THE SYSTEMS RESOURCE DICTIONARY: A SYNERGISM OF ARTIFICIAL INTELLIGENCE, DATABASE MANAGEMENT AND SOFTWARE ENGINEERING METHODOLOGIES

by

RANDALL N. SALBERG

B. S., Kansas State University, 1978

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by:

[illegible]

Major Professor

Table of Contents

Figures:

ACKNOWLEDGEMENTS

There are a number of people who deserve mention for their help in making this report possible. First, I would like to thank my wife Deborah for her support and patience during the production of this report. I also wish to thank Professor Roger Hartley, without whom this report may not have been finished, for his encouragement and suggestions. Lastly I thank the other members of my committee, Professors E. Unger and D. Gustafson for their comments and suggestions, and Professor V. Wallentine for filling in for Professor Hartley.

Chapter 1

Introduction

In the course of the past five years it has become increasingly evident that there is a profound need for the various branches of Computer Science to communicate their achievements to each other. What follows is an attempt to bring together developments achieved in three areas, Artificial Intelligence, Database Management and Software Engineering, in order to achieve advances in the areas of software engineering and database management.

The first objective is to present the findings of a literature search which encompassed selected topics within the three chosen research areas.

Within the area of artificial intelligence expert systems, what they are, how they are used and a few examples will be examined. Included in this examination will be a discussion of methodologies for representing knowledge and how this knowledge is used in expert systems.

The modeling of data and data dictionaries and their use in a database environment will be of primary interest in the area of database management. In data modeling brief definitions and examples of the three types of database models, these being the hierarchical, network and relational models, will be given. The data dictionary will be defined and a typical data dictionary described. In addition some ways in which the data dictionary could be improved are examined.

Turning to software engineering various means of shortening the software development cycle will be investigated. This discussion will include a look at various tools currently available or proposed for use in software development. In addition the basic steps required of any software development methodology for successful software development will be reviewed.

The second step will then be to draw these lines of research together to show how they are similar, how they complement each other, and how developments in one area can be used to enhance methodologies used in other areas. Ways in which this information can be used to develop a means of producing software within a database environment will then be shown. The manner in which this methodology is more efficient in all areas of major concern to the software developer and how it provides a built in means of documenting the development and expansion of the database system for the database administrator will also be shown.

The methodology proposed is essentially an enhanced data dictionary. The last section of this paper will present this enhanced data dictionary, henceforth referred to as the Systems Resource Dictionary (SRD), as a means of achieving the developments proposed earlier.

In proposing this Systems Resource Dictionary a data dictionary model proposed by Rob Phillips, called the Dynamic Data Dictionary will be used. This dictionary model provides the base upon which to expand to achieve the stated goals. The definition of the Systems Resource Dictionary will be given along with an explanation of why this methodology is useful. This will be followed by a description of how the SRD could be used in software development and database management. The required techniques for the successful completion of the SRD and a list of requirements for the Systems Resource Dictionary will then be set down. Through out this discussion ways in which artificial intelligence techniques could be used to enhance the operation and usefulness of the SRD will be examined. Finally possible problems in the development of the Systems Resource Dictionary will be discussed.

Chapter 2
Literature Review

2.1 Artificial Intelligence

The concept of artificial intelligence is an ill defined concept. There are, however, a number of tools or concepts that have proven to be of use in various disciplines both inside and outside the discipline of computer science. These topics include expert systems and the associated concept of knowledge representation systems.

2.1.1 Knowledge-Based Expert Systems

A precise definition of an expert system has not been given. An explanation as given by Hayes-Roth [9], to explain what an expert system is, what it isn't and how to recognize one, will therefore be used.

According to Hayes-Roth, an expert system is a knowledge intensive program that solves problems normally requiring human expertise. Secondary functions normally performed by the expert such as asking relevant questions and explaining its reasoning are performed by the expert system. Examining what expert systems do can determine their common characteristics. These characteristics as given by Hayes-Roth [9] are:

"They solve very difficult problems.

They reason heuristically, using what experts consider effective rules of thumb.
They interact with humans in appropriate ways, including the use of natural language.
They manipulate and reason about symbolic descriptions.
They function with erroneous data and uncertain judgmental rules.
They contemplate multiple competing hypotheses simultaneously.
They explain why they're asking a question.
They justify their conclusions."

Current expert systems are shallow, brittle, narrow, and lack the breadth of knowledge and understanding of fundamental principles shown by the human expert. The expert system is only ✓ a gross simulation, of the human experts thinking processes, that explains relevant criteria and makes educated guesses, much like a human expert does for the purpose of making major decisions. It does not however perceive significance, jump to conclusions intuitively, or examine an issue from different perspectives as the human expert does. Nor does the expert system reason from first principles, draw analogies, rely on common sense, or learn from experience.

Examining the current state of automatic data processing systems shows that they are designed to amass and process large volumes of data algorithmically, in order to automate time consuming clerical functions. On the other hand, expert systems usually address small tasks performed by professionals in minutes or hours. These tasks include interpreting, diagnosing, ✓ planning, scheduling, etcetera. An expert system makes judicious use of data and reasons with it to accomplish tasks. Unlike the algorithmic data processing approach the expert system examines a large number of possibilities or constructs a solution dynamically.

An expert system can be recognized by looking at the task performed by the system [9]. If the function performed is one that was previously done by a human expert; if the system has a knowledge base in which the knowledge is accessible, that is it can be read, you can ask for explanations requiring it and justifications validating it, and it can be modified; if the system stores a substantial body of knowledge that it reasons with in flexible ways; and if the system employs heuristics or judgmental knowledge then it is an expert system.

Now that how to recognize an expert system is known, some of the tasks they perform can be examined. The easiest way to do this is to briefly describe some generic expert tasks [11]. This description will also give an idea of what makes expert reasoning so difficult. As these tasks are examined you will notice that several issues appear repeatedly. These issues are large solution spaces, tentative reasoning, time-varying data and noisy data.

Interpretation, the first of the expert tasks to be examined, is the analysis of data to determine their meaning. It requires that consistent and correct interpretations of the data be found. Analysis systems that are as rigorously complete as possible are also necessary. The key problem in interpretation is noisy and errorful data, i.e., missing, erroneous or extraneous data values. Only partial information may be available for the interpreter and data for a given problem may seem contradictory. This requires the interpreter to hypothesize which data are believable. Unreliable data brings about unreliable interpretations and requires the identification of where uncertain or incomplete information and assumptions are used. Lastly, long and complicated reasoning chains require explanations of how the interpretation is supported by the evidence.

The requirements of the next task, diagnosis, includes those of interpretation. Diagnosis is the analysis of the cause or nature of a problem or situation based on the interpretation of potentially noisy data. An understanding of the systems organization, and the interactions and relationships between subsystems is required of the diagnostician. Some key problems here are: faults can be masked by the symptoms of other faults; faults can be intermittent; failure of diagnostics equipment; system data may be inaccessible, expensive, or dangerous to retrieve; may need to combine several partial models due to a lack of understanding of the anatomy of natural systems.

Monitoring, the third task, is the continuous interpretation of signals for the purposes of systems intervention when the appropriate alarm signal is received. A monitoring system is a partial diagnostics system with real time recognition of alarm conditions and avoidance of false alarms required. The key problem with a monitoring system is that signal expectations have to vary with time and situation due to the context-dependent nature of alarm conditions.

Prediction, the use of a model of the past and present to forecast the course of the future, is the fourth task. It requires reasoning about time. Predictors must be able to refer to events that are ordered in time and to the transformation of things over time. Prediction also requires that the manner in which various actions change the state of the modeled environment over time be adequately modeled. Key problems in prediction are: the integration of incomplete information is required; multiple possible futures should be accounted for and sensitivity to variations in the input data should be indicated; indicators of the future can be found in many places requiring the use of diverse data; nearer but unpredictable events may effect the likelihood of distant futures so predictive theory may need to be contingent, i.e., conditional, dependent for occurrence, existence, character, etc., on something not yet certain.

A program of action that is carried out to achieve stated goals is a plan. Planning, the creation of plans, is the fifth task. It requires that the constructed plan achieve its goals without the use of excessive resources or violation of constraints. The plan must avoid goal conflicts through establishment of priorities and must also prepare for contingencies. Planning also has the same requirements as prediction. Some problems with planning are: large and complicated problems require tentative action that enables exploration of possible plans; the overwhelming details require that the planner be able to focus on the most important considerations; must cope with goal interactions and relationships between plans for different subgoals in large complex problems; the planning context is often only approximately known requiring the planner to operate in the face of uncertainty; coordination is required when multiple agents are needed to carry out the plan.

