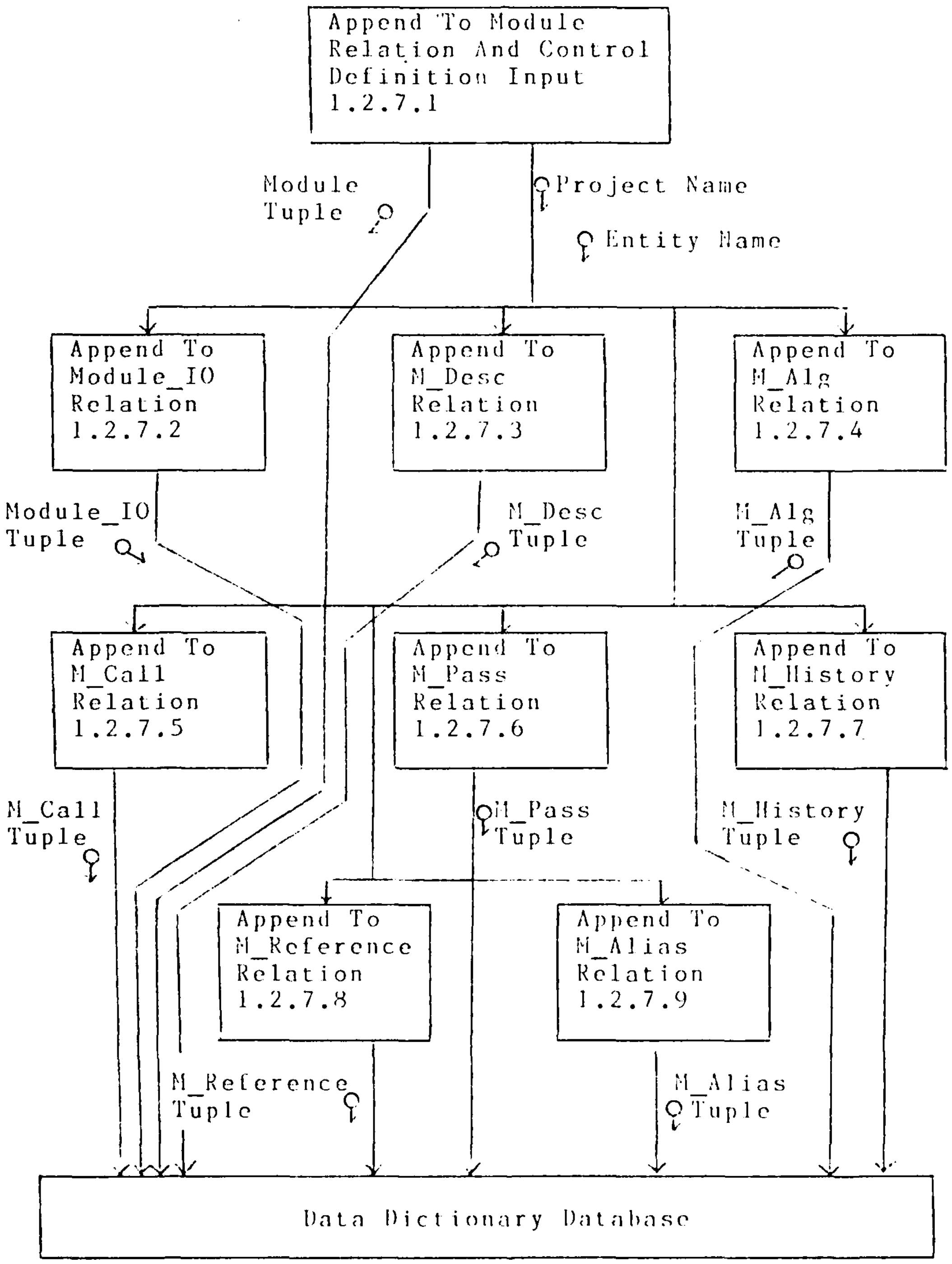

Append To Module Relation And Control Definition Input 1.2.7.1
Module Tuple
Project Name
Entity Name
Append To Module_IO Relation 1.2.7.2
Append To M_Desc Relation 1.2.7.3
Append To M_Alg Relation 1.2.7.4
Module_IO Tuple
M_Desc Tuple
M_Alg Tuple
Append To M_Call Relation 1.2.7.5
Append To M_Pass Relation 1.2.7.6
Append To M_History Relation 1.2.7.7
M_Call Tuple
M_Pass Tuple
M_History Tuple
Append To M_Reference Relation 1.2.7.8
Append To M_Alias Relation 1.2.7.9
M_Reference Tuple
M_Alias Tuple
Data Dictionary Database

Figure 2-11. Input Code Module Definition (1.2.7).

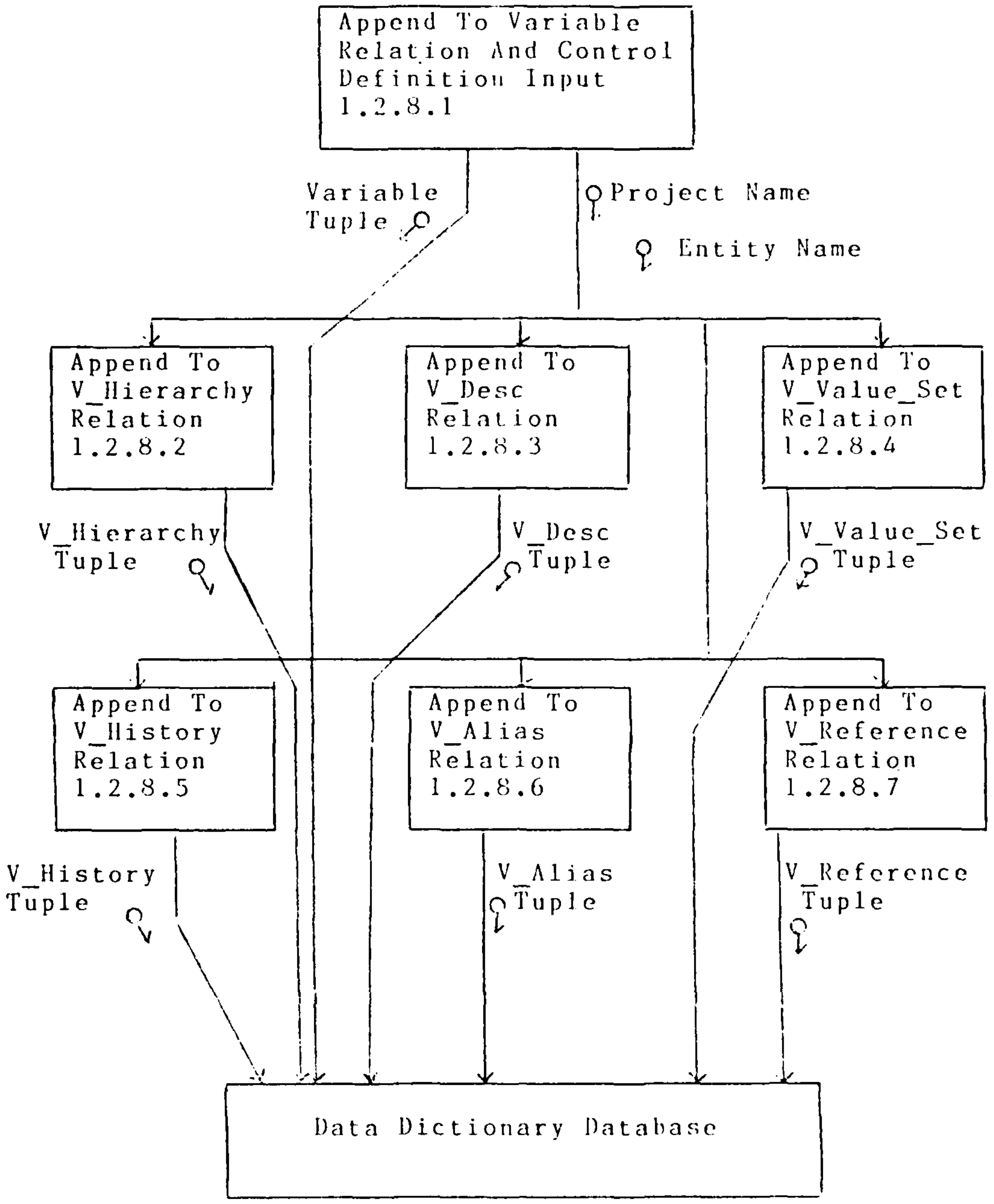

Append To Variable Relation And Control Definition Input 1.2.8.1
Variable Tuple
Project Name
Entity Name
Append To V_Hierarchy Relation 1.2.8.2
Append To V_Desc Relation 1.2.8.3
Append To V_Value_Set Relation 1.2.8.4
V_Hierarchy Tuple
V_Desc Tuple
V_Value_Set Tuple
Append To V_History Relation 1.2.8.5
Append To V_Alias Relation 1.2.8.6
Append To V_Reference Relation 1.2.8.7
V_History Tuple
V_Alias Tuple
V_Reference Tuple
Data Dictionary Database

Figure 2-12. Input Code Variable Definition (1.2.8).

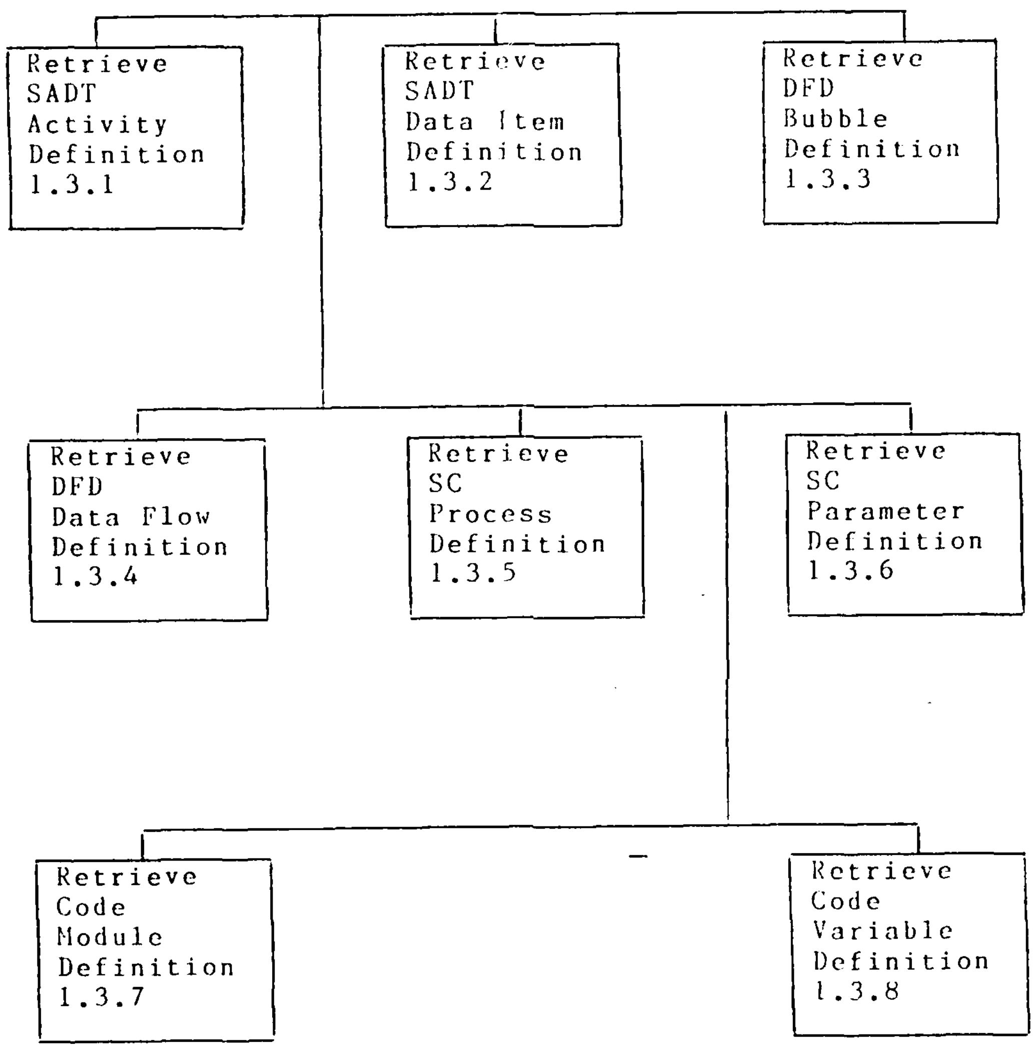

Retrieve SADT Activity Definition 1.3.1
Retrieve SADT Data Item Definition 1.3.2
Retrieve DFD Bubble Definition 1.3.3
Retrieve DFD Data Flow Definition 1.3.4
Retrieve SC Process Definition 1.3.5
Retrieve SC Parameter Definition 1.3.6
Retrieve Code Module Definition 1.3.7
Retrieve Code Variable Definition 1.3.8

Figure 2-13. Retrieve Entity Definition (1.3).

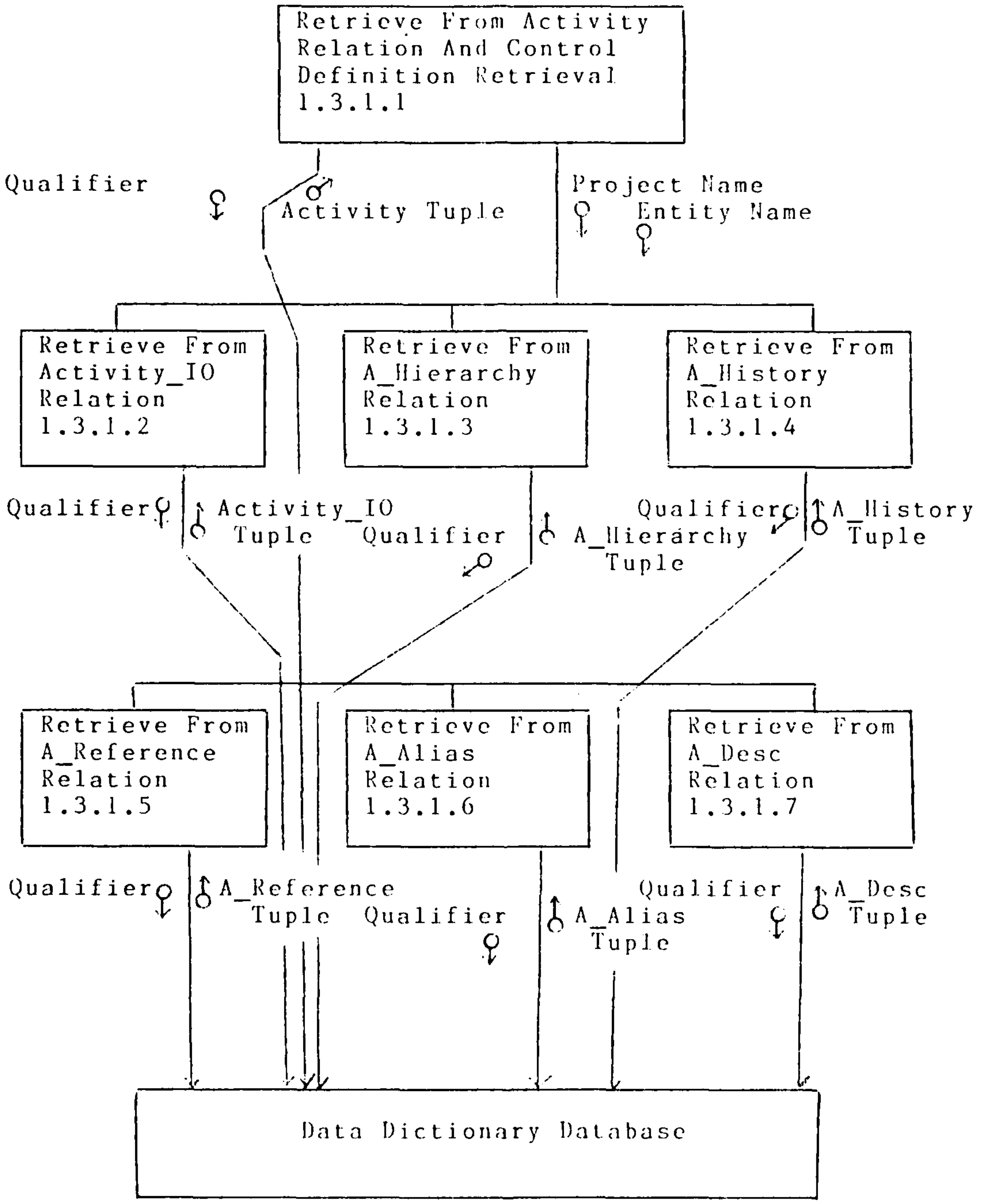

Retrieve From Activity Relation And Control Definition Retrieval 1.3.1.1
Qualifier
Activity Tuple
Project Name
Entity Name
Retrieve From Activity_IO Relation 1.3.1.2
Retrieve From A_Hierarchy Relation 1.3.1.3
Retrieve From A_History Relation 1.3.1.4
Qualifier
Activity_IO Tuple
Qualifier
A_Hierarchy Tuple
Qualifier
A_History Tuple
Retrieve From A_Reference Relation 1.3.1.5
Retrieve From A_Alias Relation 1.3.1.6
Retrieve From A_Desc Relation 1.3.1.7
Qualifier
A_Reference Tuple
Qualifier
A_Alias Tuple
Qualifier
A_Desc Tuple
Data Dictionary Database

Figure 2-14. Retrieve SADT Activity Definition (1.3.1)

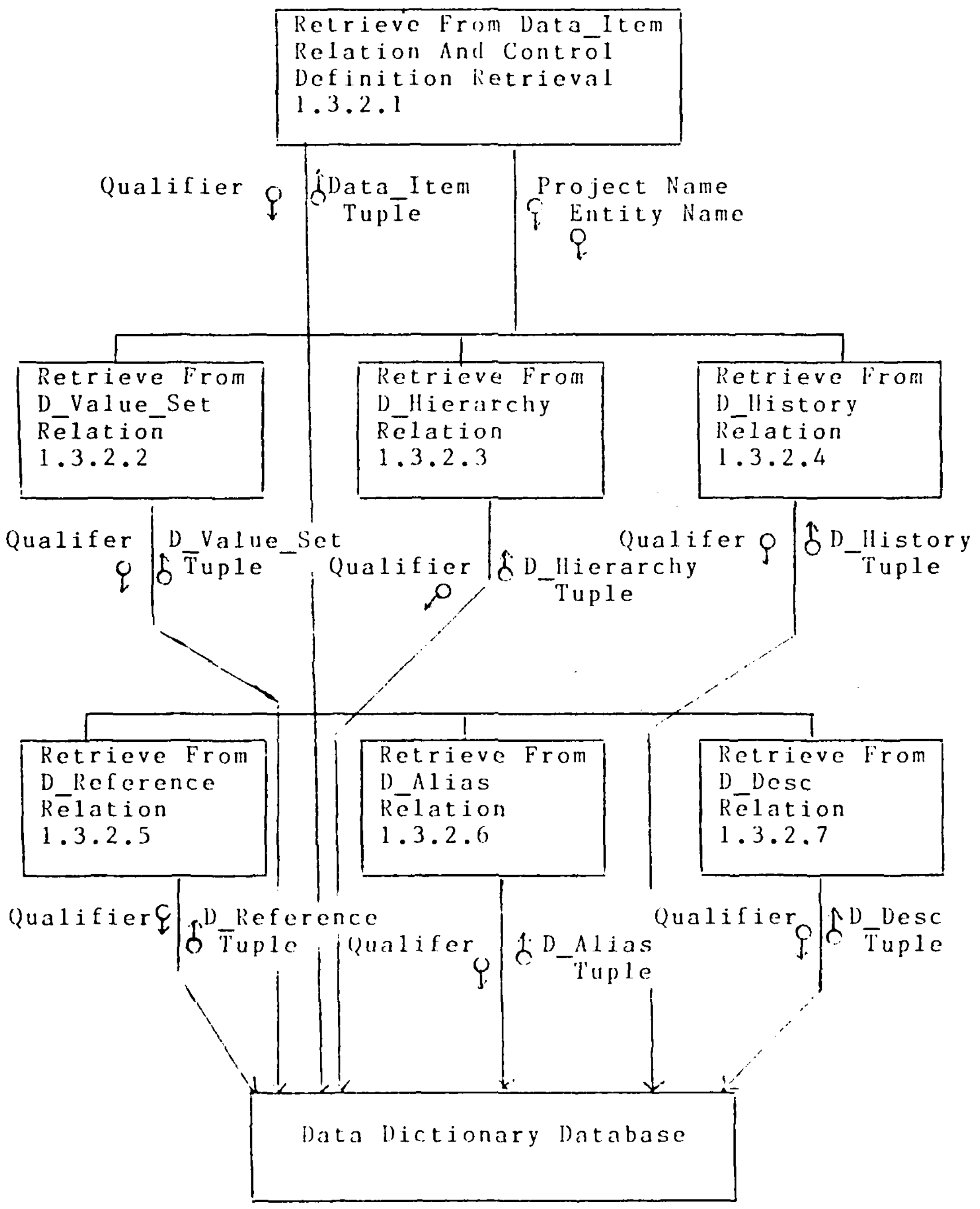

Retrieve From Data_Item Relation And Control Definition Retrieval 1.3.2.1
Qualifier
Data_Item Tuple
Project Name
Entity Name
Retrieve From D_Value_Set Relation 1.3.2.2
Retrieve From D_Hierarchy Relation 1.3.2.3
Retrieve From D_History Relation 1.3.2.4
Qualifer
D_Value_Set Tuple
Qualifier
D_Hierarchy Tuple
Qualifer
D_History Tuple
Retrieve From D_Reference Relation 1.3.2.5
Retrieve From D_Alias Relation 1.3.2.6
Retrieve From D_Desc Relation 1.3.2.7
Qualifier
D_Reference Tuple
Qualifer
D_Alias Tuple
Qualifier
D_Desc Tuple
Data Dictionary Database

Figure 2-15. Retrieve SADT Data Item Definition (1.3.2).

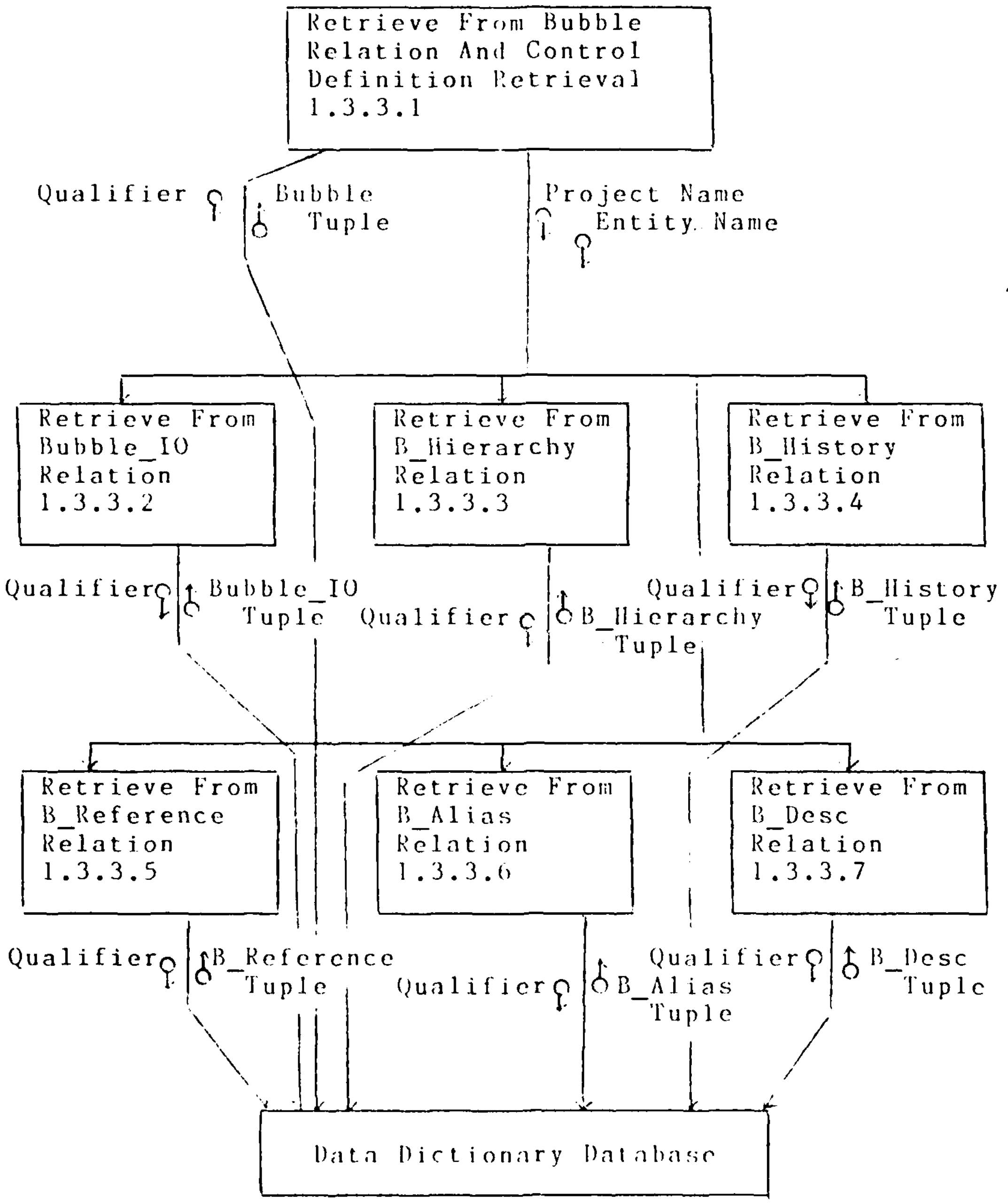

Retrieve From Bubble Relation And Control Definition Retrieval 1.3.3.1
Qualifier
Bubble Tuple
Project Name
Entity Name
Retrieve From Bubble_IO Relation 1.3.3.2
Retrieve From B_Hierarchy Relation 1.3.3.3
Retrieve From B_History Relation 1.3.3.4
Qualifier
Bubble_IO Tuple
Qualifier
B_Hierarchy Tuple
Qualifier
B_History Tuple
Retrieve From B_Reference Relation 1.3.3.5
Retrieve From B_Alias Relation 1.3.3.6
Retrieve From B_Desc Relation 1.3.3.7
Qualifier
B_Reference Tuple
Qualifier
B_Alias Tuple
Qualifier
B_Desc Tuple
Data Dictionary Database

Figure 2-16. Retrieve DFD Bubble Definition (1.3.3)>

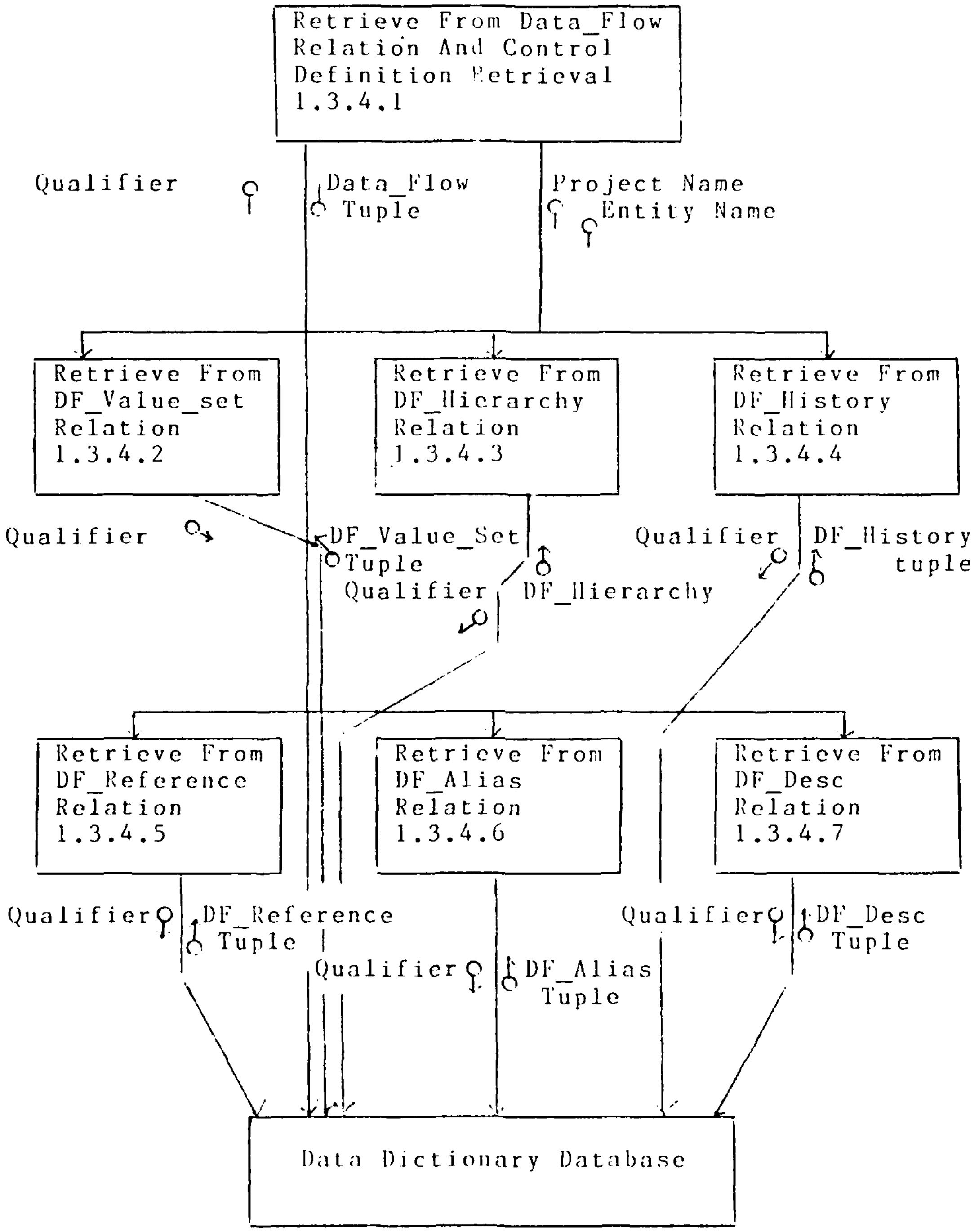
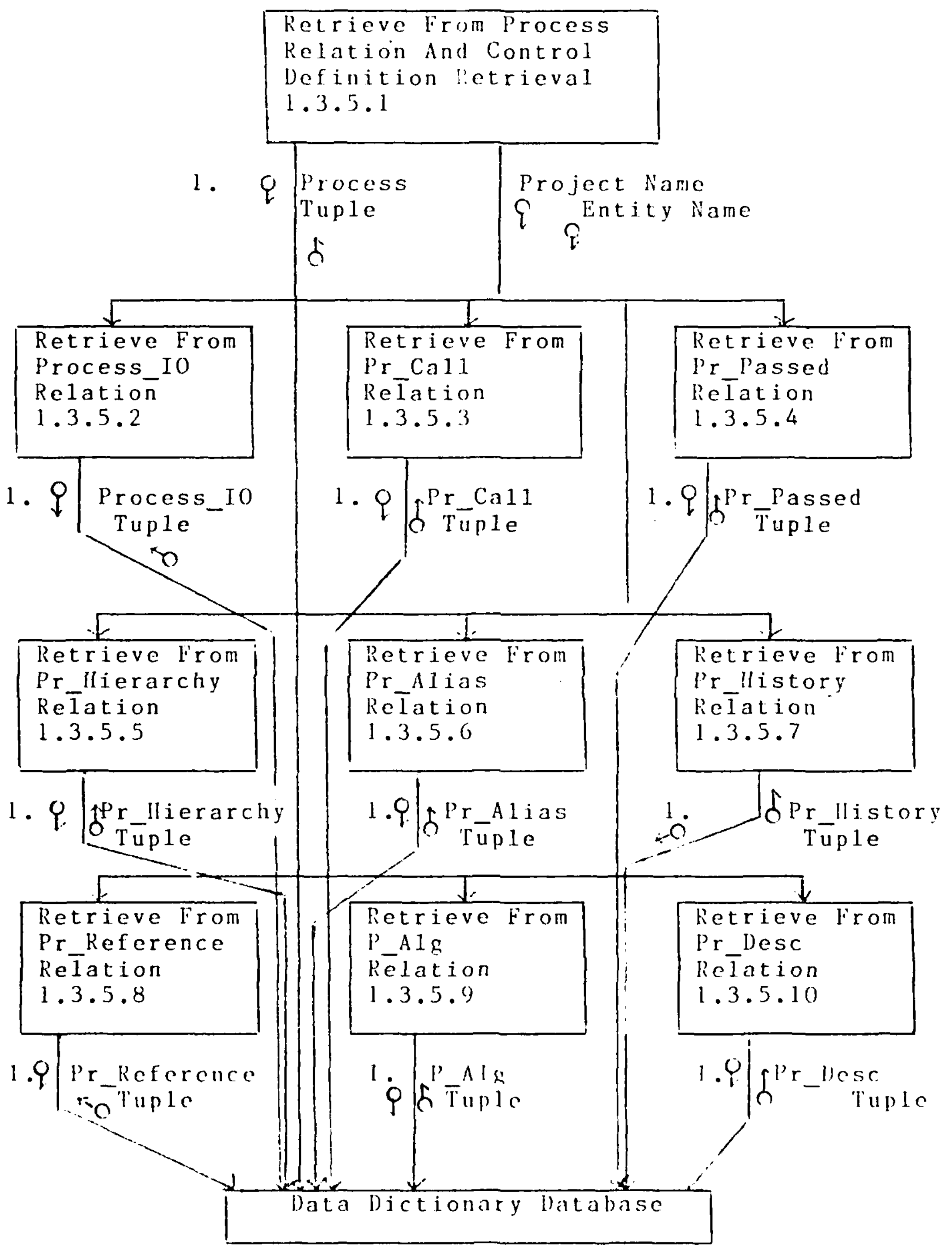

Retrieve From Data_Flow Relation And Control Definition Retrieval 1.3.4.1
Qualifier
Data_Flow Tuple
Project Name
Entity Name
Retrieve From DF_Value_set Relation 1.3.4.2
Retrieve From DF_Hierarchy Relation 1.3.4.3
Retrieve From DF_History Relation 1.3.4.4
Qualifier
DF_Value_Set Tuple
Qualifier
DF_Hierarchy
Qualifier
DF_History tuple
Retrieve From DF_Reference Relation 1.3.4.5
Retrieve From DF_Alias Relation 1.3.4.6
Retrieve From DF_Desc Relation 1.3.4.7
Qualifier
DF_Reference Tuple
Qualifier
DF_Alias Tuple
Qualifier
DF_Desc Tuple
Data Dictionary Database

Figure 2-17. Retrieve DFD Data_Flow Definition (1.3.4).

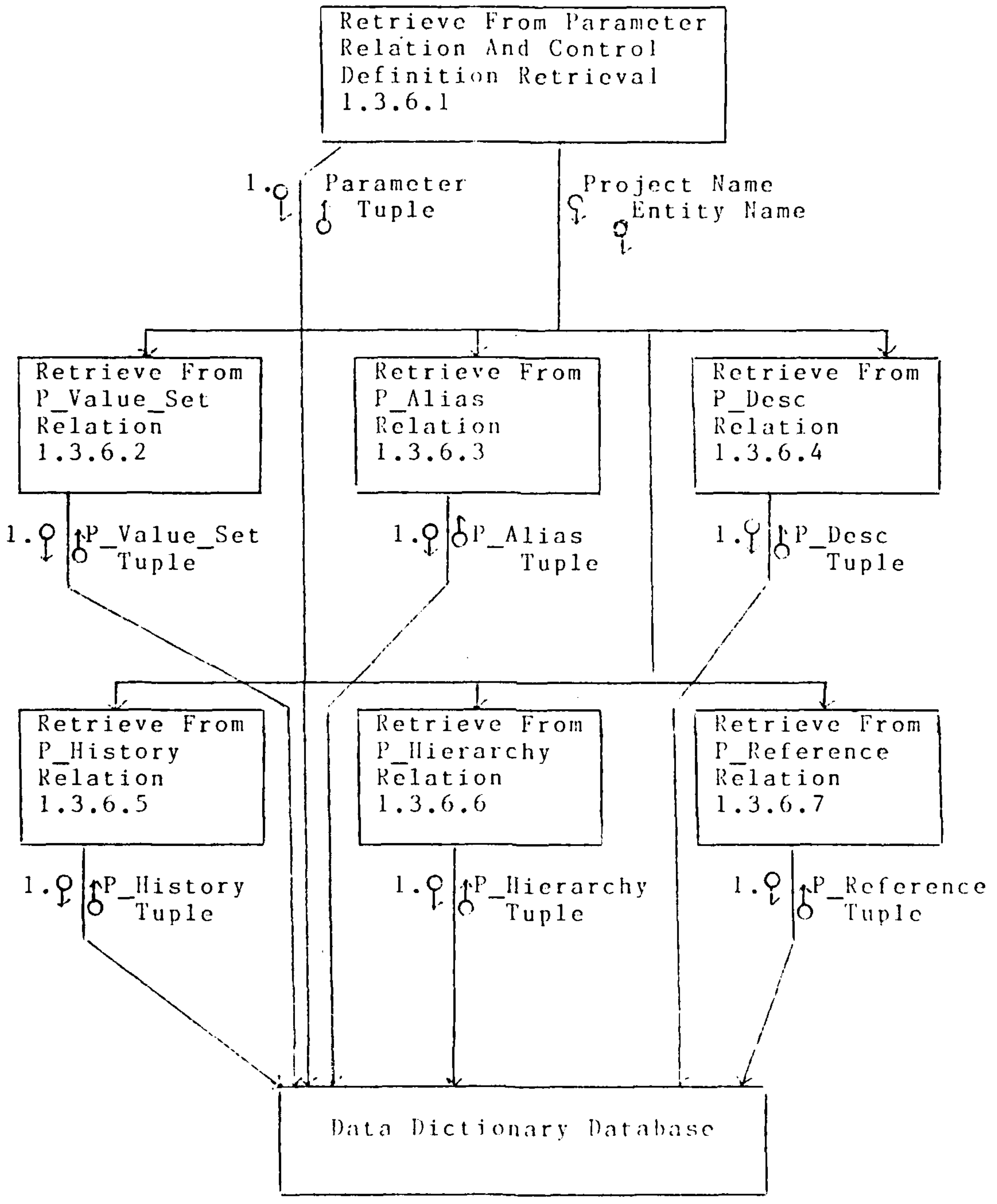
1. Qualifier



Retrieve From Process Relation And Control Definition Retrieval 1.3.5.1
1.
Process Tuple
Project Name
Entity Name
Retrieve From Process_IO Relation 1.3.5.2
Retrieve From Pr_Call Relation 1.3.5.3
Retrieve From Pr_Passed Relation 1.3.5.4
1.
Process_IO Tuple
1.
Pr_Call Tuple
1.
Pr_Passed Tuple
Retrieve From Pr_Hierarchy Relation 1.3.5.5
Retrieve From Pr_Alias Relation 1.3.5.6
Retrieve From Pr_History Relation 1.3.5.7
1.
Pr_Hierarchy Tuple
1.
Pr_Alias Tuple
1.
Pr_History Tuple
Retrieve From Pr_Reference Relation 1.3.5.8
Retrieve From P_Alg Relation 1.3.5.9
Retrieve From Pr_Desc Relation 1.3.5.10
1.
Pr_Reference Tuple
1.
P_Alg Tuple
1.
Pr_Desc Tuple
Data Dictionary Database

Figure 2-18. Retrieve SC Process Definition (1.3.5).