The last task, design, is the making of specifications to create objects that satisfy particular requirements. Design has many of the same requirements as planning. Key design problems are: the consequences of design decisions are not immediately assessable to the designer in large problems; tentative exploration of design possibilities is required; there are many sources of design constraints with the result that a comprehensive theory for the integration of constraints with design choices is usually lacking; system complexity must be coped with by factoring the design into subproblems, thus requiring the designer to cope with interactions between subproblems since they are seldom independent; it's hard to assess the impact and easy to forget the reason for design decisions or changes in design when the design is large; one must be able to reconsider design possibilities when modifying the design in order to avoid what are only locally optimal design spaces; the methodology necessary to reason approximately or qualitatively about shape and spatial relationships is not available.

In surveying the current state of expert system technology we should understand a few key points:

> Knowledge systems transform book knowledge, and "private" or distributed knowledge into an active, inspectable form capable of performing high-value work.
>
> Knowledge takes many different forms, necessitating a variety of tools or instruments for knowledge engineering.
>
> Knowledge systems must integrate with conventional data processing systems, but knowledge engineering work is different from conventional software work. [9]

Knowledge system technology consists of two basic ingredients: symbolic programming and knowledge engineering. Symbolic programming technology consists of the fundamental science of symbolic computation, practical techniques for constructing symbolic programming systems and techniques for building incremental programming environments. The three key ingredients added by knowledge engineering are, problem solving engines, knowledge bases and knowledge base maintenance.

In a knowledge system the activity of problem solving is organized by the problem solving engine. In order to understand the problem solving engine one must relate its design and intended purpose to its implementation. The aim of today's knowledge systems is to solve specific problems. The knowledge engineer analyzes a problem and adopts an overall approach. This approach consists of: (1) a problem solution paradigm (top-down refinement, multidirectional opportunistic search); (2) a general knowledge system architecture that reflects specific choices about system design; choices include: kinds of knowledge to represent, using what formalism, for what kinds of inference and allow what kind of flexibility; and (3) a specific problem solving strategy, that determines which knowledge to apply, and in what order to apply it. Currently the knowledge engineer's choices are implemented by specific devices provided by the problem solving engine.

A problem solving engine provides a knowledge representation formalism and related interpreter, a high-level control architecture and executive, and an inference procedure and related inference engine. Ways to describe conceptual taxonomies and conditional heuristic rules may be included in the the knowledge-representation formalism. Conceptual taxonomies include relationships among classes, individuals, and types of objects and determines how properties of one relationship apply to another.

Conceptual heuristic rules represent judgmental knowledge and are represented in many current problem solving engines by stylized "IF-THEN" formalisms. Many kinds of formalisms and heuristics can be supported by current engines.

A knowledge system's problem solving work is structured and carried out by the high-level control architecture and executive, and an inference procedure and related inference engine. Several architectures are used today to determine a solution.

The most frequently used architecture is a goal-directed backward-chaining system. This system has worked well for selection, diagnosis, and consultation applications. Two other architectures are the data-driven or forward-chaining system and the hear-say-like or "blackboard" system.

The forward-chaining system is useful for modeling cognitive processes and for solving diverse problems requiring very broad, but shallow knowledge. The problem solving engine functions by choosing one satisfiable rule to execute at a time.

Multiple cooperating subexpert systems, or specialists are used by hear-say-like systems. This type of system has proved useful for complex problems in design, planning, speech and vision.

Once a particular problem solving engine has been chosen, by the knowledge engineers, and before starting the application, It's knowledge base must be created. The knowledge base is generally made up of conceptual taxonomic relations and rules. Hayes-Roth has listed some generally valid heuristic rules about knowledge bases. Some of these rules are:

> Only 50 rules are needed to provide an interesting demonstration of the technology.
> Two hundred and fifty rules provide a convincing demonstration of a knowledge systems power.
> Five hundred to 1000 rules are required for an expert level of competence in a narrow area.
> Ten thousand plus rules are required for expertise in a profession. [9]

Automated aids for amassing and maintaining these rules are provided by most research and commercial knowledge-engineering tools. These tools generally include token completion, spelling correction, line of reasoning traces, knowledge base browsing, and automated system testing and validation.

Knowledge systems address problems arising from difficulties in retaining, transmitting and applying know-how by providing a means to employ know-how when and where it is needed at great speed. They address the need to distribute, preserve and reason about knowledge electronically. There are numerous knowledge system application architectures and representation schemes because knowledge is varied and resists efforts to apply or represent it in a standard way.

2.1.2 Knowledge Representation Systems

Of the many knowledge representation schemes only four will be discussed. These four consist of predicate logic, semantic networks, production systems and frames.

2.1.2.1 Predicate Logic/Calculus

The ability to express true or false propositions, as in propositional logic, is not adequate for capturing a formalism of one's knowledge of the world. It is also necessary to be able to speak of objects, to postulate relationships between these objects, and to generalize these relationships over classes of objects [1]. Predicate calculus is an attempt to achieve these goals through an extension of the notions of propositional calculus.

In predicate calculus the focus of the logic is changed while the meanings of the logic connectives are retained. Predicate calculus, rather than being interested merely in the truth value of statements, is used to represent statements about individual or specific objects.

Predicates are statements about individuals. When a specific number of individuals are used as the arguments to which a predicate is applied, the result of the predicate has a value of either true or false. Some other attributes of predicates are that they can have more than one argument and each one-place predicate defines what is called a set or sort. That is, for any one-place predicate P, all individuals X can be sorted into two disjoint groups, one whose objects satisfy P (P(X) is true) and one whose objects don't satisfy P [1].

Two notions have been introduced to refer to facts that are known to be true of all or some of a sort's members. The first notion is that of the variable. It is a place holder that is to be filled in by some constant. The second notion is that of quantifiers. There are two quantifiers, one meaning there exists

..., the other meaning for all .... Using syntactically allowed combinations of the constants, variables, quantifiers, predicates and connectives complicated expressions can be created.

The introduction of four more inference rules is implied by the use of the quantifiers [1]. These consist of rules for the introduction and elimination of each of the two quantifiers. Predicate calculus is the result of the extension of the rules of propositional calculus by predicates, quantification, and the inference rules for quantifiers.

As described predicate calculus can be clumsy and is very general. There are two additions which can be made to the logic making some things easier to say, while not really extending the range of what can be expressed. Functions or operators are the first addition. Some attributes of functions are: they have a fixed number of arguments like predicates; unlike predicates they do not have just true or false values, but they return objects related to their arguments. The predicate equals is the second addition. Informally it means that two individuals X and Y are equal if and only if they are indistinguishable under all predicates and functions.

With these additions a variety of first order logic is obtained and one no longer has a pure predicate calculus.
First order logics allow quantification over individuals but not over functions and predicates. It is impossible to prove a false statement and any true statement has a proof in a first order logic.

The AI problem can be divided into two parts, the epistemological and the heuristic. Using this distinction several reasons for logic's usefulness as a means of exploring epistemological problems can be seen. First the expression of certain notions in logic often seem natural. Three other reasons for logic's usefulness are that it is precise, flexible, and modular. However this separation of processing and representation is also a major disadvantage of logic in AI. It is the heuristic part of the system, deciding how to use the facts stored in the system's data structures not deciding how to store them, that is currently causing problems. Addressing that problem is merely postponed by separating the two aspects and concentrating on the epistemological questions.

2.1.2.2 Semantic Networks

Semantic Networks are one means of systematizing the structure of knowledge. The major thrust behind this approach is that knowledge and the sorts of things that can be done with it are reflected in ones use of language. The use to which the knowledge is put can never be entirely forgotten. The structure and use of knowledge is over and above the structure and use of language. Subservient to ones knowledge, desires, and beliefs are the pragmatic and semantic aspects of language. The assumption of some prior epistemological theory is required before an analysis of language can proceed beyond the purely syntactical level. A plausible theory of the structure of knowledge can be generated through a process of drawing analogies from the analysis of the use of language. When elucidated through a model, such a theory is called a semantic network.

The formulation of all semantic network models falls under the methodology outlined above. The basic form of all networks is the same, it is a collection of nodes, joined together by directional arcs i.e. lines with an arrow on them (see figure 1). Directed graph is the term used in mathematics for a semantic network model. A node may in general have any number of arcs going out of or coming into it. This characteristic accounts for the use of the network as opposed to hierarchy, tree, etcetera.

Semantic objects (atomic units of meaning) are represented by the nodes in semantic networks. The arcs represent relationships between the nodes. By showing how semantic objects are related and how they participate in larger structures a network can give a picture of a theory of the structure of knowledge. The network is given greater expressive power by allowing both nodes and arcs to be of different types. Type differentiation is accomplished through pictorially differentiating the nodes or by labeling the nodes and arcs in different ways. It is common to use a mixture of the two methods.

There are many different types of semantic objects (nodes). Four basic types of nodes are:

EVENT analogous to a verb, e.g., throwing

ENTITY analogous to a noun, e.g., Paul, ball

ATTRIBUTE analogous to a modifier or qualifier,
e.g., sweet, swift

MODALITY reflecting second-order concepts like
definitions, beliefs, abstractions

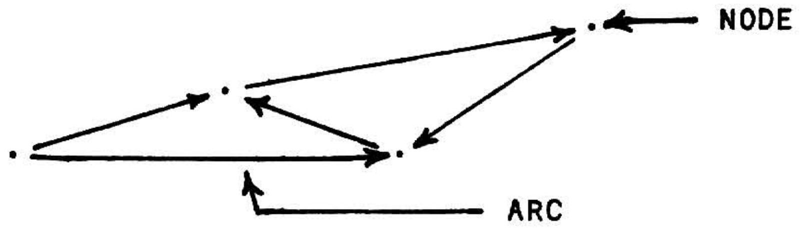


Figure 1: Semantic network.

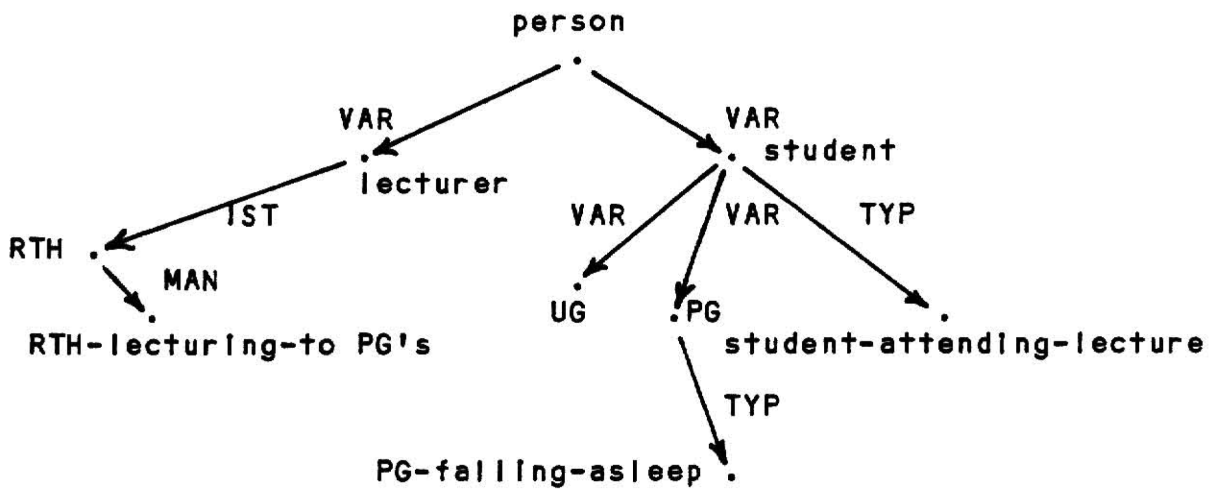


Figure 2: semantic network example.

There are three main categories for the many and various relationships between the object types. These categories are propositional representation, propositional connection, and classification. Each of these categories may have a number of different types. Figure 2 shows an example using some of the above terminology.

Semantic networks work best in situations where there is a need for a large body of slowly changing assertional sort of knowledge. By using a fixed set of procedures which traverse this network many useful forms of database inquiry can be generated. The network methodology seems to be impractical for modeling behavior where parts of the network are subject to rapid changes which effect other parts of the network. The network approach is one way of theorizing about the relationships between language, logic and human action, and with its pictorial presentation, it is easy on the brain and eye at the very least.

2.1.2.3 Production Systems

Production systems are systems based on one very general underlying idea. That idea is the notion of condition-action pairs, called production rules [1]. There are three parts to a production system. These parts are: (1) a set of production rules composing a rule base; (2) the context, which is a special buffer-like data structure; and (3) a systems activity controller called an interpreter.

Production rules take the general form:

```
IF Assertion 1 and
   Assertion 2 and ...
THEN Action 1,
     Action 2, ...
```

Most systems consider the two sides of the rules to be independent. The IF part of the rule states the conditions to be tested against current context and is called the condition part or left-hand side (LHS). The THEN part of the rule is the action(s) to be carried out if the condition(s) are meet. It is called the action part or right-hand side (RHS). A production rule whose LHS is satisfied can have its RHS executed by the interpreter during execution of the production system. A typical production system may contain hundreds of these productions in their rule bases.

The focus of attention of the production rule is the context, sometimes called the short-term memory or data buffer. Before the action side of a production in the rule base can be executed its condition side must be present in the context data structure. The context can be changed by action parts of the production rules so that the condition parts of other rules are satisfied. Usually the context data structure is a medium-sized buffer with some internal structure of its own, it may however be a simple list or a very large array.

Lastly there is the interpreter, which like an interpreter in any other computer system, is a program whose task it is to decide what to do next. In production systems it is the interpreter's job to decide which production rule to execute next.

Production systems operate in cycles. The interpreter specifies the manner in which the production rules are to be examined in each cycle to determine which productions are appropriate and can execute. A single production is selected from among those that are found appropriate and it is fired. These steps, matching, conflict resolution and action, are the three phases of each cycle.

Certain features of the production system formalism, both good and bad, can be generalized. These features are: Modularity - the formalism allows the addition, deletion, or change of individual productions in the rule base; Uniformity - knowledge in the rule base has a uniform structure imposed on it; Naturalness - certain kinds of important knowledge can be expressed with ease; Inefficiency - a high overhead results from the use of strongly modular and uniform production rules in problem solving; Opacity - the flow of control in problem solving is hard to follow because algorithmic knowledge is not expressed naturally.

Characterizing the domains for which production rules might be a useful knowledge representation scheme is a more fruitful method of evaluating the utility of production systems than evaluating their features. The appropriate domain for production systems can be characterized as consisting of tasks which can be viewed as a sequence of transitions from one state to another in a problem space [1]. Since each transition can be effectively represented by the execution of one or more productions, tasks displaying this behavior can be modeled with production systems.

2.1.2.4 Frames and Scripts

Representing knowledge about objects and events typical to specific situations is the focus of the AI knowledge representation ideas called frames and scripts. Frames and scripts refer to methods of organizing the knowledge representation so as to direct attention and facilitate recall and inference [1].

New data are interpreted in terms of concepts acquired through previous experience within a structure, a framework, provided by frames. Expectation driven processing, looking for expected outcomes based on the context of the situation you are in, is facilitated by this organization of knowledge. This kind of reasoning is made possible by a representational mechanism called a slot. The slot is the place within the larger context created by the frame where knowledge fits (see figure 3). The slot mechanism permits reasoning based on seeking confirmation of expectations, "filling in slots", by supplying a place for knowledge, and creating the possibility of missing or incompletely specified knowledge.

Slots can be used to represent many different kinds of knowledge and in some systems can have a complex frame-like structure, referred to as subslots, of their own. A few examples (the terminology used is intended only to give a sense of the structure of knowledge in a frame) of different kinds of slots are [1]: Specialization-of slot, allows information about the parent frame to be inherited by its children by establishing a property inheritance hierarchy among the frames; If-Needed slot, a procedure attached to this slot can be used to determine the slot's value if necessary; Default slot, an important slot type which, unless there is contradictory evidence, suggests a value for the slot; Event-Sequence slot, allows the use of a script to represent knowledge about what typically happens in a given situation (see figure 4); and Range slot, provides an expectation about the what kinds of things a slot might be (an example of a subslot). Figure 5, is an example of a frame using a different terminology that is still quite similar to the previous two examples.

The Events-sequence slot indicates that knowledge about what typically happens in a given situation might be represented in a script. A normal or default sequence of events as well as exceptions and possible error situation are specified by the script. A few static descriptions, such as Roles and Props that refer to other frames, are also required by the script.

An important dynamic or procedural aspect of frame-based systems underlies the declarative structure of frames and scripts. As mentioned above, in order to drive the reasoning or problem solving behavior of the system, procedures can be attached to the slots [1]. Attached procedures can be the primary mechanism for directing the reasoning process. They are activated to fill in slots or triggered when a slot is filled.