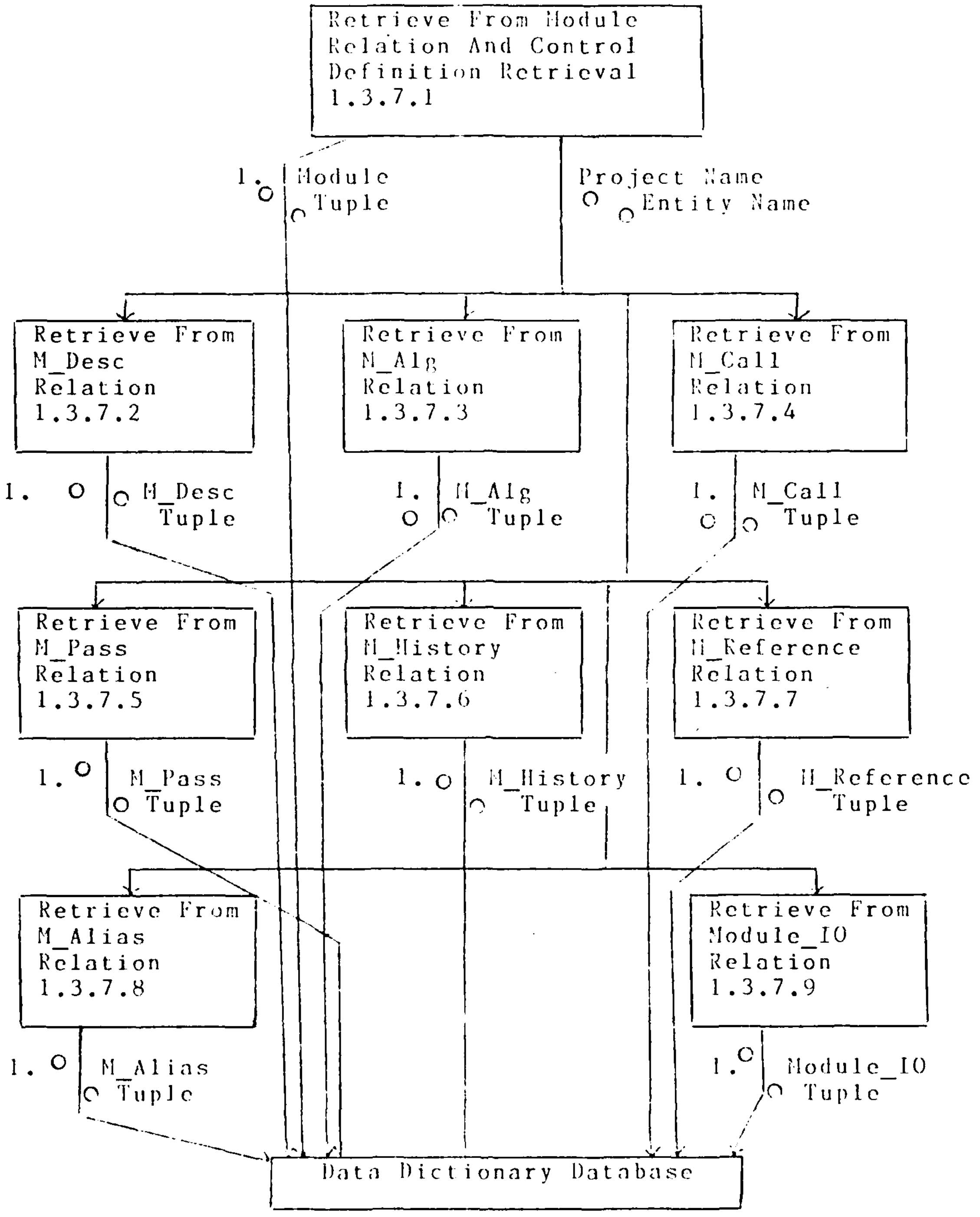
1. - Qualifier



Retrieve From Parameter Relation And Control Definition Retrieval 1.3.6.1
1.
Parameter Tuple
Project Name
Entity Name
Retrieve From P_Value_Set Relation 1.3.6.2
Retrieve From P_Alias Relation 1.3.6.3
Retrieve From P_Desc Relation 1.3.6.4
1.
P_Value_Set Tuple
1.
P_Alias Tuple
1.
P_Desc Tuple
Retrieve From P_History Relation 1.3.6.5
Retrieve From P_Hierarchy Relation 1.3.6.6
Retrieve From P_Reference Relation 1.3.6.7
1.
P_History Tuple
1.
P_Hierarchy Tuple
1.
P_Reference Tuple
Data Dictionary Database

Figure 2-19. Retrieve SC Parameter Definition (1.3.6).

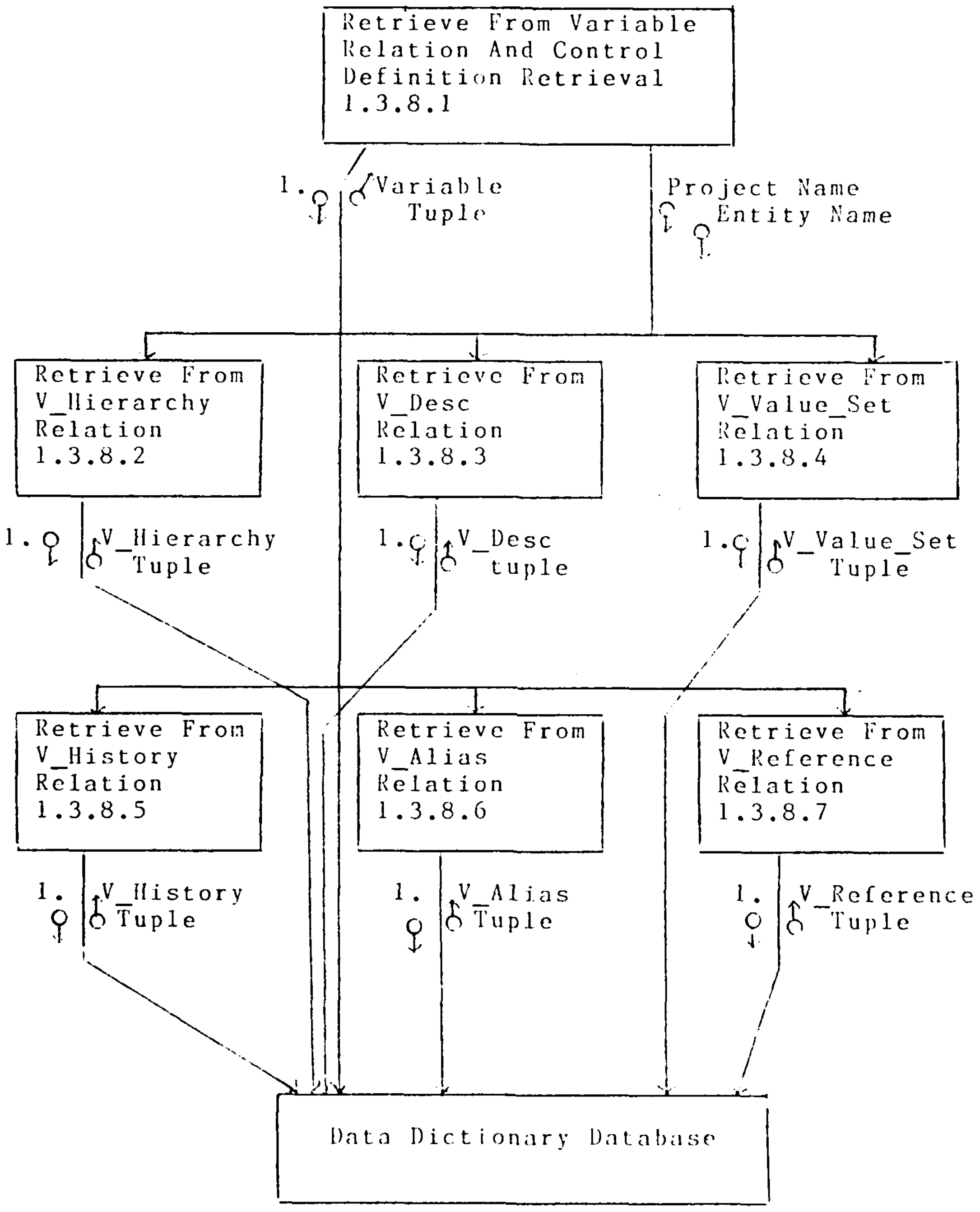
1. - Qualifier



Retrieve From Module Relation And Control Definition Retrieval 1.3.7.1
1.
Module Tuple
Project Name
Entity Name
Retrieve From M_Desc Relation 1.3.7.2
Retrieve From M_Alg Relation 1.3.7.3
Retrieve From M_Call Relation 1.3.7.4
1.
M_Desc Tuple
1.
M_Alg Tuple
1.
M_Call Tuple
Retrieve From M_Pass Relation 1.3.7.5
Retrieve From M_History Relation 1.3.7.6
Retrieve From M_Reference Relation 1.3.7.7
1.
M_Pass Tuple
1.
M_History Tuple
1.
M_Reference Tuple
Retrieve From M_Alias Relation 1.3.7.8
Retrieve From Module_IO Relation 1.3.7.9
1.
M_Alias Tuple
1.
Module_IO Tuple
Data Dictionary Database

Figure 2-20. Retrieve Code Module Definition (1.3.7).

1. - Qualifier



Retrieve From Variable Relation And Control Definition Retrieval 1.3.8.1
1.
Variable Tuple
Project Name
Entity Name
Retrieve From V_Hierarchy Relation 1.3.8.2
Retrieve From V_Desc Relation 1.3.8.3
Retrieve From V_Value_Set Relation 1.3.8.4
1.
V_Hierarchy Tuple
1.
V_Desc tuple
1.
V_Value_Set Tuple
Retrieve From V_History Relation 1.3.8.5
Retrieve From V_Alias Relation 1.3.8.6
Retrieve From V_Reference Relation 1.3.8.7
1.
V_History Tuple
1.
V_Alias Tuple
1.
V_Reference Tuple
Data Dictionary Database

Figure 2-21. Retrieve SC Parameter Definition (1.3.8).

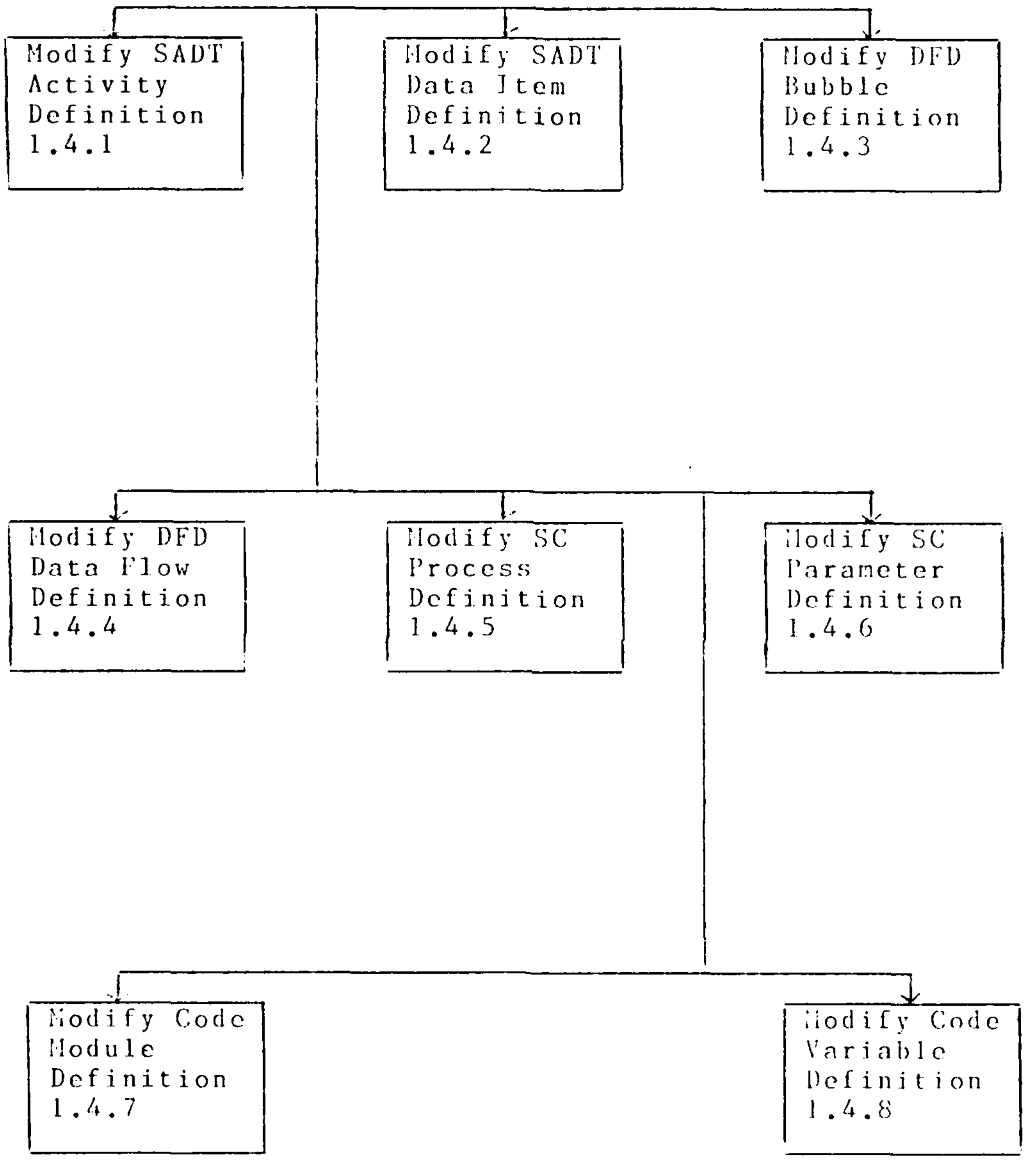
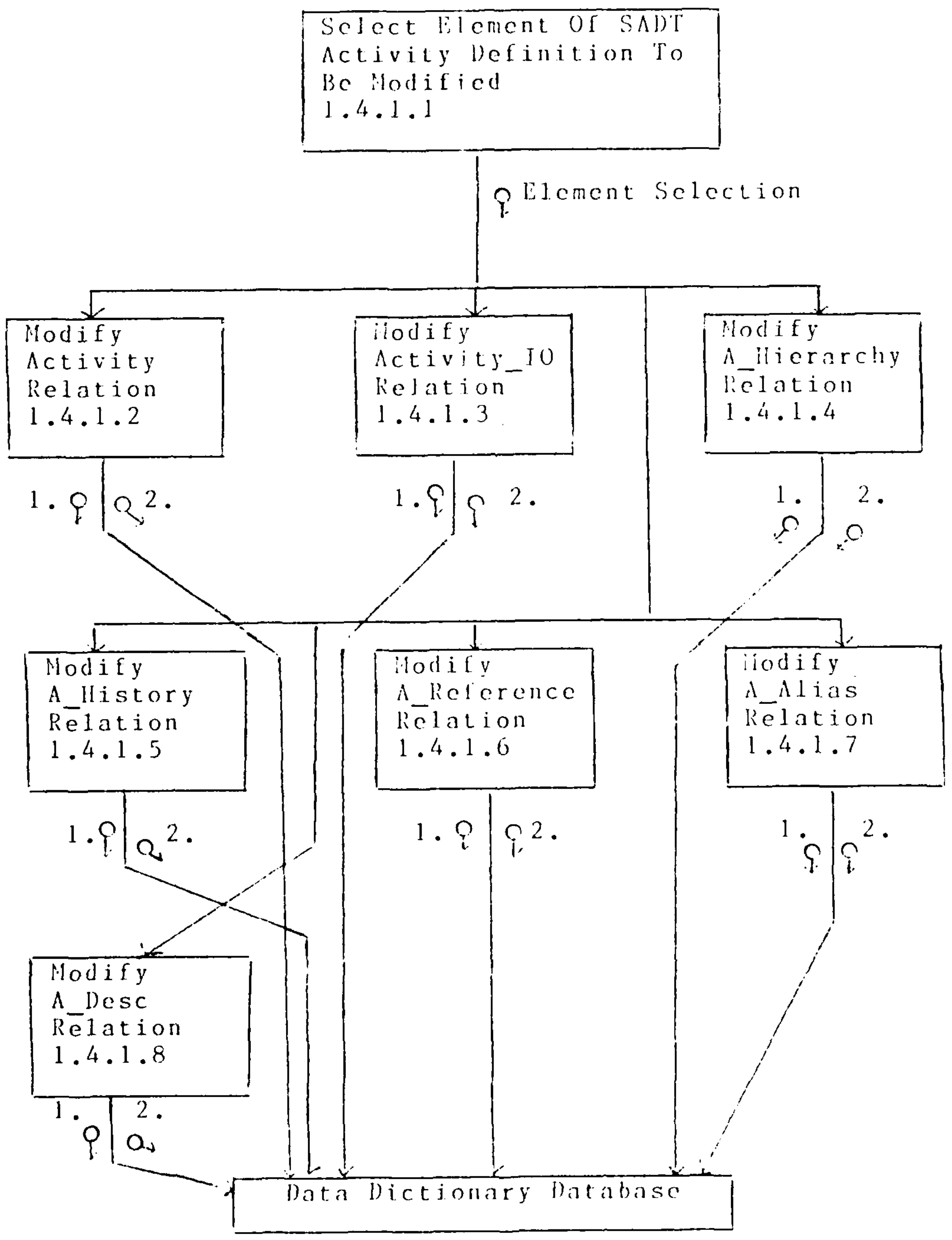

Modify SADT Activity Definition 1.4.1
Modify SADT Data Item Definition 1.4.2
Modify DFD Bubble Definition 1.4.3
Modify DFD Data Flow Definition 1.4.4
Modify SC Process Definition 1.4.5
Modify SC Parameter Definition 1.4.6
Modify Code Module Definition 1.4.7
Modify Code Variable Definition 1.4.8

Figure 2-22. Modify Entity Definition (1.4).

1. - Qualifier
2. - Tuple Modification



Select Element Of SADT Activity Definition To Be Modified 1.4.1.1
Element Selection
Modify Activity Relation 1.4.1.2
Modify Activity_IO Relation 1.4.1.3
Modify A_Hierarchy Relation 1.4.1.4
Modify A_History Relation 1.4.1.5
Modify A_Reference Relation 1.4.1.6
Modify A_Alias Relation 1.4.1.7
Modify A_Desc Relation 1.4.1.8
1.
2.
Data Dictionary Database

Figure 2-23. Modify SADT Activity Definition (1.4.1).