The primary process in a frame-based reasoning system, after a certain frame or script has been selected to represent the current situation or context, is often filling in the details called for by the slots. The two methods of filling in slots accounting for most of the power of frames are default and inherited values [1]. They are relatively inexpensive and don't

```
CHAIR Frame

     Specialization_of: FURNITURE
     Number_of_legs:    an integer (default=4)
     Style_of_back:     straight, cushioned, ...
     Number_of_arms:    0, 1, or 2
```

Figure 3: Example of a frame [1].

```
EAT_AT_RESTAURANT Script

     Props:  (Restaurant, Money, Food, Menu, Tables, Chairs)
     Roles:  (Hungry_Persons, Wait_persons, Chef_persons)
     Point_of_View: Hungry_Persons
     Time_of_Occurrence: (Time_of_Operation of Restaurant)
     Place_of_Occurrence: (Location of Restaurant)
     Event_Sequence:
          first:   Enter_Restaurant Script
          then:    If (Wait_To_Be_Seated_Sign or
                   reservations)
                   Get_Maitre-d's_attention Script
          then:    Please_Be_Seated Script
          then:    Order_Food Script
          then:    Eat_Food Script unless (Long_Wait) then
                    Exit_Restaurant_Angry Script
          then:    If (Food_Quality was better then
                    Palatable) then
                    Compliments_To_The_Chef Script
          then:    Pay_For_It Script
          Finally: Leave_Restaurant Script
```

Figure 4: Example of a script [1].

```
THING
  AKO          $IF-ADDED     (ADD-INSTANCE)
               $IF-REMOVED   (REMOVE-INSTANCE)
  INSTANCE     $IF-NEEDED    (INSTANTIATE-FRAME)
               $IF-ADDED     (ADD-AKO)
               $IF-REMOVED   (REMOVE-AKO)
```

Figure 5: A general frame representation as used in NUDGE [6].

require powerful reasoning processes. They allow the augmentation of general problem solving methods by domain-specific knowledge about how to accomplish specific, slot-sized goals.

Procedural attachment routines, that take over control only when certain events or data occur, implement event or data-driven processing. When the value of a slot is found or changed these "trigger" procedures are activated. Some systems use these trigger procedures to decide how to proceed in the event that the frame, to which they are attached by special slots, is found not to match the current situation.

Large amounts of knowledge are needed to perform cognitive tasks. Frames and scripts are recent attempts to provide a method for organizing this knowledge. Some of the domains and problems that researchers in this area address include: medical diagnosis, natural language understanding, management scheduling requests, and physics. Researchers have also addressed the development of frame-based programming and representation languages. Lastly the understanding of sequences of events and the notion of causality have been addressed using scripts.

2.2 Database Management Systems

Data management is a major task in the operation of any large computer system. As systems have grown the amount of data processed has grown as well. To cope with this data explosion, database management systems were developed. A database is a collection of data that is integrated, sharable and nonredundant. The software that allows one or more persons to use and/or modify a database is the database management system, a computer system that provides for the storage and retrieval of information about some domain.

A database management system (DBMS) consists of:

An organized collection of data about some subject, the database.

A data manipulation language, for querying and altering the data.

A data definition language, for developing a description of the organization of the data called the database schema.

Database subschemas (views) for presenting customized versions of the database schema to users. Constructed using a data definition language.

Constraints for ensuring the integrity of the database.

Provisions for concurrency, backup, and security [2].

and may contain:

A data dictionary containing the data definitions and other useful information about the data.

Rather than deal with the data as the computer stores it the DBMS allows the user to deal with the data in abstract terms. There are many levels of abstraction between the user and the computer. A viewpoint of data abstraction that is fairly standard consists of a single database with three different levels of abstraction (the database may be one of many using the same DBMS). The abstraction levels are the users view(s), the logical database and the physical database. Each user view is an abstraction of a portion of the logical database which is itself an abstract representation of the physical database.

2.2.1 Database Modeling

It is at the logical level of abstraction that databases are defined. There are three alternate models for viewing and manipulating data at this level regardless of the underlying and supporting physical data structure. These models are the hierarchical model, the network model, and the relational model. Using any of these three models, any logical database structure, called the logical scheme, can be defined.

2.2.1.1 Relational Model

The relational model is a mathematically derived methodology of data management. Simply defined, it is a database model in which the database is made up of a set of flat tables or relations, in which relationships are expressed by the fact that two relations have a field or domain in common and in which the relationships can be 1:N or M:N [12].

The set theoretic relation is the mathematical concept underlying the relational model. A relation is a subset of the cartesian product of a list of domains. A domain is a set of values and D1 x D2 x ... x Dk, the cartesian product of domains D1, D2, ..., Dk, is the set of all k-tuples (v1,v2,...,vk) such that v1 is in D1, v2 is in D2, etcetera.

Any subset of the cartesian product of one or more domains is a relation. Each relation has members called tuples and each relation that is a subset of a given cartesian product is said to have arity k. A tuple (v1, v2, ..., vk) has k components [26].

As stated above a relation can be viewed as a flat table, each row is a tuple and each column, often given names, called attributes, corresponds to one component. The order of the columns becomes unimportant when attribute names are attached to the columns. The tuples are viewed, in mathematical terms, as mapping from attributes' names to values in the domains of the attributes. Certain relations are made equal by this change in viewpoint that were not equal using the more traditional definition of a relation.

It is sometimes necessary and fortunately easy to convert from one relation definition to another. Given a set-of-mappings view of a relation, the conversion to a set-of-lists view is achieved by simply fixing an order for the attributes. Conversely, starting with a set-of-lists, conversion to a set-of-mapping is done by giving arbitrary attribute names to the relations columns.

Taking the set-of-mapping view of relations as standard allows the relation scheme to be defined as the list of attribute names for a relation. The relation scheme is analogous to a record format. Similar analogies can be seen between a relation and a file, and between a tuple and a record. These analogies show that one possible implementation for a relation is as a file of records with a record format equal to the list of attributes in the relation scheme and with one record for each tuple.

The relational database scheme is made up of a collection of relation schemes used to represent information. The relational database is the current values of the corresponding relations. Any interpretation can be placed on tuples and relations can be created with any set of attributes for a relation scheme.

2.2.1.2 Network model

The network model can be defined as a model for defining databases in which any record type can be related either as a child or as a parent record type to any number of other record types; for each parent record occurrence there may be one or many related child record occurrences [12]. Ullman [26] also restricts relationships in the network model to be binary many to one relationships. More general definitions allow many to many relationships requiring only that they be binary. The restriction to many to one relationships in the network model allows the use of a simple directed graph model for data, and makes implementation of relationships simpler.

In discussing the network model Ullman's terminology shall be used. Ullman in discussing the network model refers to logical record types which are defined as essentially a relation, that is, a named set of tuples. The possibility of the existence of two identical records of the same record type, distinguishable only by their relationship to records of another logical type, is admitted. The term logical record is used in place of the relational term *tuple*, and logical record format is used in place of relation scheme. The term field is used to refer to the component names in a logical record format.

Ullman uses the term link to discuss binary, many to one relationships. To represent record types and their links a directed graph called a network is drawn. The nodes of a graph correspond to record types. When there are two record types T1 and T2 that are linked in a many to one relationship from T1 to T2 then a directed arc is drawn from the node for T1 to that for T2. This link is said to be from T1 to T2. The arcs and nodes of the directed graph are labeled by the names of their links and record types.

Only relationships that are binary and many to one (one to one as a special case) can be represented directly by links. Arbitrary relationships, such as many to many, can be represented using the following trick. Given logical record types E1, E2, ..., Ek with a relationship R, a new logical record type T, representing k-tuples (e1, e2,..., ek) of entities that stand in relationship R, can be created. This record types format might consist of one field that is a serial number identifying logical records of that type. The new record type T may have other information carrying fields in its format when it is convenient. Links L1, L2, ..., Lk are then created where link Li is from record type T to record type Ti for entity set Ei. The intention is that record type T for (e1, e2,..., ek) is linked to the record type Ti for ei, so each link is many to one [26].

2.2.1.3 Hierarchical model

A hierarchy is a network, a collection of trees or a forest, whose links all point from child to parent. The hierarchical model is a model or data structure for defining databases in which a parent record type can have one or more child record types, but which does not allow (1) a child record type to have more than one parent record type, nor (2) an M:N relationship of instances between two record types [12]. In talking about hierarchies the same terminology as was used in talking about networks will be used. The virtual logical record, a pointer to a record of a given logical type, is a useful concept that is added to the discussion. When it is intuitively useful to place a record type in two or more trees of a hierarchy or in several places in the same tree the virtual record type is used. The use of the virtual record type means each logical record type appears only once in the hierarchy while other instances of that record are given virtual records instead.

Any network can be converted into a hierarchy, perhaps using virtual record types. The root of the tree starts with a logical record type R. R is chosen to have no links leaving if possible. Any types having links entering R are its children and their children are found by following links backwards from them. Upon finding a type that is already placed in the hierarchy a virtual record type is created and placed in the hierarchy in place of the logical record encountered and that virtual records children are not added. When no more children can be added to the tree under construction a logical record type not already placed in the hierarchy is looked for. Using a previously unplaced logical record type the tree construction process is repeated. Construction is done if none can be found.