1. Qualifier
2. - Tuple Modification

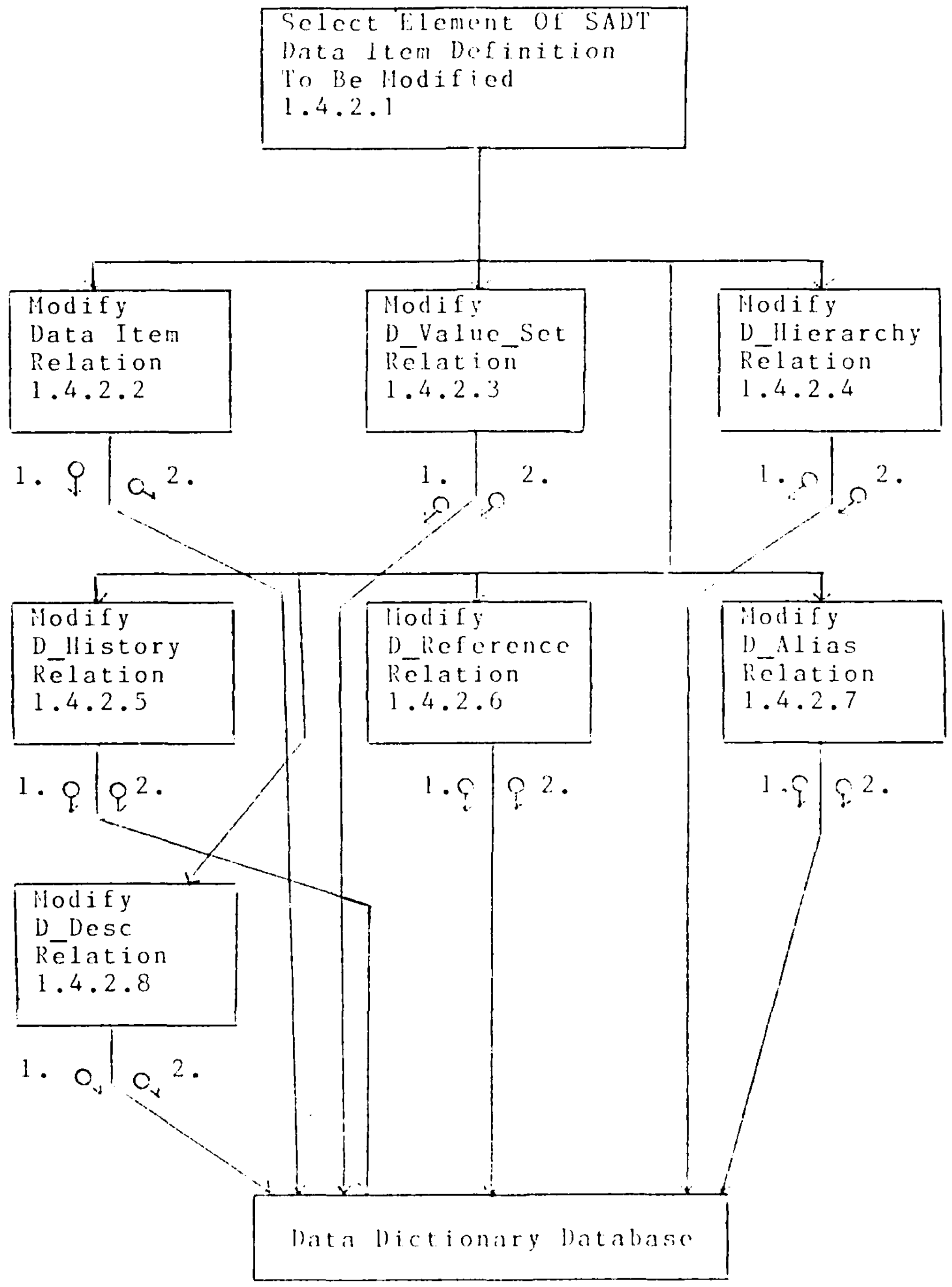
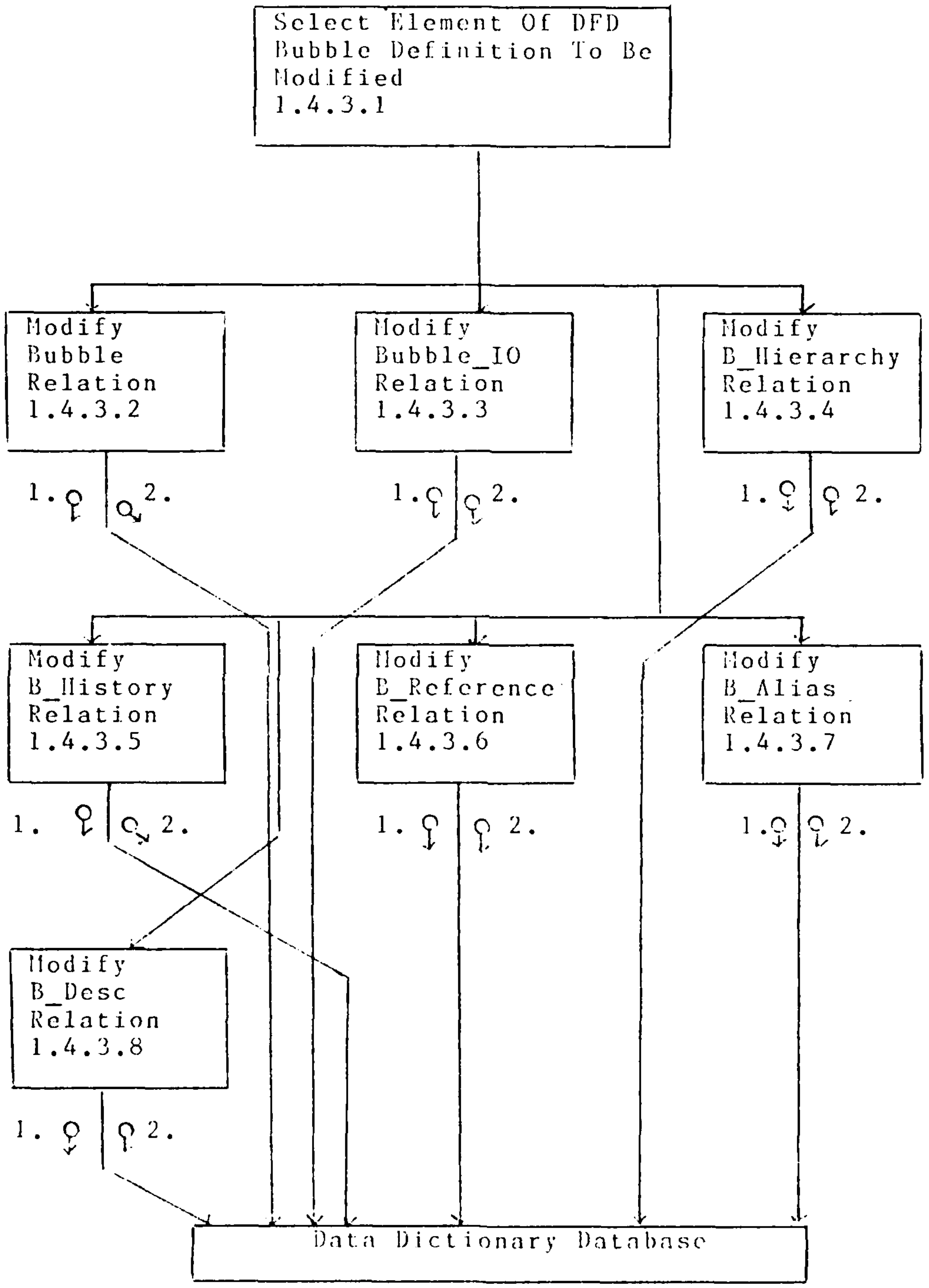

Select Element Of SADT Data Item Definition To Be Modified 1.4.2.1
Modify Data Item Relation 1.4.2.2
Modify D_Value_Set Relation 1.4.2.3
Modify D_Hierarchy Relation 1.4.2.4
Modify D_History Relation 1.4.2.5
Modify D_Reference Relation 1.4.2.6
Modify D_Alias Relation 1.4.2.7
Modify D_Desc Relation 1.4.2.8
1.
2.
Data Dictionary Database

Figure 2-24. Retrieve SADT Data Item Definition (1.4.2).

1. Qualifier
2. Tuple Modification



Select Element Of DFD Bubble Definition To Be Modified 1.4.3.1
Modify Bubble Relation 1.4.3.2
Modify Bubble_IO Relation 1.4.3.3
Modify B_Hierarchy Relation 1.4.3.4
Modify B_History Relation 1.4.3.5
Modify B_Reference Relation 1.4.3.6
Modify B_Alias Relation 1.4.3.7
Modify B_Desc Relation 1.4.3.8
1.
2.
Data Dictionary Database

Figure 2-25. Modify DFD Bubble Definition (1.4.3).

1. - Qualifier
2. - Tuple Modification

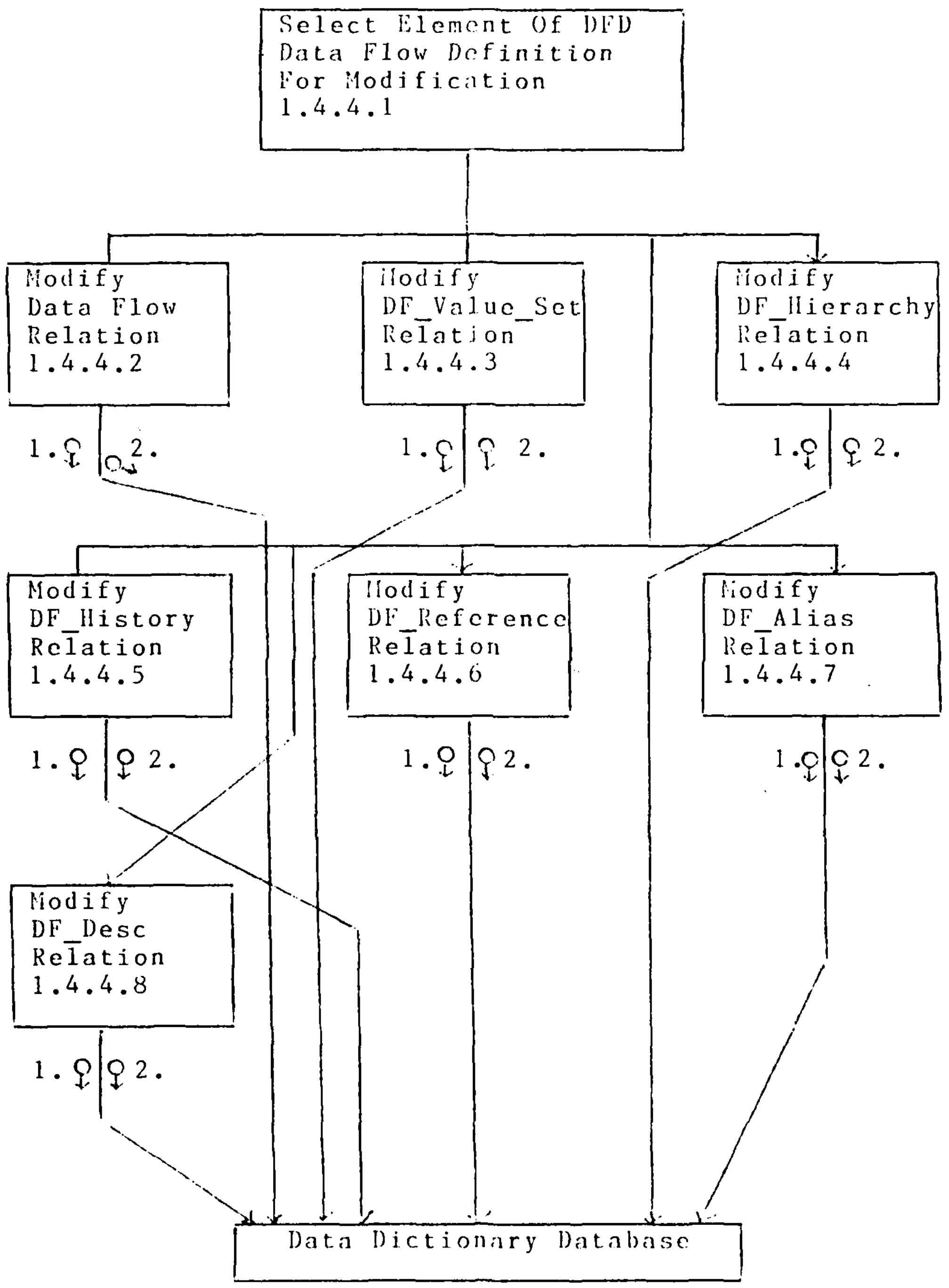

Select Element Of DFD Data Flow Definition For Modification 1.4.4.1
Modify Data Flow Relation 1.4.4.2
Modify DF_Value_Set Relation 1.4.4.3
Modify DF_Hierarchy Relation 1.4.4.4
Modify DF_History Relation 1.4.4.5
Modify DF_Reference Relation 1.4.4.6
Modify DF_Alias Relation 1.4.4.7
Modify DF_Desc Relation 1.4.4.8
1.
2.
Data Dictionary Database

Figure 2-26. Modify DFD Data Flow Definition (1.4.4).

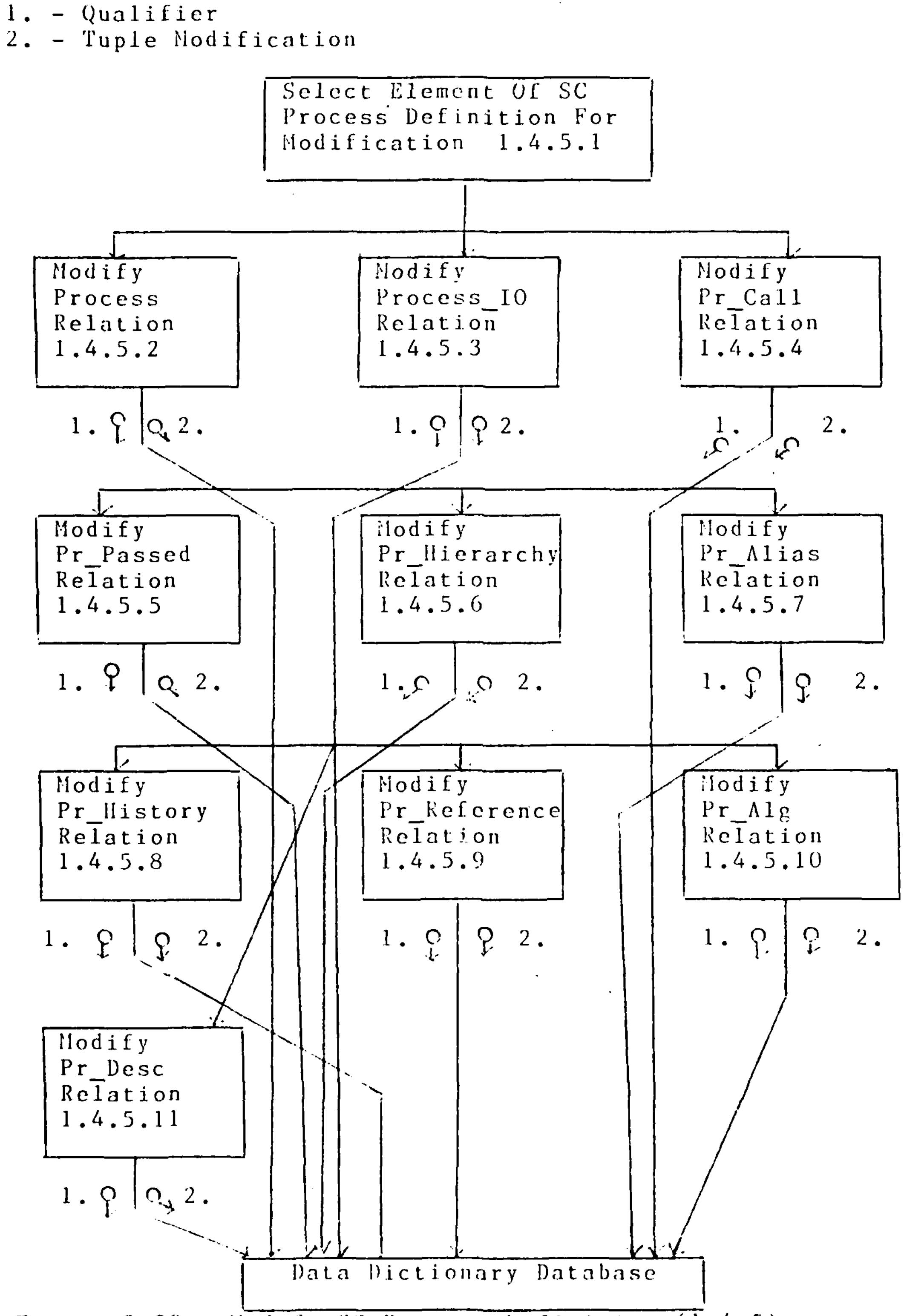

1. - Qualifier
2. - Tuple Modification
Select Element Of SC Process Definition For Modification 1.4.5.1
Modify Process Relation 1.4.5.2
Modify Process_IO Relation 1.4.5.3
Modify Pr_Call Relation 1.4.5.4
Modify Pr_Passed Relation 1.4.5.5
Modify Pr_Hierarchy Relation 1.4.5.6
Modify Pr_Alias Relation 1.4.5.7
Modify Pr_History Relation 1.4.5.8
Modify Pr_Reference Relation 1.4.5.9
Modify Pr_Alg Relation 1.4.5.10
Modify Pr_Desc Relation 1.4.5.11
1.
2.
Data Dictionary Database

Figure 2-28. Modify SC Process Definition (1.4.5).