Zero or more nodes, representing a logical record type present in the hierarchy, will be present in the database. A dummy root with child record occurrences of the root record type is sometimes useful as a tool for drawing the trees in the database.

2.2.1.4 Data Definition languages

Data definition languages (DDL) are provided by the DBMS to specify the logical scheme and some of the details of the implementation of the logical scheme by the physical scheme. The DDL is a high-level language that enables the description of the logical scheme in term of a data model. Rather than being a procedural language, it is a notation for describing the types of entities, and relationships among entities in terms of a particular data model [26].

The DDL is used when the database is designed or modified not when the data itself is being obtained or modified. It usually has statements that describe in somewhat abstract terms what shpuld be the physical layout of the database. The DDL statements are processed by DBMS routines that produce a detailed design of the physical database.

Data definition languages exist for the definition of both the logical scheme and subschemes. The subscheme DDL is needed to describe the subschemes or users' views and their correspondence to the logical scheme. The subschema DDL can be quite similar to the DDL. However, it can use a data model different from that of the DDL and it is possible to have several different subschema DDLs, each using a different data model.

2.2.1.5 Data Manipulation Languages

Specialized languages for the manipulation of the database are called data manipulation or query languages (DML or QL). They are used to express commands for the manipulation of the data in the database. DMLs are usually invoked by a host language such as COBOL or PL/1.

DML commands are invoked in either of two ways depending on characteristics of the DBMS. One way is for commands to be invoked by calls to procedures provided by the DBMS. These procedures are given access to the subscheme and logical scheme definition. In the second way, command statements are provided by an extension to the host language. The application program is written in the extended host language and the commands of the DML result in calls to procedures provided by the DBMS.

An application program's view of the data can be seen in figure 6. The figure's solid lines represent manipulation and transfer of data and the dashed lines represent causation. Data manipulation commands cause data transfers between the program's work area and the database; data manipulation commands are invoked by the applications program; as in any ordinary programming situation, statements of the application program cause data transfer and calculation within the program's workspace.

A data manipulation language allows four basic operations:

(1) Insert - place a new record in an existing database

(2) Delete - remove an existing record from the database

(3) modify - change a field or fields of an an existing record

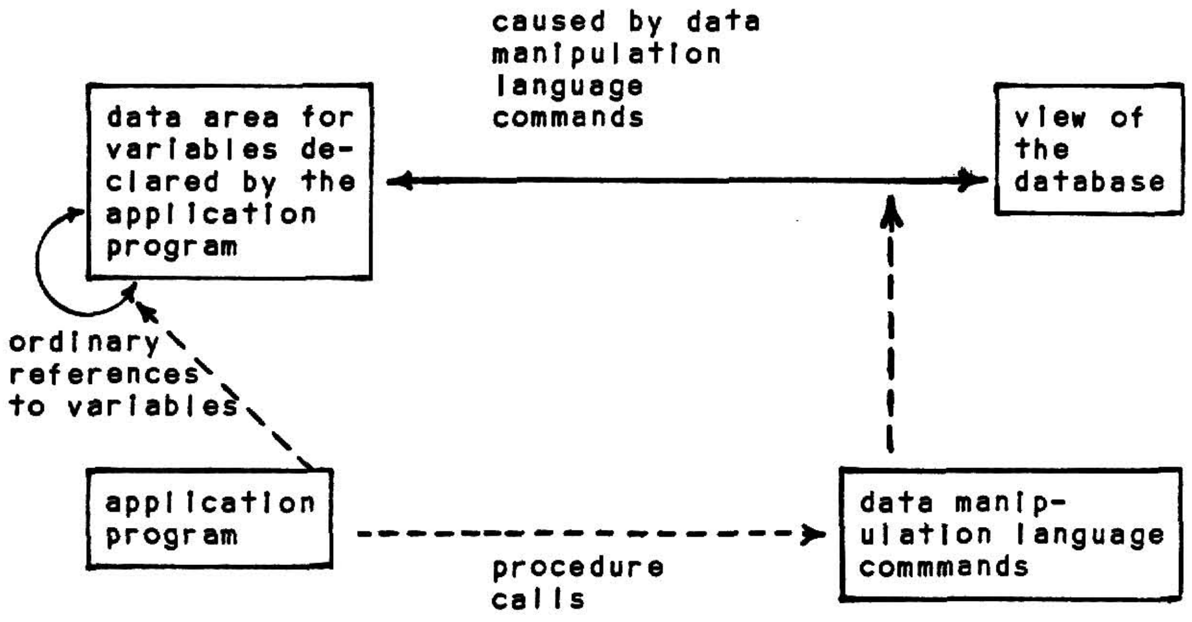

caused by data manipulation language commands
data area for variables declared by the application program
view of the database
ordinary references to variables
application program
procedure calls
data manipulation language commmands

Figure 6: The data seen by an application program [26].

(4) Lookup or search, i.e., find a record with a particular value in a particular field or a combination of values in a combination of fields.

The implementation of these four functions depends on how the file implementation strategy used for data storage is done and is outside the scope of this paper.

A brief description of DMLs for the relational model will give a better idea of what a DML does. The notation for expressing queries is usually the most significant part of a DML. A relational DML's or QL's nonquery aspects are often straight forward concerned with the insertion, deletion, and modification of tuples, while its queries, in the most general case, are arbitrary functions applied to relations often using a rich high level language for their expression.

Relational model QLs can be broken down into two broad classes:

(1) Algebraic languages, where specialized operators are applied to relations to express queries and

(2) Relational calculus languages based upon predicate calculus, where queries describe a desired set of tuples by specifying a predicate the tuples must satisfy [26].

Calculus based languages are further divided into two classes, depending on whether the primitive objects are tuples or are elements of the domain of some attribute, giving a total of three distinct kinds of QLs.

The three types of abstract QLs are relational algebra, tuple relational calculus, and domain relational calculus. They serve as bench marks for evaluating existing systems as they are not themselves implemented. Real QLs usually provide the capabilities of the abstract languages along with additional capabilities.

The three notations are equivalent in their expressive power. While almost all modern QLs embed one of these notations within themselves, some are best viewed as having a combination of these notations embedded in themselves.

As mentioned earlier all DML include commands for insertion, deletion, and modification. Other features which are frequently available include:

(1) arithmetic capability. Allows atoms in calculus expressions or selections in algebraic expressions to involve arithmetic computation as well as comparisons.

(2) assignment and print commands. Permit the printing of the relation constructed by a DML expression or the assignment of a computed relation to be the value of a relation name.

(3) aggregate functions. Allow a single quantity to be obtained from a relation's columns by applying such operations as average, sum, minimum, or maximum. [26]

These features make most implemented DMLs more than complete. That is, they are able to compute functions that have no counterpart in relational algebra or calculus.

2.2.1.6 Sample of Methodologies

Two tools that are of interest in this discussion are SDM, a semantic database model for database description, developed by M. Hammer and D. McLeod, and TAXIS, a language facility for designing database-intensive applications developed by J. Mylopoulos, P. Bernstein and H. Wong.

SDM, a Semantic Database Model [17] is a high level database model based on semantics. It is designed to capture more of the meaning of an application environment than contemporary database models do by describing, the kinds, classifications and groupings, and the structural interconnections among the entities that exist in the applications environment through the use of high-level modeling primitives to capture the semantics of the applications environment.

TAXIS was developed as a language for the design of interactive information systems. It includes facilities for the management of relational databases, a means for the specification of semantic integrity constraints and a mechanism for exception handling. Using the concepts of class, property, and the IS-A relationship, these facilities are integrated into a single language.

2.2.2 Data Dictionaries

Considerable improvements have been made in the disciplined way in which we design and code application programs. However these gains have been primarily in the area of program or process definition. A disciplined way to effectively utilize the raw materials for our processes, the raw material of data, is still lacking [13].

When it comes to understanding the characteristics and relationships of the data itself we are still relative novices. We find ourselves attempting to build high quality systems while ignoring the characteristics of our data. (high quality systems are defined here as systems that are well structured, user friendly, and easily maintained and expanded.) Only recently have attempts been made to define and document information about data, and concentrate on ways to use this data efficiently, effectively, and consistently.

The data dictionary has been defined by A. F. Cardenas [12] as "a centralized repository of data about data. It is in fact a database about the databases managed by the data dictionary software package and/or the generalized Database Management System (GDBMS)." Cardenas goes on to say that the term "data dictionary" has evolved with database development as a repository of data for direct use by the database administrator and users, as well as the GDBMS itself.