1. - Qualifier
2. - Tuple Modification

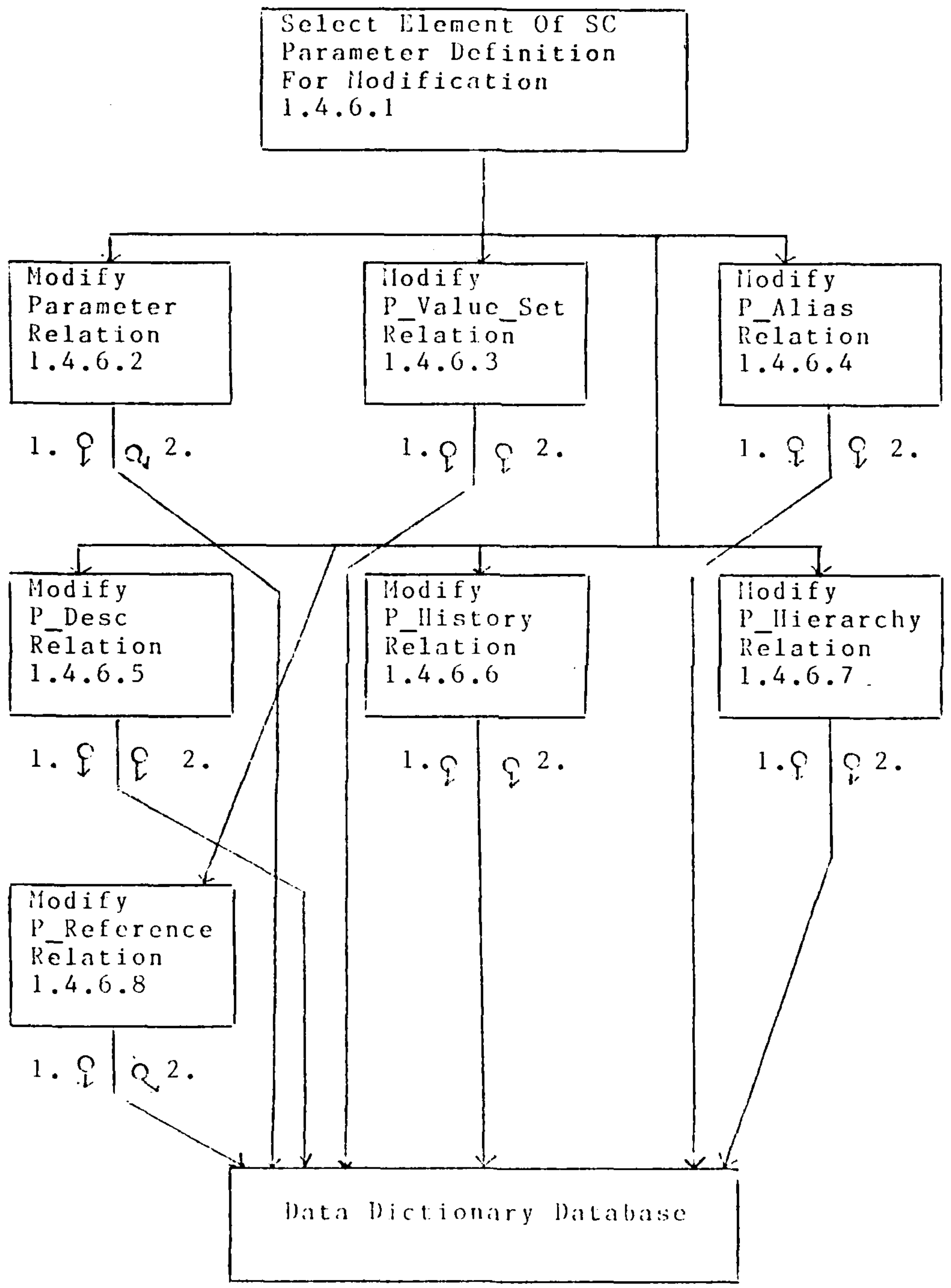

Select Element Of SC Parameter Definition For Modification 1.4.6.1
Modify Parameter Relation 1.4.6.2
Modify P_Value_Set Relation 1.4.6.3
Modify P_Alias Relation 1.4.6.4
Modify P_Desc Relation 1.4.6.5
Modify P_History Relation 1.4.6.6
Modify P_Hierarchy Relation 1.4.6.7
Modify P_Reference Relation 1.4.6.8
1.
2.
Data Dictionary Database

Figure 2-28. Modify SC Parameter Definition (1.4.6).

1. - Qualifier
2. - Tuple Modification

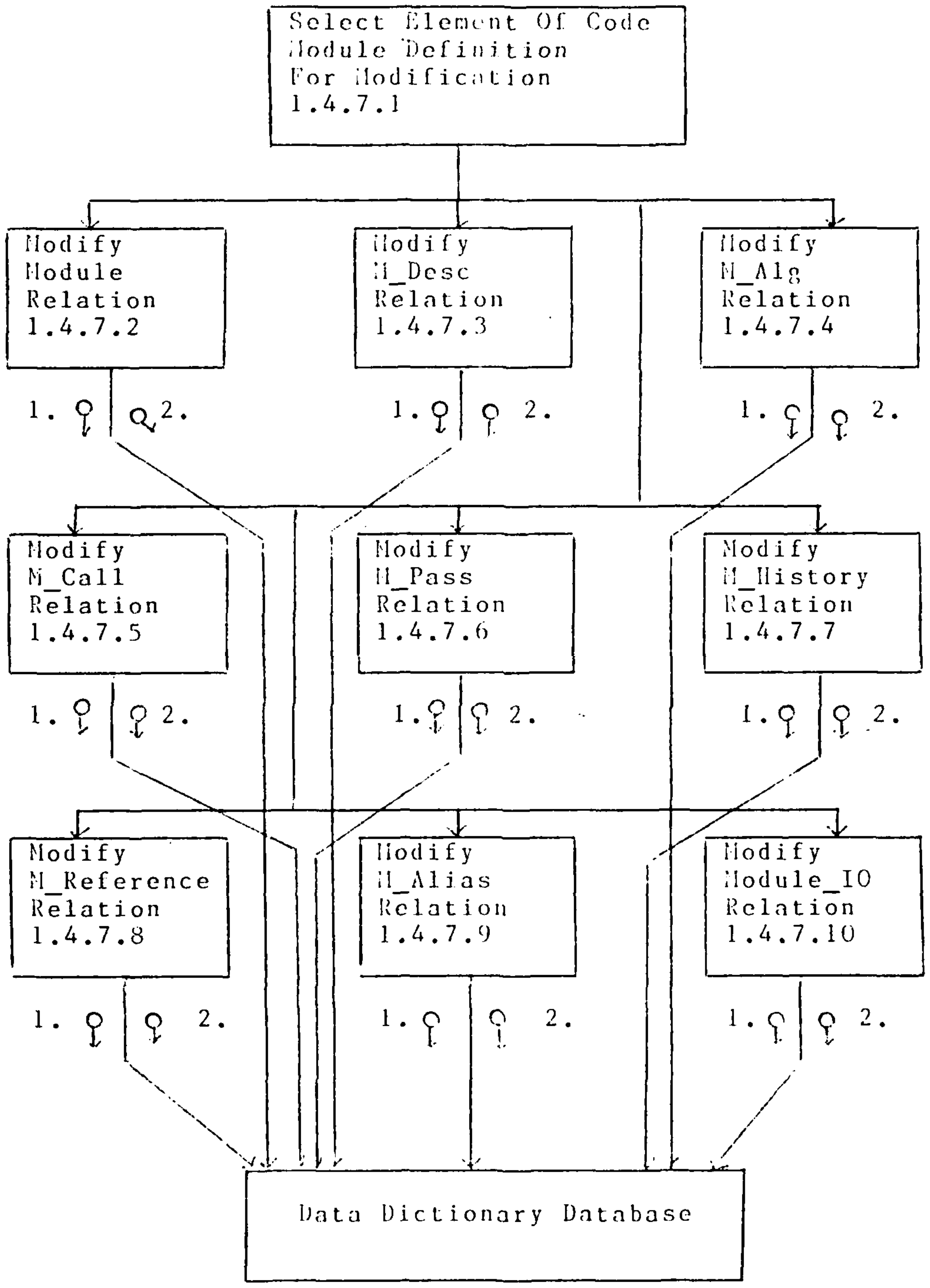

Select Element Of Code Module Definition For Modification 1.4.7.1
Modify Module Relation 1.4.7.2
Modify M_Desc Relation 1.4.7.3
Modify M_Alg Relation 1.4.7.4
Modify M_Call Relation 1.4.7.5
Modify M_Pass Relation 1.4.7.6
Modify M_History Relation 1.4.7.7
Modify M_Reference Relation 1.4.7.8
Modify M_Alias Relation 1.4.7.9
Modify Module_IO Relation 1.4.7.10
1.
2.
Data Dictionary Database

Figure 2-29. Modify Code Module Definition (1.4.7).

1. - Qualifier
2. - Tuple Modification

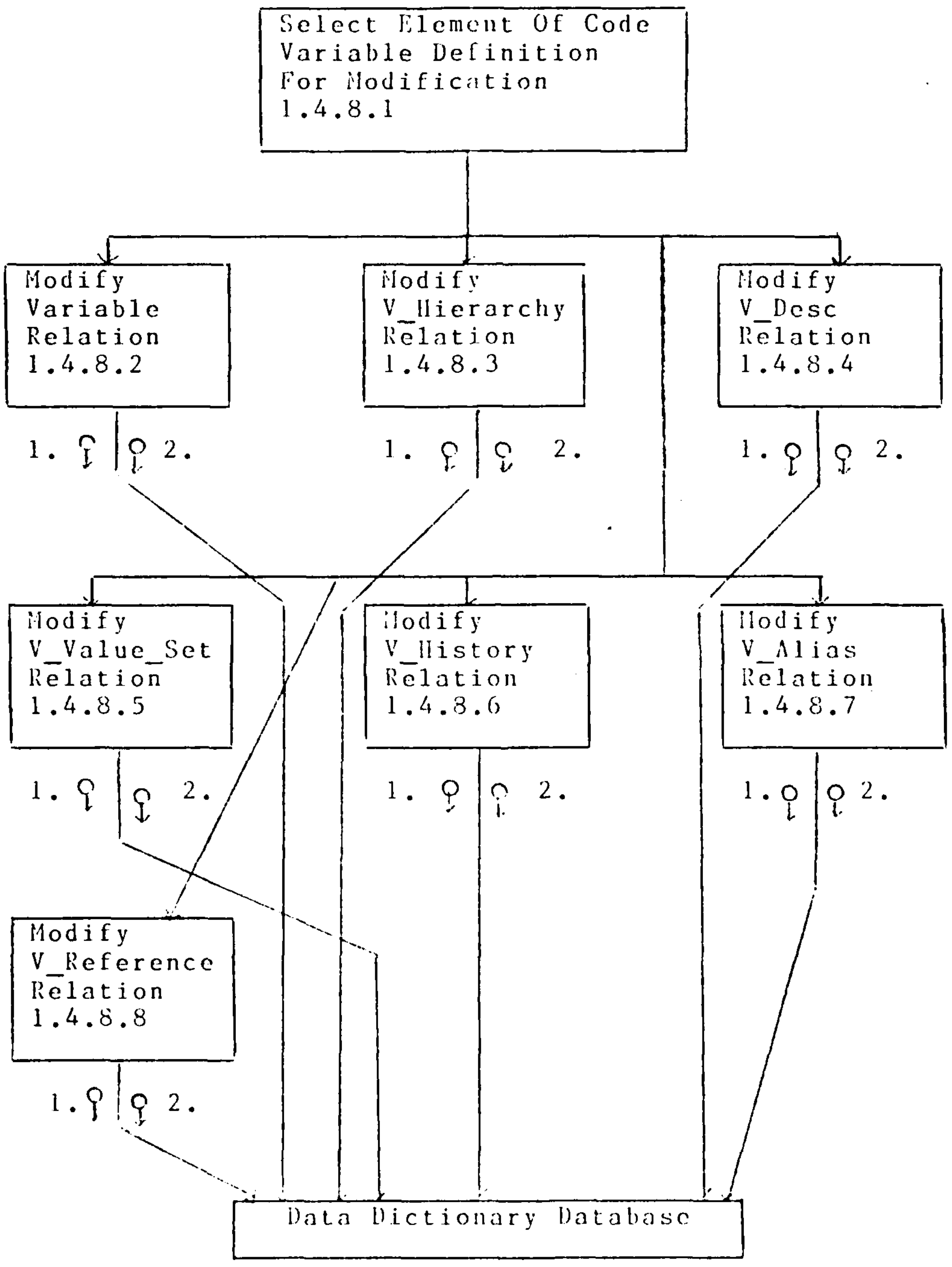

Select Element Of Code Variable Definition For Modification 1.4.8.1
Modify Variable Relation 1.4.8.2
Modify V_Hierarchy Relation 1.4.8.3
Modify V_Desc Relation 1.4.8.4
Modify V_Value_Set Relation 1.4.8.5
Modify V_History Relation 1.4.8.6
Modify V_Alias Relation 1.4.8.7
Modify V_Reference Relation 1.4.8.8
1.
2.
Data Dictionary Database

Figure 2-30. Modify Code Variable Definition (1.4.8).

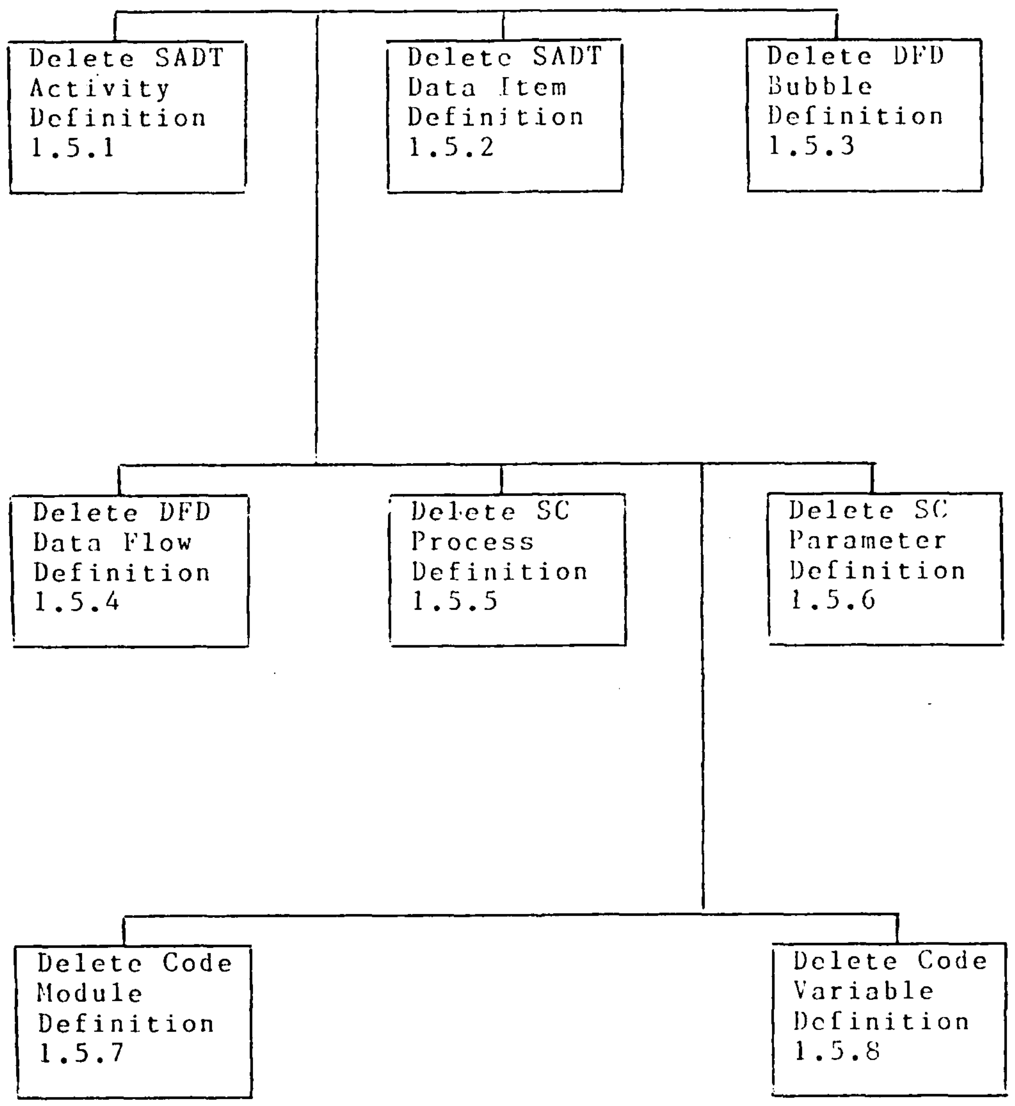
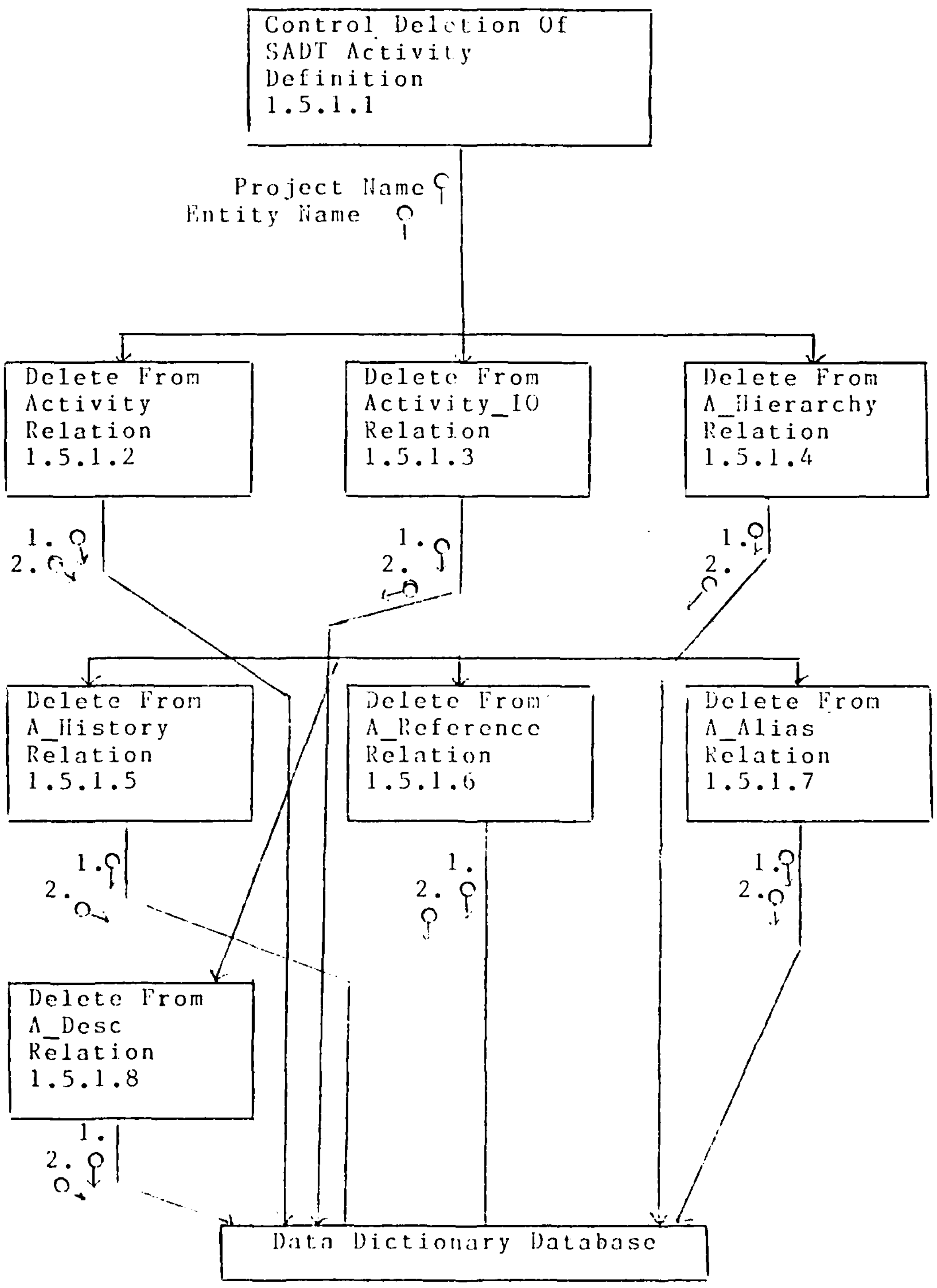

Delete SADT Activity Definition 1.5.1
Delete SADT Data Item Definition 1.5.2
Delete DFD Bubble Definition 1.5.3
Delete DFD Data Flow Definition 1.5.4
Delete SC Process Definition 1.5.5
Delete SC Parameter Definition 1.5.6
Delete Code Module Definition 1.5.7
Delete Code Variable Definition 1.5.8

Figure 2-31. Delete Entity Definition (1.5).

1. - Qualifier
2. - Delete Command



Control Deletion Of SADT Activity Definition 1.5.1.1
Project Name
Entity Name
Delete From Activity Relation 1.5.1.2
Delete From Activity_IO Relation 1.5.1.3
Delete From A_Hierarchy Relation 1.5.1.4
Delete From A_History Relation 1.5.1.5
Delete From A_Reference Relation 1.5.1.6
Delete From A_Alias Relation 1.5.1.7
Delete From A_Desc Relation 1.5.1.8
1.
2.
Data Dictionary Database

Figure 2-32. Delete SADT Activity Definition (1.5.1).

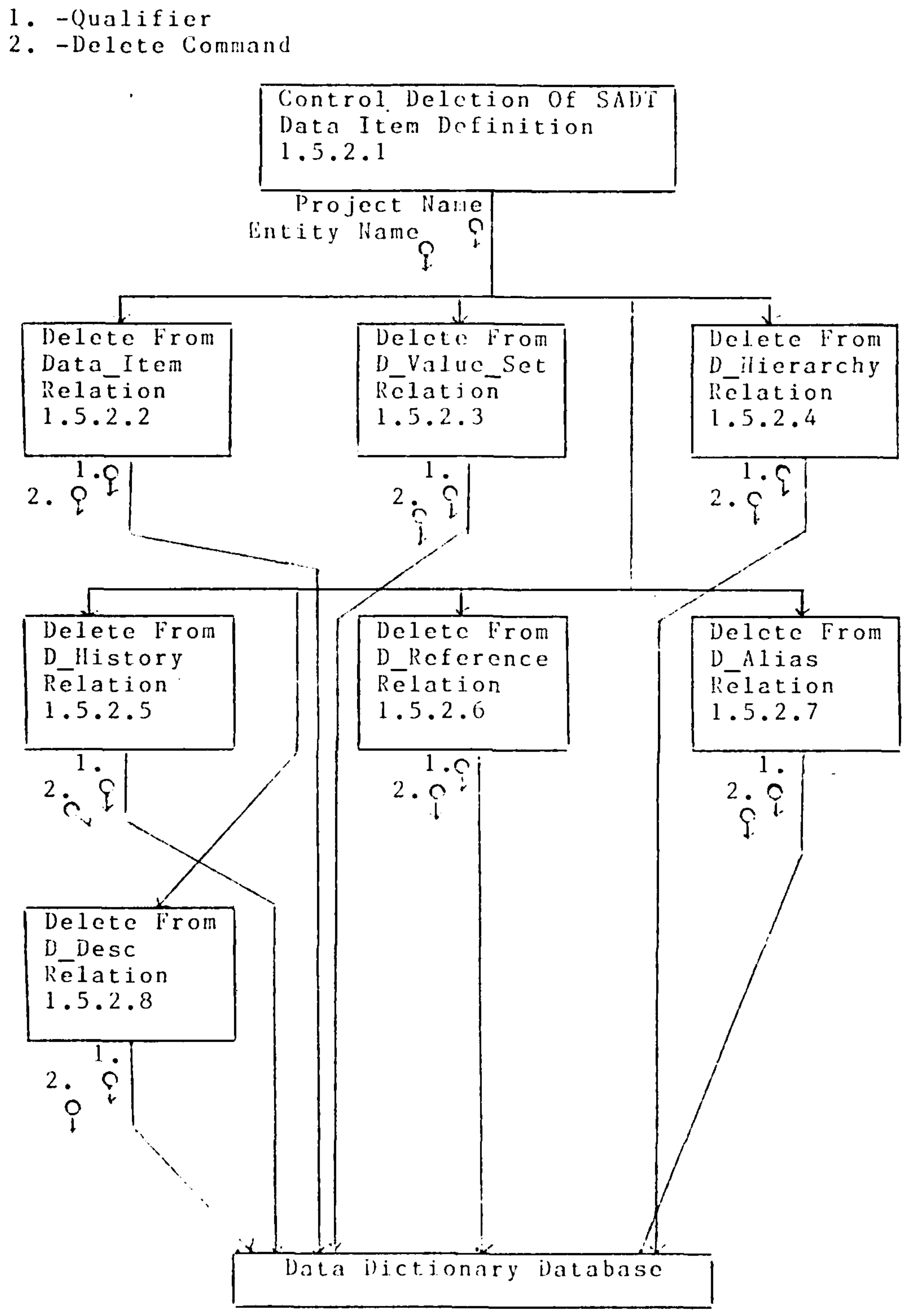
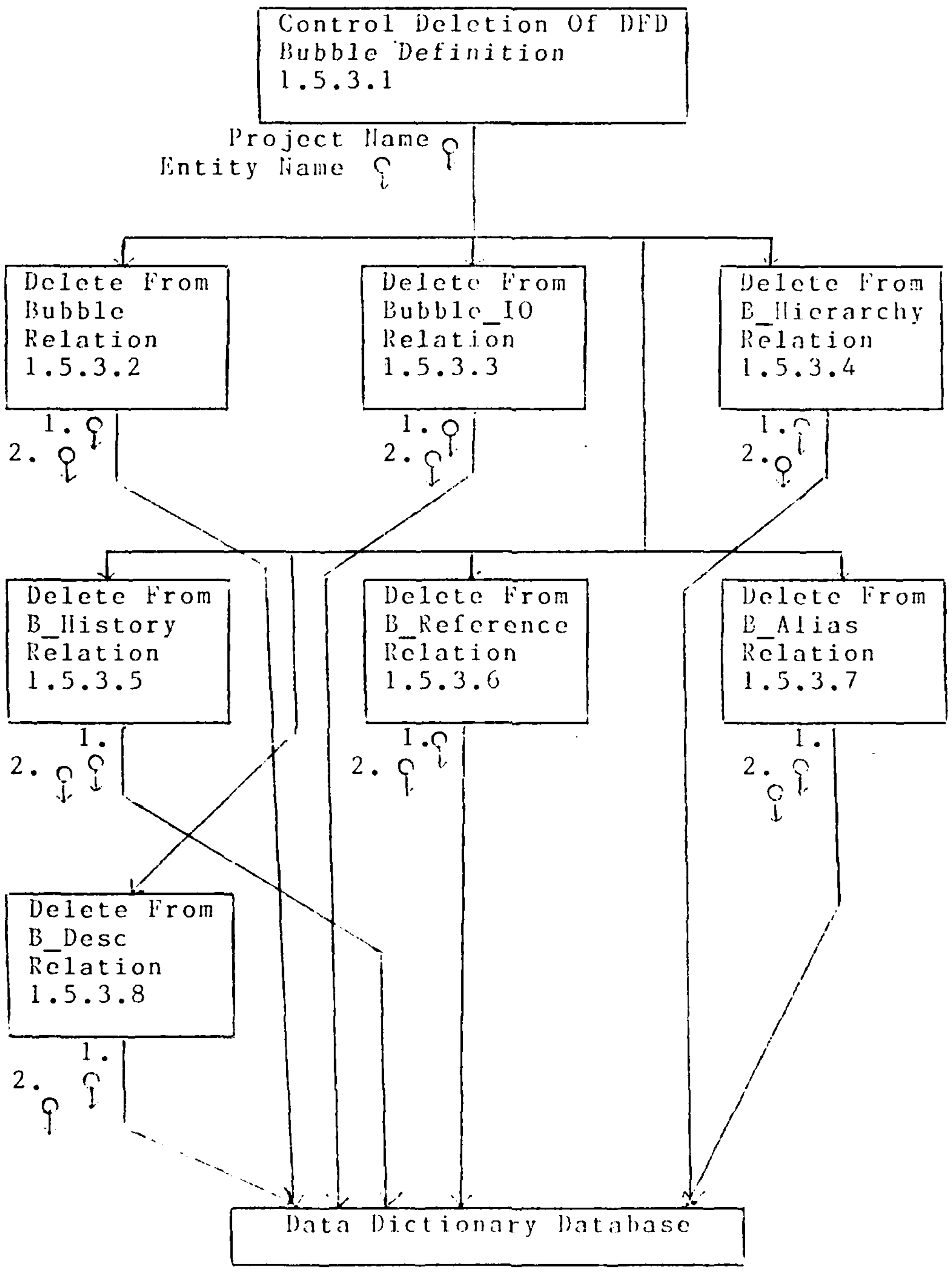

1. -Qualifier
2. -Delete Command
Control Deletion Of SADT Data Item Definition 1.5.2.1
Project Name
Entity Name
Delete From Data_Item Relation 1.5.2.2
Delete From D_Value_Set Relation 1.5.2.3
Delete From D_Hierarchy Relation 1.5.2.4
Delete From D_History Relation 1.5.2.5
Delete From D_Reference Relation 1.5.2.6
Delete From D_Alias Relation 1.5.2.7
Delete From D_Desc Relation 1.5.2.8
1.
2.
Data Dictionary Database

Figure 2-34. Delete SADT Data Item Definition (1.5.2).

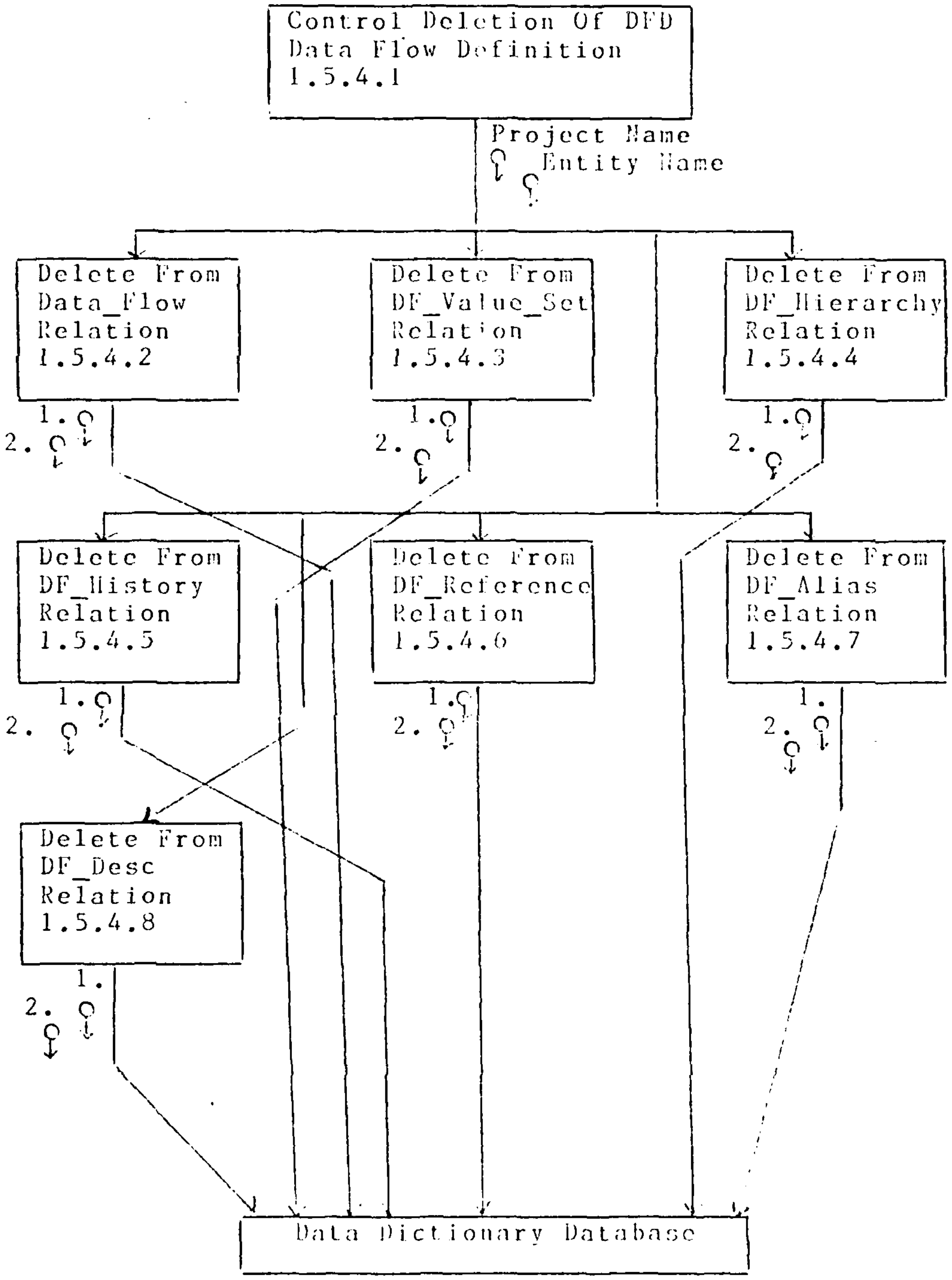
1. Qualifier
2. Delete Command



Control Deletion Of DFD Bubble Definition 1.5.3.1
Project Name
Entity Name
Delete From Bubble Relation 1.5.3.2
Delete From Bubble_IO Relation 1.5.3.3
Delete From B_Hierarchy Relation 1.5.3.4
Delete From B_History Relation 1.5.3.5
Delete From B_Reference Relation 1.5.3.6
Delete From B_Alias Relation 1.5.3.7
Delete From B_Desc Relation 1.5.3.8
1.
2.
Data Dictionary Database

Figure 2-34. Delete DFD Bubble Definition (1.5.3).

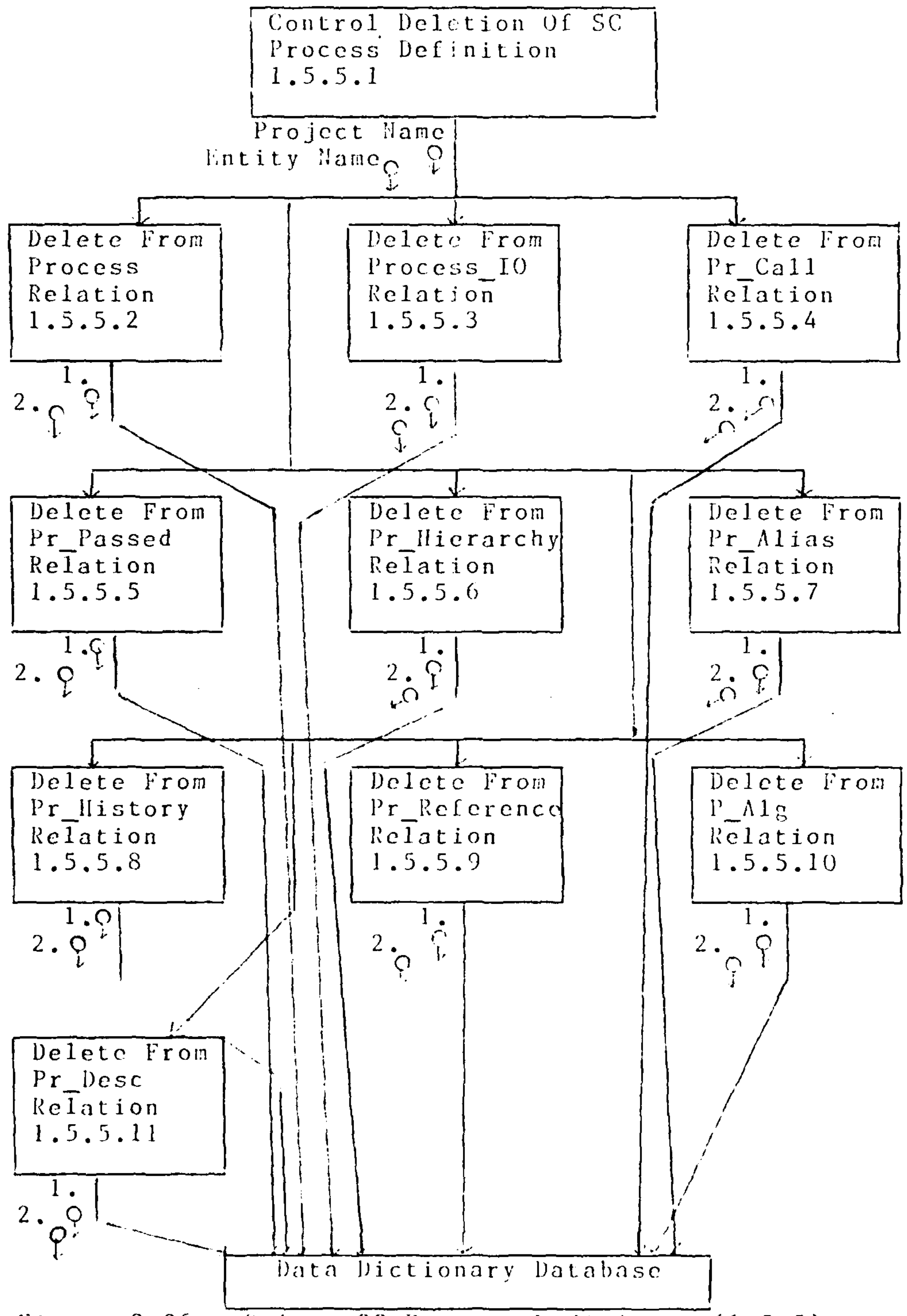
1. Qualifier
2. Delete Command



Control Deletion Of DFD Data Flow Definition 1.5.4.1
Project Name
Entity Name
Delete From Data_Flow Relation 1.5.4.2
Delete From DF_Value_Set Relation 1.5.4.3
Delete From DF_Hierarchy Relation 1.5.4.4
Delete From DF_History Relation 1.5.4.5
Delete From DF_Reference Relation 1.5.4.6
Delete From DF_Alias Relation 1.5.4.7
Delete From DF_Desc Relation 1.5.4.8
1.
2.
Data Dictionary Database

Figure 2-35. Delete From DFD Data Flow Definition (1.5.4).

1. *Qualifier*
2. Delete Command



Control Deletion Of SC Process Definition 1.5.5.1
Project Name
Entity Name
Delete From Process Relation 1.5.5.2
Delete From Process_IO Relation 1.5.5.3
Delete From Pr_Call Relation 1.5.5.4
Delete From Pr_Passed Relation 1.5.5.5
Delete From Pr_Hierarchy Relation 1.5.5.6
Delete From Pr_Alias Relation 1.5.5.7
Delete From Pr_History Relation 1.5.5.8
Delete From Pr_Reference Relation 1.5.5.9
Delete From P_Alg Relation 1.5.5.10
Delete From Pr_Desc Relation 1.5.5.11
1.
2.
Data Dictionary Database

Figure 2-36. Delete SC Process Definition (1.5.5).