The typical data dictionary package contains a language for defining entries to the dictionary called a definition language; a language for inserting, modifying and deleting entries called a manipulation language; a means for validating the inputs to the dictionary; and a report program generator for preparing reports.

There is no industry-wide standard for data dictionaries. They have been implemented in two basic forms. The first form consists of those data dictionaries developed to nearly stand alone. That is, the organization that uses this type of dictionary does not necessarily have to use it with a GDBMS. These products have at the same time been provided with interfaces for use with the most popular GDBMS available.

The second form of data dictionary consists of those dictionaries developed for use with a specific GDBMS as an integral part of that GDBMS. The choice then is to use a data dictionary designed as an integral part of the GDBMS or one developed as an add-on for use with a specific GDBMS or with interfaces to work with your GDBMS.

Data dictionaries can contain information about and maintain relationships between its entries in most of the following categories:

field or data items
file or record type
database schema
subschema
physical database
transaction
source document
report
program
system
user [12]

Reports can be produced on any or all of the entries in a category by the report generator in the data dictionary package. These reports, which are a direct help to the database administrator, systems analyst, user, and programmer, can show an entries relationship both across and between categories. Here are descriptions of a few of the reports that can be produced by a typical data dictionary:

1. Listing of database names and all file names or record names making up each database.
2. Listing of file or record names and all data field names contained in each file.
3. Listing of field names and all attributes (e.g., data type, format) of each field.
4. Listing of field names and all alias names (synonyms) of each field.
5. Listing of field or record names and the passwords assigned to each record or field.
6. Listing of field names and names of all application programs which use each field.
7. Listing of system names and all application programs which make up each system. [12]

In general a list of all the other entries in each category to which a given entry is related may be made. In this way the data dictionary provides and manages a database about databases and related categories (e.g.,application programs).

All data definition entities are built on the foundation of the element definition. The entity definition provides the following information about a piece of data:

What the data element's name is. (ELEMENT NAME)
Who created the definition. (OWNED BY)
When the definition was created. (ENCODED)
The data element's aliases or synonyms. (ALIAS)
How you find the data element in the DD. (CATALOGED AS)
Relationship of the data element to other data. (NOTE)
The valid values for the element (VALID VALUES)
How the data element is stored. (FORMAT) [13]

The compilation of all the above information will obviously require considerable time spent in management, research and data entry. A number of benefits are derived from this expenditure of time and money.

First, by storing such knowledge in a data dictionary, and providing adequate backup and recovery to the dictionary a company can be safeguarded from the disaster of data documentation destruction. The data a company compiles about itself is an important company asset. Therefore all sensible data centers take precautions to safeguard that data. Should not the same precautions be taken with the information about a business's data? Could a company survive the catastrophe of having its documentation libraries destroyed?

Second, by providing "public access" to one's knowledge via the data dictionary, time and money can be saved that would otherwise be consumed in question and answer sessions. A programmer's or system analyst's knowledge has tended to be the private property of each individual. This knowledge should not be inaccessible to others. It should be placed in a data dictionary where it can be used by all members of an organization.

Third, the data dictionary can be a communications tool. In the past organizations have lacked a central pool of information that could be shared by all its members. The data dictionary can act as a central repository of information that can be accessed by all departments of a company.

Fourth, the data dictionary can reduce and guard against data redundancy. In the process of filtering and loading data from existing systems into a data dictionary it has been found that on the average there are 20 different data names for each unique data element in a system [13]. This is known as reference redundancy. Through the inclusion of automated dictionary search capabilities the data dictionary can be used as a tool to enforce standards of naming consistency thus restricting the admittance of redundant names into a system.

A second form of redundancy is format redundancy which occurs when there are variations of the format and length of a data element. Through the use of the data dictionary these format variations on a given data element can be exposed and catalogued.

Group redundancy is a third form of data redundancy. It occurs when a non-essential data name is created to identify one or more data elements. Such names only add to the number of data entries in a system and increase data definition complexity. Why add these entries when a comment inserted into the data definition would serve better?

When repetitious data names are used to identify multiple generations of the same data elements you have occurrence redundancy. Data element names containing numeric constants, such as "pay-increase-1-amount" are likely candidates for occurrence redundancy.

A fifth form of redundancy is definition redundancy. It occurs when a data element is used for more than one purpose. This form of redundancy creates excessive linkage between data stores and processes and complicates the design of the system. Data should be designed with a single function and purpose.

One last form of redundancy is storage redundancy, which occurs when the same data element is stored in more than one place. Storage redundancy can be justified in some instances. The most common such instance is when you are dealing with distributed processing environments or distributed databases. Unjustified redundancy is a liability to any system. Multiple definitions of a piece of data require multiple processes to update and safeguard this data to ensure consistency through all data stores. The data resource can not be worn out or depleted so why not recycle data by using the same element in multiple processes?

Data redundancy will not, however, be prohibited by the data dictionary itself. The dictionary can in fact accommodate data variations through the use of aliases and data element versions. The tools the dictionary supplies however can detect and publish inconsistencies in data and this itself helps inhibit unnecessary data variations.

The data dictionary can make the systems easier to maintain, more compact, and less expensive by allowing more effective use of the data.

The data dictionary can be thought of as a glossary of definitions. A data dictionary is not unlike an English language dictionary; it contains a glossary of terms about a company or project. As such it can be an invaluable tool in the training of new employees in the data processing and user areas.

In the areas of system development and maintenance the data dictionary can be used to centralize the control of program data definitions (e.g., file, record, and segment definitions for various programming languages). This helps to ensure consistency of data use and inhibit data redundancy. Used in this way the dictionary becomes an effective tool in change control management. This makes the data dictionary very effective as a tool to support structured analysis and design. Furthermore by enforcing consistency in data naming and format variation, program maintenance costs can be significantly reduced.

System documentation has historically tended to be inadequate because little or no time is allocated to this important task during project development, and often the documentation that exists does not accurately reflect how the system actually works. For these reasons little faith is given to the documentation of programs by analysts and programmers.

The data dictionary provides "living", perpetual documentation that is available to anyone who has access to a computer terminal. Through the use of the data dictionary's automated search and cross-referencing tools the user can span systems, multiple programs, report definitions, databases and any other entity type.

Because the data dictionary contains documentation about the data definitions, the reliability of documentation concerning the data used in a system can be greatly improved. By generating the source program's data definitions using the dictionary the analyst actually derives the data portion of the program from the documentation. The accuracy of the data documentation is thus guaranteed by the direct link between the documentation and the system definition.

Recently techniques have been developed to utilize the end user staff in helping data processing employees in the development of automated systems. These techniques include fourth generation languages, and user friendly inquiry and reporting tools. The data dictionary is another tool which can increase user effectiveness in system development. The process of system development still relies heavily on limited data processing resources. By utilizing the data dictionary along with other modern development aids the DP/user staff workload can be brought into balance. A development process, suggested by Durell [13], would work as follows:

> "1. With the assistance of the data administration staff, the user defines the data elements used to perform their day-to-day business functions. These data elements would be collected from all of the input forms and reports used in the end-users daily operations. These data elements would, in fact, compromise the majority of all the data processing system data elements. The only other data elements needed would be those necessary to define the system

> and program controls. These data elements would later be defined by the data processing staff during the design phase.
> 2. Using the data characteristics and relationships defined in the data dictionary, a logical database model is created.
> 3. Using the logical database model, and applying to it the physical constraints of the hardware and operating system, the database administration staff builds a physical database.
> 4. Depending upon the complexity of the process and the interface of this process with the existing systems, each process is coded by either the user or data processing staff. The coding process may use traditional languages or user-friendly procedureless languages."

The user, by defining the characteristics, relationships and editing criteria of the data, becomes directly involved in the design of the system. This saves DP staff time in data definition and gives the user more control over system design. This makes the data dictionary a tool to more effectively utilize the talents of both the user and data processing personnel.

A means of using a data dictionary to assist in enterprise analysis, which is related to and could extend Durell's development process, is presented by Sakamoto and Ball [22]. The use of IBM's DB/DC Data Dictionary as a means of providing computer assistance to the Business Systems Planning (BSP) methodology was shown to provide the necessary requirements for capturing and subsequently reporting of BSP study data.

BSP provides a structured means of assisting a business in the establishment of an information systems plan that satisfies both near and long term information needs. Techniques used in BSP include bottom-up systems implementation, top-down analysis and planning, managing data as a corporate resource, and orientation around business activities. A study team to collect and analyse the data required to run the business is the center of the BSP. This study team collects documented facts about the business and then organizes, abstracts, and analyzes these facts. Members of top management enhance these facts explaining the business and adding those points not usually documented. The study then progresses to identifying the major activities and decision processes in the business. Management then assists in validating and enlarging upon the facts gathered and analyzed. The consolidation and comparison of the data from all sources ends the analysis.

It was determined that the use of a data dictionary would provide a number of benefits to this process. Specifically the use of the data dictionary: (1) allows the storage of large quantities of data making the manipulation and analysis of complex relationships manageable, (2) allows each piece of information to be stored only once helping to reduce errors and maintain analysis consistency, (3) lessens the possibility of information loss, (4) provides a connection between the end of a BSP study and subsequent data management and information systems activities, and (5) ongoing BSP-like studies are facilitated.

Sakamoto and Ball [22] list the minimum requirements for this tool as:

A means of describing different kinds of information related enterprise phenomena.

A means of storing the descriptions in a database in a form that is computer processable.

A means of changing, adding, and deleting the descriptive data in the database.

A means to analyze and query the study data in the database.

A well designed data dictionary such as the IBM DB/DC Data Dictionary was found to supply most of these requirements and the it should be possible to provide the remainder.

The above benefits are just some of the ways that the data dictionary can help in system design and management. The data dictionary is not however the answer to all the information ills of a company. Rigorous data administration standards must be applied when data is defined. Otherwise the dictionary itself will reflect the same inefficiencies and inconsistencies as the data that is to be improved.

As the dictionary increases in size it also increases in value. The data dictionary information pool increases in value as its ability to do global cross-referencing and multiple systems inquiries increases. The main advantage of the data dictionary lies in its ability to assist in the discipline of data design. It increases the efficient and proficient use of data.

2.2.3 Related Topics

There are two topics that are important to the understanding and success of this report. These topics are the extension and intension of the database model, and the universal relation assumption as it relates to the database or "business world" model.

2.2.3.1 Extension and Intension

In the course of reading and discussing database theory and database management systems it is often found that there is confusion over just what an individual means when talking about some particular point involving the database scheme or model. Often the writer(s) or speaker(s) will confuse intension with extension and spend a great deal of time worrying over problems that are often purely semantic in nature. They will slip back and forth between the two without ever clarifying whether they are discussing the intension or the extension of the database. This can lead to the implementation of a databases extension before the users intentions have been fully ascertained and verified.

To hopefully avoid any such confusion here, the two terms (intension and extension) are defined and their meanings as they apply to and are implemented in a database are given.

The Random House College Dictionary defines intension and extension, for the purposes of this paper, as follows:

> Extension: 9. Also called extent. Logic. The class of things to which a term is applicable
>
> Intension: 5. Logic. the set of attributes belonging to anything to which a given term is correctly applied; connotation; comprehension.

With these definitions as a base, I have defined the database extension and intension as follows:

> Database Extension - an instance of the physical database which obeys the intension of the database as set down by the database administrator.

Database Intension - a model of the relations, data dependencies and uses of the user's data based on the user's descriptions and purpose.

Keeping these definitions for the database extension and intension in mind should alleviate some of the problems that occur when discussing databases and their models.

2.2.3.2 The Universal Relation Assumption

The second topic necessary for a solution of the problems, to be discussed later, is the universal relation assumption. The universal relation assumption as given by W. Kent [18] states that for a given set S of relations under consideration, there exists in principle a single universal relation U such that each relation in the set S is a projection of U.

This means, of course, that each column name in U is unique and identical column names occurring in several relations of set S are projections of the same column in U and mean the same thing. In other words, every occurrence of a given column name across a set of relations refers to the same set of data.

2.3 Software Engineering

Software Engineering can be defined as the the practical application of scientific knowledge in the planning, design, and construction of computer programs and the associated documentation required to develop, operate, and maintain them [31]. D. Ross, et al. [44], states that software engineering clearly implies at least the disciplined and skillful use of suitable software development tools and methods, as well as a sound understanding of certain principles. Ross goes on to discuss certain issues of software engineering in terms of four fundamental goals. These goals are:

*Modifiability > Able to change or evolve easily.

*Understandability > Easily understood by others who did not have any involvement in developing the application originally.

*Reliability > Must work as intended.

*Efficiency > Easy to use, efficient operation (must meet other goals first). [44]

The process of attaining these goals is affected by seven principles. These principles consist of :

*Abstraction > identify essential properties; characteristics common to superficially different entities.

*Hiding > Making the inessential inaccessible or unavailable.

*Modularity > Dividing things into modules, logical functional units, typically some sort of structure and purpose to divisions.

*Localization > Similar things together.

*Confirmability > Verify that something is happening; want to see what is going on. Build in ability to see what has been produced.

*Completeness > Providing complete (whole) set of items, with ability to change same.

*Uniformity > Standardization, data entered in a standard form, standard interface with user. [44]

The process of planning, designing, constructing, operating, and maintaining software is known as the software development life cycle or process. The software development process when properly implemented will help achieve the above goals in terms of the seven principles listed.

2.3.1 The Software Development Process

There are many different descriptions of the software development process. Two typical descriptions are given by M. Zelkowitz [52] and K. Orr [42]. Zelkowitz describes the software development process as consisting of the following stages: (1) requirements analysis, (2) requirements specification, (3) design, (4) coding, (5) testing, and (6) operation and maintenance. Orr defines The development process as consisting of the following eight steps: (1) plan, (2) define, (3) design, (4) construct and test, (5) install, (6) operate, (7) use, and (8) evaluate.

The process here is based mainly on the process as described by Orr for his systems development methodology. To Orr's description, certain explanations of process steps, as described by Zelkowitz [52], have been added to help give a better understanding of the process. In addition, Orr's process is shortened to five steps with the last four steps (install through evaluate) combined into one final step.

Orr stresses structure in the development process, calling it structured systems development. An application or information system is defined as a collection of related entities and transactions that makeup an essential, feasible, functional, workable, correct, and operational model within an organization. Each term indicates the most important characteristic of a phase of the systems life cycle.

The first step in the software engineering process is often referred to as requirements analysis. The Random House College Dictionary defines analysis as the separating of an abstract entity into its constituent elements; a method of studying the nature of something or of determining its essential features and their relations. Determining the essential features of a system is certainly necessary for the development of a complete and accurate system that meets all its requirements.

Zelkowitz defines this first step as the definition of the requirements for an acceptable solution to the problem. He goes on to state that this step focuses on the interface between the tool and the people who need to use it. It contributes to the best solution by aiding in understanding the problem and the trade-offs among conflicting constraints. It also should distinguish hard requirements and optional features, and determines the resources needed to implement the system.

Orr divides this phase of systems development into logical and physical subphases. Logical systems planning is characterized by the essentiality of the problem. Essentiality [42] is defined in terms of identifying: problems, users, uses, and the applications context. The key question at this stage is: is this application really necessary? In other words, an essential application is one that is necessary to the operation of the organization, enables the exploitation of major opportunities or solves major problems, and from the organizational standpoint is a major priority.

The second half of the planning phase is physical systems planning. It is characterized by application feasibility. An application is feasible if, given the organizational environment, it is judged economically, technically, and organizationally possible. When analyzed many applications prove to be deficient in one or more of the areas of possibility. Feasibility, in Orr's structured systems development process is defined in term of economic, technological, and organizational risk (benefit).

The primary focus of the systems planning process is determining which of the many applications that could be developed are actually carried forward to the requirements definition phase. Orr suggests that the document that results from this phase should answer at least the following three questions:

*What are the real (as opposed to apparent) problems or opportunities the system is supposed to affect?

*Who are the users and what will the primary uses of the system be?

*What is the context and preliminary scope of the system? [42]

In addition, at least preliminary answers should be provided for two other questions. They are:

*What resources are needed to implement the proposed system?

*How will progress towards systems completion be controlled and monitored? [52]

The document that results from this systems planning phase represents the input for the next step in the software development process.

The second step in the process is termed requirements specification or definition. Requirements definition deals with the definition of what the system is supposed to do and what its inputs and outputs are. Here again, Orr divides the task into two subphases. The first phase of this step is logical requirements definition. It deals with the functionality of the application. An application is functional if it does what it is supposed to do and no more. If it works well and operates in a consistent manner for a variety of operators, managers and users, it is a functional system. In its fullest sense, functional means ignoring the internal procedures and organization structure. Functions, in Orr's structured systems development process, are defined in term of functional flow and steps, context, cycles, decision control feedback, scope, and outputs.

The second phase of requirements definition is physical requirements definition. The main characteristic it is concerned with is workability [42]. An application that is workable can be supported economically, technically and operationally, and can be operated by personnel of the organization. A workable system is concerned with such factors as response time, human factors engineering, volumes, and many physical constraints. Workability, in structured systems development, is defined in terms of alternative solutions, constraints, recommended solutions and benefits/risks.