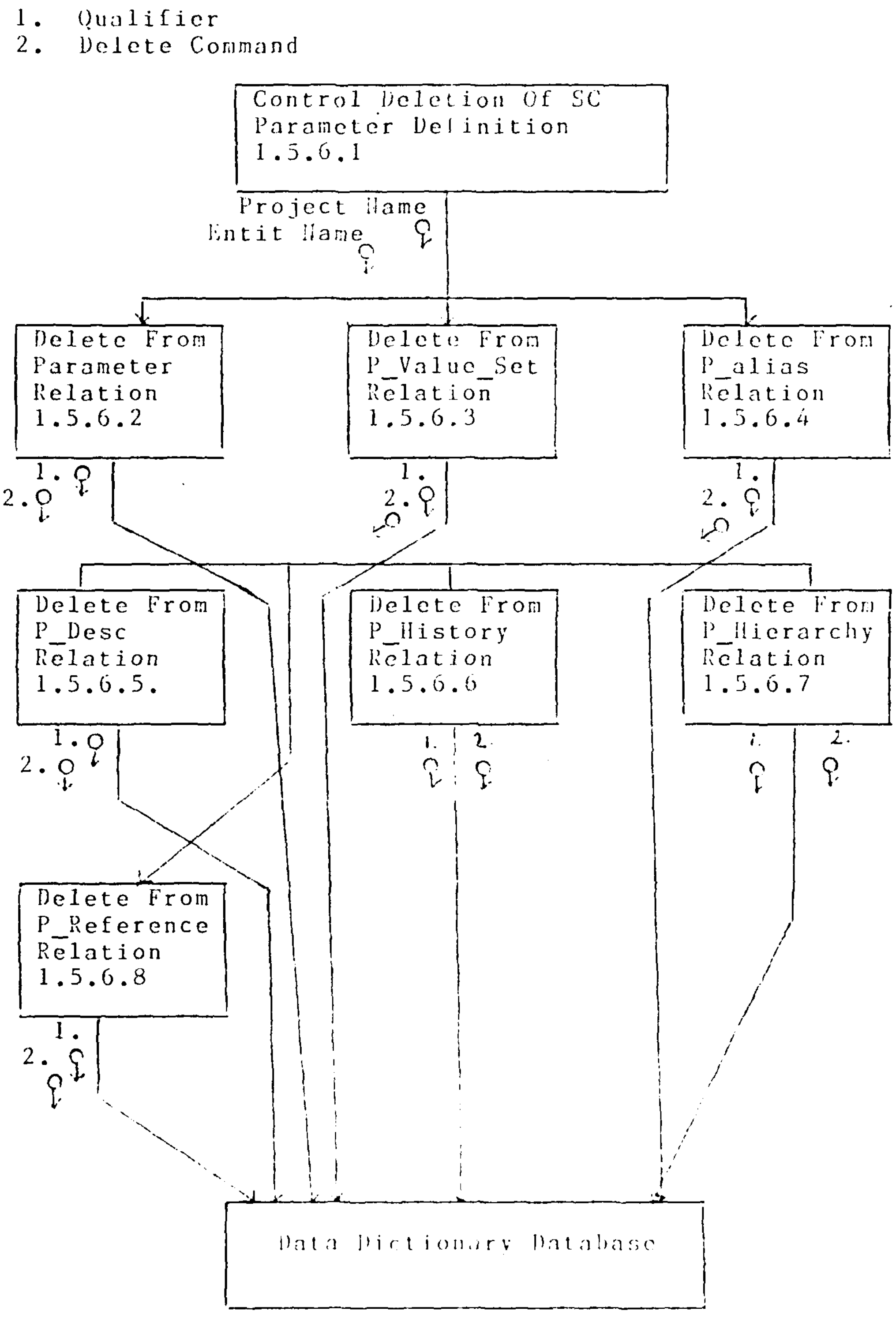
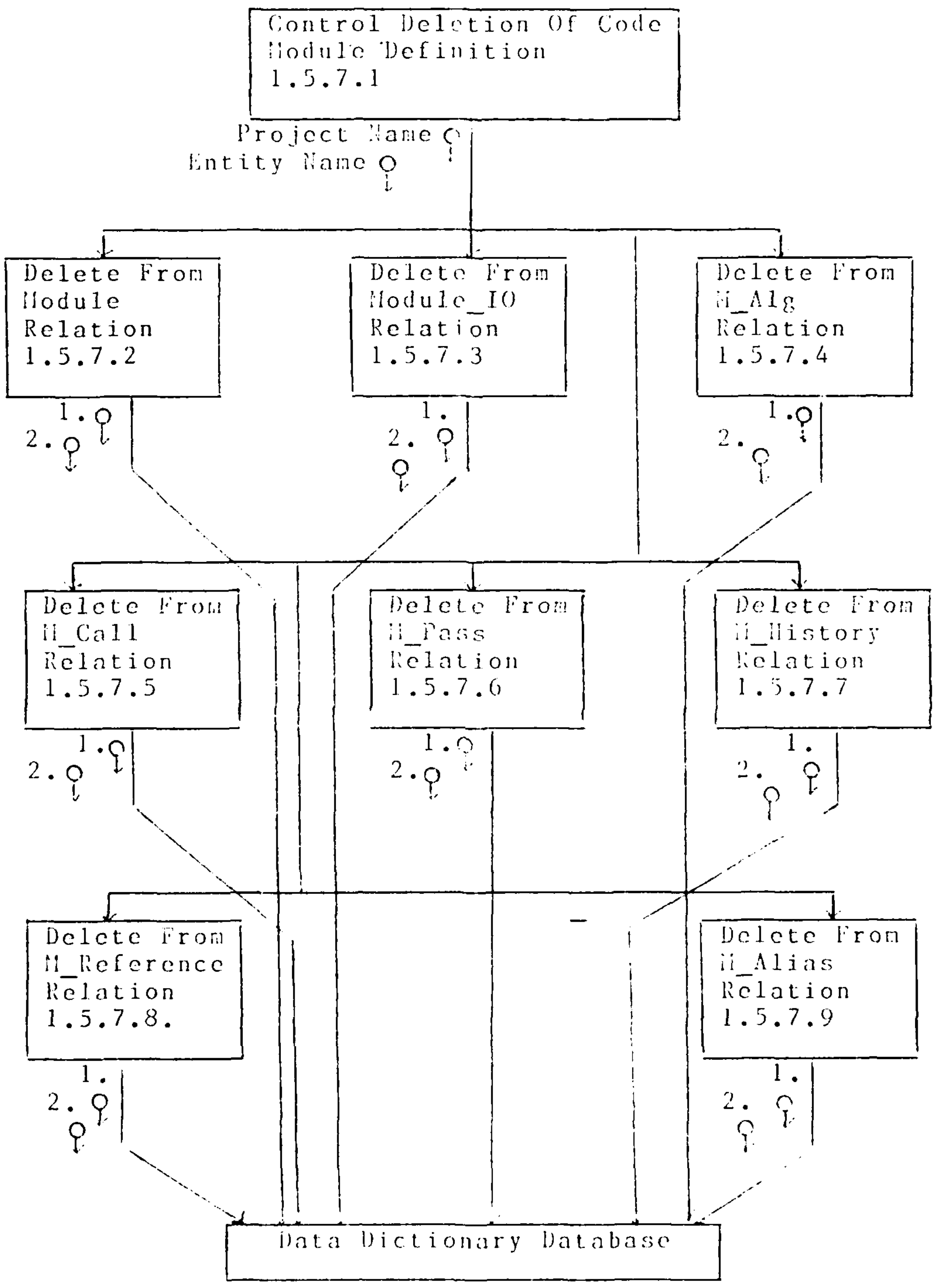

1. Qualifier
2. Delete Command
Control Deletion Of SC Parameter Definition 1.5.6.1
Project Name
Entit Name
Delete From Parameter Relation 1.5.6.2
Delete From P_Value_Set Relation 1.5.6.3
Delete From P_alias Relation 1.5.6.4
Delete From P_Desc Relation 1.5.6.5.
Delete From P_History Relation 1.5.6.6
Delete From P_Hierarchy Relation 1.5.6.7
Delete From P_Reference Relation 1.5.6.8
1.
2.
Data Dictionary Database

Figure 2-37. Delete SC Parameter Definition (1.5.6).

1. Qualifier
2. Delete Command



Control Deletion Of Code Module Definition 1.5.7.1
Project Name
Entity Name
Delete From Module Relation 1.5.7.2
Delete From Module_IO Relation 1.5.7.3
Delete From M_Alg Relation 1.5.7.4
Delete From M_Call Relation 1.5.7.5
Delete From M_Pass Relation 1.5.7.6
Delete From M_History Relation 1.5.7.7
Delete From M_Reference Relation 1.5.7.8.
Delete From M_Alias Relation 1.5.7.9
1.
2.
Data Dictionary Database

Figure 2-38. Delete Code Module Definition (1.5.7.).

1. Qualifier
2. Delete Command

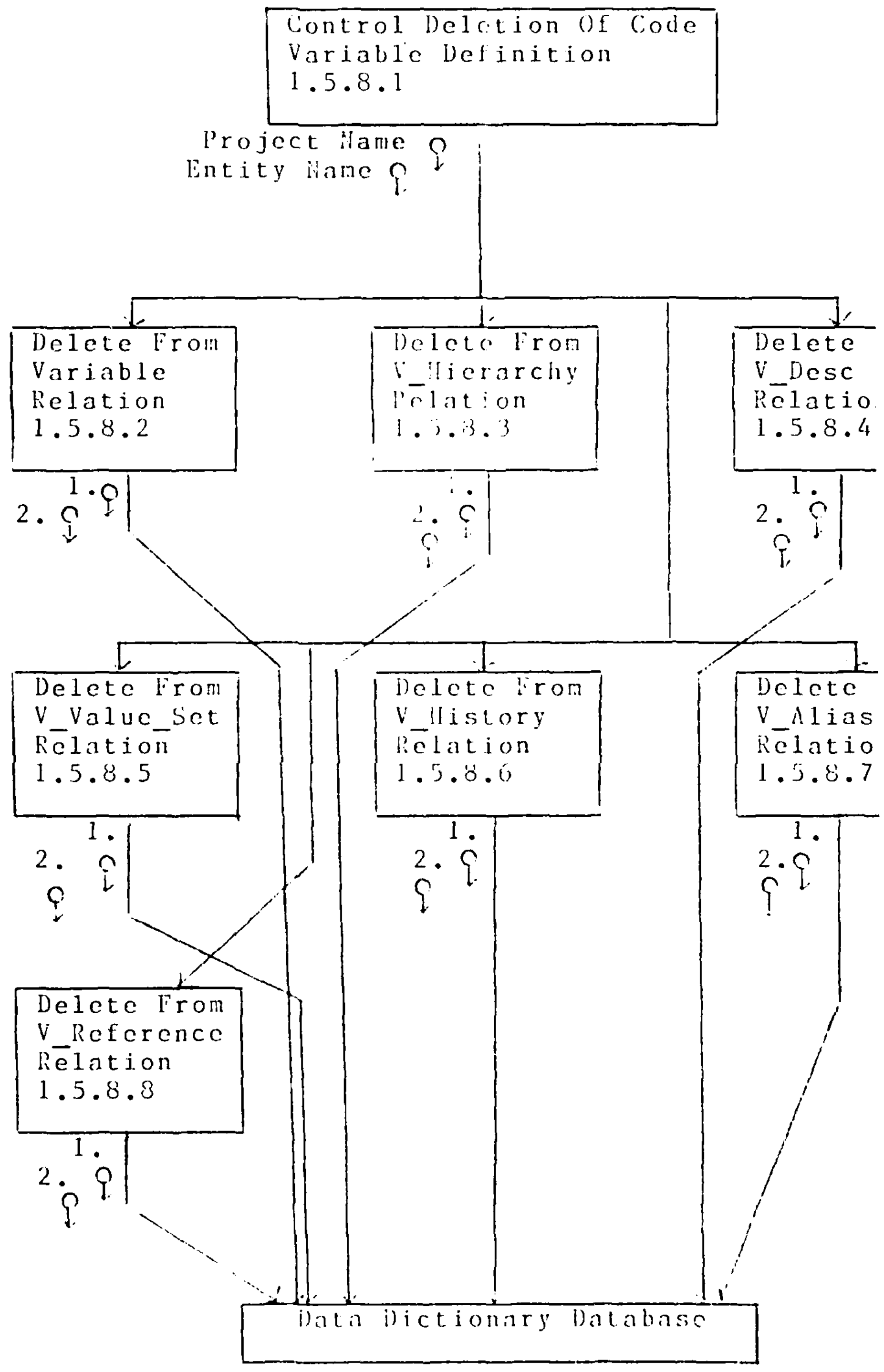

Control Deletion Of Code Variable Definition 1.5.8.1
Project Name
Entity Name
Delete From Variable Relation 1.5.8.2
Delete From V_Hierarchy Relation 1.5.8.3
Delete V_Desc Relatio 1.5.8.4
Delete From V_Value_Set Relation 1.5.8.5
Delete From V_History Relation 1.5.8.6
Delete V_Alias Relatio 1.5.8.7
Delete From V_Reference Relation 1.5.8.8
1.
2.
Data Dictionary Database

Figure 2-39. Delete SC Variable Definition (1.5.8).

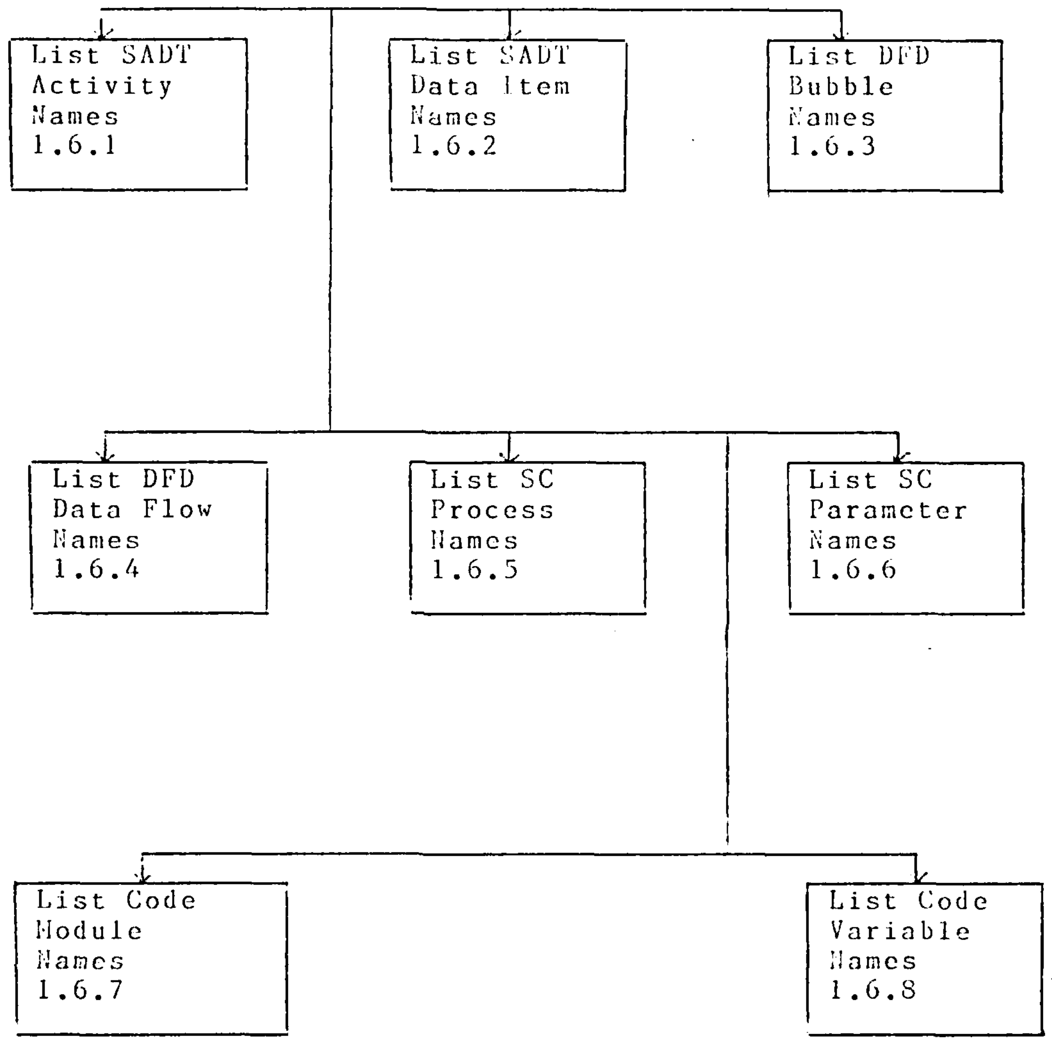

List SADT Activity Names 1.6.1
List SADT Data Item Names 1.6.2
List DFD Bubble Names 1.6.3
List DFD Data Flow Names 1.6.4
List SC Process Names 1.6.5
List SC Parameter Names 1.6.6
List Code Module Names 1.6.7
List Code Variable Names 1.6.8

Figure 2-40. List Entity Names (1.6).

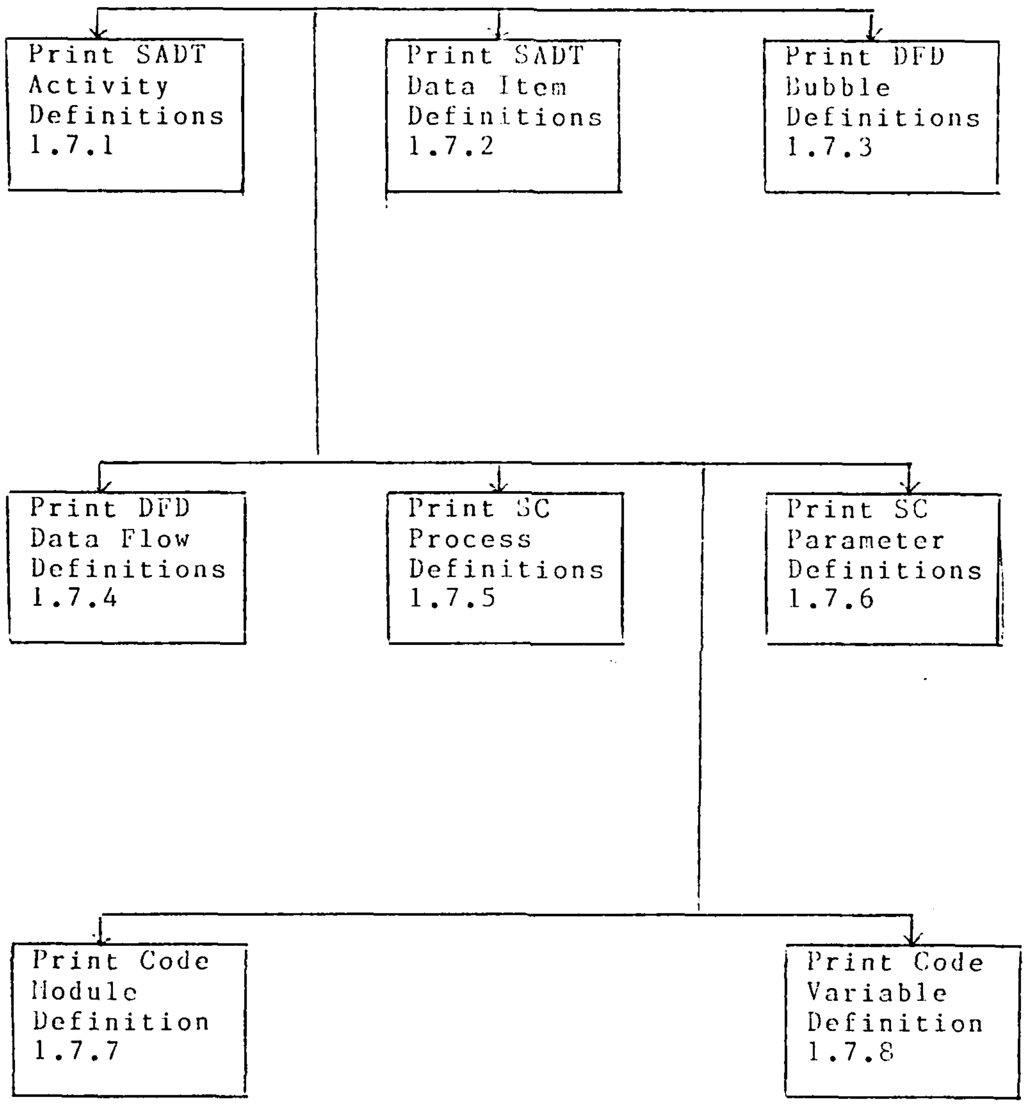

Print SADT Activity Definitions 1.7.1
Print SADT Data Item Definitions 1.7.2
Print DFD Bubble Definitions 1.7.3
Print DFD Data Flow Definitions 1.7.4
Print SC Process Definitions 1.7.5
Print SC Parameter Definitions 1.7.6
Print Code Module Definition 1.7.7
Print Code Variable Definition 1.7.8

Figure 2-41. Print Entity Definitions (1.7).

Appendix C: User's Guide For The Data Dictionary Generation Tool

This appendix provides information on how to use the current implementation of the data dictionary generation tool.

General Information: Data dictionary information is entered and retrieved from this tool in two formats; a data definition and a action definition. A data definition describes the data or information used by the action components of a program. Included under the category of data definitions are files and hardware accessed and used by a program. Action definitions apply to those components of a program which use and manipulate information or data in performing a functions. Procedures, functions, activities, processes etc. are described in action definitions.

Introduction: A series of menu driven displays allow the user to select the particular dictionary operation he or she desires to perform. The user will also, by means of these menu driven displays, select either a action or data definition to manipulate. The user will also select by means of these display menus the type of software representation they wish to enter or retrieve dictionary information for. This tool supports four different types of software representations: SADTs, Data Flow Diagrams, Structure charts, and Code.

General Instructions: The user will communicate with the program in one of the following three methods.

1. You will be provided with a menu display of acceptable response to questions. You will indicate your response by entering the number corresponding to your selection and striking the carriage return on the keyboard.

2. You will be prompted to enter from the keyboard the answer to a question asked by the program. A guideline will be drawn on the screen to indicate the allowable length of your response. If your response exceeds the guideline, that portion of text will not be accepted by the program. When you have completed your response, strike the carriage return to inform the program that you are ready to continue.

3. You will be asked a yes or no question by the program. You will indicated your respone by typing a y for yes or a n for no and striking the carriage return.

The following section will provide more specific guidance on how to use the data dictionary tool to input definitions for the structure chart, SADT, data flow diagrams, and code software representations.

1. STRUCTURE CHARTS

A. Process Definition. You will be provided with the following prompts.

1. Enter project name. You will be given a project name which will identify your entries into the dictionary database. The tool will check this input. If you enter a project name not known to the tool, your input will be

rejected and you will prompted for this information again.

2. Enter process name. Type the name of the process you wish to define. Process names may not exceed 25 characters.

3. Enter process number. Type the number associated with this process on the structure chart.

4. Enter process description. Enter a text description of the process being defined. You may enter this description in the form of several lines of text each containing 60 characters. The guide line will give you a visual fix on the allowable line length. When you near the end of a line of text, simply strike the carriage return and continue entering text on the next line. When you have completed the description, strike the carriage return two consecutive times to inform the program that you are ready to continue.

5. Enter algorithm. The directions for this entry are identical to those for the description entry discussed above except that the information being entered describes the algorithm used by the process being defined.

6. Enter the name of an input or output parameter for this process. This prompt asks you to identify the internal input and output requirements of this process. This is not a request for the parameters which are passed to this process during a subroutine call. This information will not always be on the structure charts. You may find it necessary to create a new name for these parameter. This information is used to define the interface requirements for

this process.

7. You will be asked to identify the parmeter entered above as either an input or output of this process. Enter the number corresponding to the proper selection in the display menu.

8. You will be asked to classify this parmeter as either data or flag information. This is done by entering the number corresponding to the proper selection in the display menu.

9. You will be asked to enter a number which identifies the position this parmater has is a call to this process ie 1 for first, 2 for second, etc.

10. You will be asked if any more input or output parameter exist for this process. Items 6 through 9 will be repeated until you respond n to this query.

11. Does this process call any other processes? A response of y to this questions allows items 10 through 19 to appear. A response of n to this question shifts execution to item 20.

12. Enter the name of a process called by this process.

13. Are any parameters passed during this process call? A response of y to this question causes items 14 through 18 to be executed.

14. Enter the name of the parameter being passed. Enter the name of the parameter which is involved in the process call between the process being defined and the process

identified in item 12.

15. Indicate if the parmeter is data or flag information. Enter the number corresponding to the correct response listed in the display menu.

16. Indicate the order of this parameter in the process call. Enter 1 for first or 2 for second etc.

17. Identify this parameter as one of the following:

a. Passed from calling routine to called routine. Parameter X is passed from calling process A to called process B.

b. Parameter is returned by the called process to the calling process . Parameter Y is returned by the called process to the calling process.

c. Parameter is both passed from the calling process to the called process and returned by the called process to the calling nrocess. Parmater Z is both passed to the called process and returned to the calling process.

18. Are there any more parameters passed during this call. Enter y and the programs executes items 14 through 17.

19. Does this process call any other process? A y response to this question causes items 12 through 18 to be executed.

20. Does this process use or change any global parameters? A y response to this question causes the program to prompt you to enter the name of the global parameter and to indicate if it is used or changed by this process.

21. Enter the parent process of the process being defined.

This prompt askes for the parent structure chart of which the structure chart being defined is its child. For example, process A with a process number of 1.2 is the parent of processes C and D with process number 1.2.1 and 1.2.2 respectively.

22. Does a reference to a previous development stage exist for the process being defined? A y response to this question causes the program to prompt the user for the reference type. Acceptable response to this prompt are provided on a display menu. The program will then prompt for the reference itself.

23. Do any alias names exist for this process? A y response to this question will cause the program to prompt for the alias name and a comment as to why an alias was used.

24. Does this process read from or write to any files? A y response to this question will cause the program to prompt for the name of the file and ask for a indication as to whether the file is read from or written to by the process being defined.

25. Does this process read to or write from any hardware? A y response to this question will cause the program to prompt for the name of the hardware item and a indication as to whether the hardware is written to or read from.

26. Enter the date.

27. Enter the author's name.

1. SRUCTURE CHARTS

B. Parameter Definition. You will be provided with the following prompts.

1. Enter project name. Enter your team designation. Team a, Team b, etc.

2.Enter parameter name. Enter the name of the parameter you wish to define.

3. Classify the parameter as one of the following:

   1. Global Parameter

   2. Local Parameter

   3. Hardware Input or Output

   4. File

Enter the number corresponding to the correct response for the parameter being defined.

4. Can a data type be designated for this parameter? Enter y or n. If a n response is entered the program jumps to item 5. If a y response is entered, the following display appears.

1. Character

2. Real

3. Integer

Enter the number corresponding to the data type for the parameter being defined.

5. Can a minimum value be specified for this parameter? Enter y or n. A response of n will cause the program to jump to item 6. A response of y will cause a prompt for the

minimum value of the paramet r. Enter this minimum value when this prompt appears.

6. Can a maximum value be specified for this parameter? Enter y or n. A n respone will the cause the program to jump to item 7. A y response will cause a prompt for the maximum value of the parameter to appear. Enter the maximum value when this prompt appears.

7. Can a description of the range of values assumed by the parameter be entered? Enter y or n. A n response will cause the program to jump to item 8. A y response will cause a prompt for the range of values to appear.

8. Does a finite and reasonably small set of values exist which the parameter can assume? Enter y or n. A n response causes the program to jump to item 9. A y response will cause a prompt to appear asking you to enter a value the parameter can assume. Once you have entered the value, the program will ask you if any more values exist. As long as you respond with a y to this question, you can continue to enter a set of values the parameter can assume. When a n response is detected for this question, the program jumps to item 9.

9. Enter description. This prompt allows you to enter a description of the parameter being defined. The description will be entered in a format which allows 60 characters of text per line. A guide line will give you an indication of how much space you have left on a line. When you approach

the end of a line, hit the carriage return and the cursor will move to the next line down. You can then continue entering your text description. When you have completed the description, hit the carriage return 1 time enter the last line of your description and a 2nd time to signal the program that the description has been completed.

10. Is this parameter part of another parameter? Enter y or n. A n response to this question will cause the program to jump to item 11. A y response will cause a prompt to appear which asks you to enter the parent parameter of the parmeter being defined.

11. Can the parameter being defined be decomposed into other parameters? Enter y or n. A n response will cause the program to jump to item 12. A y response will cause a prompt to appear asking for the name of a parameter which make up the parameter being defined. After you have entered the name of a parameter which makes up the parameter being defined, the program will ask you if any more parameter exist which constitute the parameter. A y response to this question will allow you to continue entering the names of parameters. A n response will cause the program to jump to item 12.

12. Does a reference to another development phase exist for this parameter? Enter y or n. A n response will cause the progr'm to jump to item 13. A y response will cause the following display to appear.

Indicate the reference type associated with this parameter by entering the number associated with the selection given below.

1. REQUIREMENT'S NUMBER
2. SADT DATA ITEM NAME
3. DFD DATA FLOW NAME

You will select the appropriate reference type by entering the number associated with one of the items given in the menu. Once the reference type is selected, a prompt will appear which asks you to enter the actual reference to the previous design phase. Once you have entered this reference, the program will ask you if any other references exist for this parameter. A response of y to this question will continue to allow you to enter references to previous design phases. A n response will cause the program to jump to item 13.

13. Do any alias names exist for this parameter? Enter y or n. A n response will cause the program to jump to item 14. A y response will cause the program to prompt for an alias name for the parameter. Once the parameter is entered, the program will prompt the user for a comment concerning the alias, ie , why an alias name was used. The program will then ask the user to enter the name of the process where the alias is used. The program will then ask if this alias name is used in any other processes. If the response is y, the program will allow additional process

names to be entered. If the response if no, the program will ask if any other alias names exist for the parametere. A y responses will cause item 13 to be repeated. A n response will cause the program to jump to item 14.

14. Enter the date, example mo-day-yr, 12-14-84.
Enter the current date in the above format.

15. Enter the authors name, Enter your name

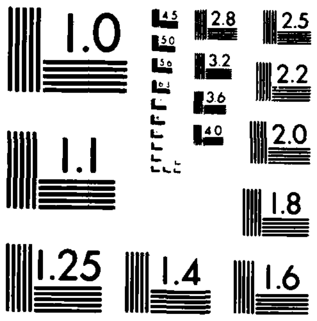
The items presented above provide a detailed example of the sequence of tool prompts which will be provided by the tool during the input of a structure chart process and parameter. The input definition routine for the other software representations are similar.

2. Definition Retrieval

In order for the tool to retrieve a definition, it requires four pieces of information from the user: type of representation, entity type (action or data), project name, and the name of the action or data entity. The type of representation and the entity type are designated by entering the appropriate selections from menu displays. Once this information is obtained the tool will prompt the user to enter the project name and entity name associated with the definition the user wishes to retrieve. The tool will then display the entity definition to the terminal screen. If the entity name entered by the user does not correspond to any definition being maintained by the tool,

AD-A152 215   AN AUTOMATED/INTERACTIVE SOFTWARE ENGINEERING TOOL TO   4/4
GENERATE DATA DICTIONARIES(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. C W THOMAS
UNCLASSIFIED   DEC 82 AFIT/GCS/ENG/84D-29   F/G 9/2   NL

END



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963 A

a message to that effect will be displayed to the user.

3. Definition Deletion

To delete an entity definition from the dictionary, the user must again select the deletion operation and the representation type and entity type (action or data) from the display menus. The tool will prompt the user for the project name and entity name associated with the definition to be deleted. Two deletion option are available :

1. Delete the results of an input operation. This option simply removes from the dictionary the effects of an input operation. This is the option to use when the user wishes to erase the effects of an erroneous input defintion operation.

2. Delete all reference to an entity from the dictionary. This option removes the entity completely from the dictionary. Upon completion of this option, no reference to the deleted entity exists anywhere in the dictionary.

4. Print Entity Definition.

To print all entity definitions belonging to a particular project, the user selects the appropriate representation type and entity type (action or data). The tool then prompts the user to enter the project name associated with these definitions. The tool then proceeds to write the definition to a file. As each entity is written to this file, the tool displays the entity name on the terminal screen. The user can then use the conventions of the operating systems to obtain a hard copy of these definitions. The following are the file names to which the corresponding definitions are written.

FILENAME	
prodef	Structure Chart Process Definitions
padef	Structure Chart Parameter Definitions
modef	Code Module Definitions
vadef	Code Variable Definitions

5. Error Messages.

The tool checks for certain errors in user input and displays the error to the user and provides the user with an opportunity to correct errors. The tool checks to see if prompts which ask yes or no questions receive y or n in response. If not, then the erroneous response is displayed to the user and he/she is asked to re enter their response to the question. The tool also checks to see if selection from menu displays are correct (ie number within the range of allowable response has been entered.)

6. At current time, the data dictionary generation tool is made up of several different programs stored under the EE 690/DD690 directory on the VAX/UNIX system. A synopsis

of these programs and the functions they perform are presented below:

Program Name

dd.out - Input and Retrieval of SADT Activity and Data Item Definitions, Structure Chart Process and Parameter Definitions.

incodemo.out - Inputs a code module definition.

outcodemo.out - Retrieves a code module definition.

incodeva.out - Inputs a code variable definition.

outcodeva.out - Outputs a code variable definition.

printpa.out - Prints out all parameter definitions under a a specific project name.

printpro.out - Prints out all process definitions under a specific project name.

printmod.out - Prints out all code module definition under a specific project name.

printvar.out - Prints out all code variable defintions under a specific project name.

Bibliography

1. Pressman, Roger S. Software Engineering: A Practitioner's Approach. New York: McGraw-Hill Book Company,1982.

2. Zelkowitz, Marvin V. "Perspectives On Software Engineering", Computing Surveys, 10: 197-215 (June 1978).

3. Softech, Inc. Model of the Current Reporting and Information Retrieval System for Air Force Program Element Monitors. Report Number 1032-1, AD Number A073119, 17 December 1976: IEEE, INC. 1979.

4. Miller, Edward. Requirements/Specification Tools, Tutorial: Automated Tools For Software Engineering, New York: IEEE, INC. 1979.

5. Peters, Lawrence J. "Software Representations and Composition Techniques", Proceedings of IEEE, 68: 1085-1093, (September 1980).

6. Bergland, G.D. "A Guided Tour of Program Design Metodologies", Computer, 11: 13-36 (October 1981).

7. Bergland, G.D. "Structured Design Methodologies", Tutorial: Software Design Strategies (Second Edit.). New York: IEEE, INC. 1981.

8. Bergland, G.D. and Ronald D. Gordon. "Software Design Strategies", Tutorial: Software Design Strategies (Second Edition). New York: IEEE, INC. 1981.

9. Jones, Meilir Page. "Transform Analysis", Tutorial: Software Design Strategies (Second Edition). New York: IEEE, INC. 1980.

10. Peter, Lawrence J. and Leonard L. Tripp. "Comparing Software Design Methodologies", Tutorial: Software Design Strategies (Second Edition) New York: IEEE, INC. 1981.

11. Lefkovits, Henry C. Data Dictionary Systems. Wellisley: Q.E.D. Information Sciences, Inc. 1980.

12. National Bureau of Standards. Prospectus For Data Dictionary Systems Standard. Report Series NBSIR 80-2115, Washington: National Technical Information Service, September 1980.

13. Hadfield, 2Lt Steven M. An Interactive and Automated Software Development Environment. MS Thesis, AFIT/GCS/EE/82D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982 (AD-A210 920).

14. Rullo, Thomas A. Advances In Data Base Management. Philadelphia: Heyden and Sons, 1980.

15. Leong-Hong Belkis W. and Bernard K. Plagman. Data Dictionary/Directory Systems Administration, Implementation, and Usage. New York: John Wiley And Sons. 1982.

16. National Bureau of Standards. Functional Specifications for A Federal Information Processing Standard Data Dictionary System. Report Series NBSIR 82-2619, National Technical Information Service, January 1983.

17. Date, C.J. An Introduction To Database Systems. Reading: Addison Wesley Publishing Co., 1981.

18. Weldon, Jay-Louise. Data Base Administration. New York: Plenum Press, 1981.

19. Atre, s. Database: Structured Techniques For Design, Performance, and Management. New York: John Wiley and Sons, 1980.

20. Ralph H. Sprague, Jr and Eric D. Carlson. Building Effective Decision Support Systems. London: Prentice Hall International, INC., 1982.

21. Weinburg, Victor. Structured Analysis. New York, New York: Yourdon Press, 1978.

22. National Bureau of Standards. Federal Requirements For A Federal Information Processing Standard Data Dictionary System. Report Series NBSIR 81-2354, Washington: National Technical Information Service, September 1981.

23. National Bureau of Standards. Federal Information Processing Standard For Data Dictionary Systems Volume 1, General Description of FIPS DDS. Washington: National Technical Information Service, August 1983.

24. INGRES Self-Instruction Guide (VAX/VMS Version 1.4, September 1981) Relational Technology Inc., 1982.

25. EQUEL/C User's Guide (VAX/VMS Version 2.1, September 1983) Relational Technology Inc., 1981.

26. Woodfill, John et al. "INGRES Version 6.3 Reference Manual. February 1981.

27. Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language. London: Prentice-Hall International, INC., 1978.

28 INGRES Reference Manual (Version 2.1, VAX/VMS, September, 1983) Relational Technology Inc., 1983.

VITA

Captain Charles W. Thomas was born on 10 May 1954 in Florence, South Carolina. He graduated from high school in Hartsville South Carolina in 1972 and attended Newberry College from which he received the degree of Bachlor of Science in Chemistry and Business Administration in May 1976. Upon graduation, he received a commission in the USAF. He completed Communications-Electronics Officer School in October 1977. He then served as Maintenance Supervisor 773rd Radar Squadron, Montauk AFS, New York until March 1981. He then served as Chief of Maintenance Photographic Processing Interpretation Facility , 16th TRS, Shaw AFB, South Carolina. While assigned at Shaw AFB, he completed all requirements and was awarded a degree of Masters Business Administration fro Golden Gate University. He entered the School of Engineering, Air Force Institute of Technology in May 1983.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/84D-29	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL (If applicable) AFIT/EN
7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.

11. TITLE (Include Security Classification)
See Box 19

12. PERSONAL AUTHOR(S)
Charles W. Thomas, B.S., Captain, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1984 December	293

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB. GR.
9	3	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Computer Software, Software Engineering,
Automated Tools, Software Development,
Data Dictionary, Database, [illegible]

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: AN AUTOMATED/INTERACTIVE SOFTWARE ENGINEERING TOOL TO GENERATE DATA DICTIONARIES

Thesis Chairman: Dr Gary B. Lamont

Approved for public release: IAW AFR 190-17.
[illegible]
[illegible] Development
[illegible]
Wright [illegible]

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr Gary B. Lamont	22b. TELEPHONE NUMBER (Include Area Code) (513) 255-5533 — 22c. OFFICE SYMBOL AFIT/ENG

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

4, thesis

The purpose of ~~this~~ investigation is to design and develop an automated/interactive software engineering tool which generates data dictionaries. This tool is to provide the user with an interactive data dictionary tool to support the develop of software in all phases of the software life cycle. The tool supports data dictionary information specific methods of software representation The initial implementation of this tool supported four methods of software representation: SADT, data flow diagrams, structure charts, and code. The requirements definition for the tool includes a discussion of the objectives and concerns associated with the tool development. Data flow diagrams are used to formulate a requirements model. The preliminary design specifies the type of information to be contained in the data dictionary for each of the methods of software representation supported and the database design required to maintain the data dictionary information. The structural framework of the application software which provides the interface between the tool user and the dictionary database is specified and structure charts are used to model this structural framework. In detailed design, algorithms are developed for the tool's application software. [illegible] typed keywords included; → [illegible]

The dictionary database is implemented through the use of the INGRES database management system. The application software is coded using the C programming language. The application software interfaces with the dictionary database by means of embedded EQUEL (INGRES Embedded Query Language) statements in the C language source code. The tool was implemented on the VAX 11/780 computer using the UNIX operating system.

SECURITY CLASSIFICATION OF THIS PAGE