In order to produce a system that will be used within the organization, requirements definition concentrates upon applying physical constraints to the functional requirements. The document that results from this step represents a functional, workable application definition, and is the input to the design phase of the systems development process.

Zelkowitz defines the design phase as the stage at which the algorithms called for in the requirements definition are developed, and the over all structure of the system takes shape. Modularity of design is stressed.

In Orr's structured systems development process design is divided, like the two previous phases, into two subphases. These subphases consist of a logical design stage, which exhibits correctness as its main characteristic, and a physical design stage, which exhibits operationality as its main characteristic.

Orr states that "the test of logical design is at all time the ability to prove correctness" [42]. An application that produces, at the required times, according to the required calculations and decision rules, the correct results is a correct application. Whether the application produces the correct answers all the time is the first and foremost technical issue in design. A strong emphasis is placed upon defining the logical correctness of systems, based on the results of the requirements definition phase, in structured systems development. The correctness of an application is defined in terms of its logical specifications.

Physical design deals with the incorporation into a system of those features that will make the system efficient to operate, easy to run, capable of recovery, etc. These application features, that make the application flexible, robust, efficient to operate, and maintainable, give an application the characteristic of operationality that Orr stresses for the physical design subphase. This characteristic of operationality is defined in terms of a set of physical specifications. The logical and physical specifications provide what can be thought of as a blueprint for system construction and testing, the fourth phase of systems development.

The implementation of the set of specifications or blueprints from the design phase is carried out in the fourth phase of the systems development process. This fourth phase is called variously construction and testing (Orr), coding (Zelkowitz), or implementation and testing. Testing should proceed concurrently with system construction. In fact, preparation for testing should start in the design phase and possibly earlier.

Construction is the process of actually producing the software satisfying the specifications or blueprints produced in the design phase. The software is produced using a specified programming language. It is important to produce a program that is structured and modular so structured programming techniques should be stressed in system construction. The use of high level languages and structured programming techniques will help simplify the construction and testing of the system. The implementation of a modular program will be important in later stages of the systems life cycle as it will make maintenance and modification easier.

As emphasized above, testing should proceed concurrently with the construction of the system. Zelkowitz suggests that a plan for testing the system should be designed early in the development process and that the data used to test the system should be specified during the design phase of the project. The actual testing of the system under construction should start with the production of the first systems modules.

Zelkowitz suggests dividing testing into three distinct operations. This division aids in the implementation of test procedures early in the construction process.

The first stage of testing will consist of testing individual modules for correctness and is called module or unit testing. The second stage of testing consists of testing groups of modules together and eventually produces a completely tested system. This operation is called integration testing. The last stage of testing before the system is ready for operation and use involves an out side group testing the completed system. This operation is known as systems testing.

The result of this fourth stage of the systems development process is the completed and tested system software along with documentation of the software explaining the use and workings of the system, and the tests done and their results.

Once the system has finished the testing process it is ready for the next phase of the systems development process. Zelkowitz call this phase operation and maintenance. Orr divides this phase into four separate steps, installation, operation, use, and evaluation.

Orr's four steps are contained in the phase which, as Zelkowitz does, shall be referred to as operation and maintenance. The operation and maintenance of a system occurs simultaneously. For this reason Orr's last three steps are concurrent. That is they occur at the same time. The installation of the system occurs at the start of this phase and may not be required in the future of the system.

Installation is the process of bringing the the system "on-line", making it available for use by the users. This is followed by systems operation and use which should occur concurrently with the evaluation of the system.

As the system is used it should be evaluated to uncover any system faults or shortcomings. This process is part of system maintenance. It ends when the evaluations carried out determine that the program is no longer needed or that further modifications or corrections are not economically feasible and a complete new system should be developed.

When a fault or shortcoming in the system is uncovered, by the evaluation, the system is repaired or updated (modified) to correct the problem. Updating and repairing the system is, therefore, the process of modifying the system to meet requested changes in the delivered system, and the correction of errors missed in systems testing.

Modification of the system should be handled in the same manner as the development of a new system. That is, the modification should go through all the steps of systems development from planning through operations and maintenance. It has been suggested [40] that this last phase could be renamed the evolution phase.

The systems development process outlined above consists of five phases: systems planning or requirements analysis, requirements definition, design, construction and testing, and operation and maintenance. Following this process and using selected software engineering tools should aid in the attainment of the four goals of software engineering outlined at the beginning of this discussion.

2.3.2 Tools and Methodologies

Methodologies exist that can aid all of the phases of the software development process. Methodologies and tools exist to help in requirements analysis, requirements definition, design, construction and testing, and operation and maintenance. They are highly important in reaching the goals of software system development. Not all these methods or tools were originally developed for software development, never-the-less they can be of direct help in the development process and will be discussed in the following chapter. At this time certain methods that pertain to software engineering shall be examined. The methodologies and tools to be examined include development support systems, requirements definition languages, program libraries and guides, reuse of old code or program modules, structured programming techniques, and tools to develop and improve tools.

Even the use of a set of well constructed tools (a toolbox) and adherence to a few basic methodologies such as reuse of old code will not guarantee the successful development of a system.

They must be used in connection with or patterned after a standardized structured software development process such as the one outlined earlier. The methodology and software tools that are developed for use with it are together a development support system [40].

This development support system consists of two parts: (1) a model representing a particular view of the development process, and a methodology for proceeding with system development, and (2) software tools based on the model, and supporting the development process according to the methodology used [40].

The logical place to begin the discussion of tools and methodologies used in supporting the development process is with planning, the first step in the development process. Structured planning is a current area of research that holds great promise. It deals with the careful planning of systems before applications development. Systems planning in software development involves the analysis of the business' functions, the data these functions need, and the information processes needed to maintain the data of an organization. This overlaps with what has traditionally been management or enterprise analysis and services as a vehicle for communicating information requirements from the end user to the data processing function. Enterprise analysis, as used here, is the analysis of a businesses goals and policies to determine its information requirements. An example of a tool that serves this function is the Business Information Control Study (BICS) and an extension to it called REQGEN [38].

BICS is an enterprise analysis methodology That has been implemented in EAS-E, which is an application development system based on the entity, attribute, and set (EAS) view of systems modeling. An extension to BICS called REQGEN produces requirements definitions for all of the data processing requirements of an organization for which an analysis is made. It generates specifications for the processes required to maintain the data, lists the events that trigger each process and generates the data definitions.

Other enterprise or management analysis methodologies exist. Some of them have been used in the data processing effort outside software development others, have yet to be exploited in this manner. The use of enterprise analysis methodologies, such as BICS, in software development should enhance the ability of users to communicate their needs to the systems analysts and at the same time enhance the analysts' ability to produce systems that are essential, feasible, functional, workable, correct, and operational.

The main tool in the requirements definition phase is the Requirements Definition Language (RDL), which is used for the specification or definition of data or algorithmic structures. A good RDL, according to Orr [42], should exhibits the following principle characteristics:

Simple > Contain a minimum of basic structured representations.

Logical > Notation must be directly related to basic logical primitives.

Complete > The vast majority of important relationships and structures may be described by the notation's set of structures.

Graphical > The solution is "pictured" by the definition language as much as possible. The RDL delineates the context as clearly as possible providing a significant step in improving communications between the user and the vendor.

According to Orr, if the definitional language is to model the real world adequately it must be capable of representing the following logical structures: sequence, alternation (selection), repetition, hierarchy, concurrency, and recursion. Some useful extensions to these logical structures can be added to the language. These extensions include: conventions for arithmetic operators, names, labels, keys, and the use of operators with repetitive sets. It also useful if the language is capable of representing both processes and data. That is it is able to define the flow of control and the flow of access respectively [42].

Some examples of requirement definition languages are PSL/PSA [46], SA [43], SREM [29], REQGEN [38], Warnier/Orr Diagrams [42], and a framework for requirements models (RMF) [36]. These languages provide the input for the next stage of systems development. A well structured requirements definition will ease the system designers task by making the use of certain design techniques simpler.

In design, the next phase of the software development process, there are a number of techniques and tools that can help produce a correct and operational design. Some topics in this area that are of importance to this paper are structured or modular design and the reuse of old code.

There are many reasons for using structured design techniques in the software development process. The more important reasons are that it allows the application to be developed one function or procedure at a time, makes system construction and testing easier, and makes systems maintenance easier as well. One more reason for using modular design is that it makes possible or at least easier the reuse of old code.

In database management one key issue is the reduction of data redundancy. The redundancy issue should be no less important in system development. In systems development redundancy refers to the redesigning and reimplementation of processes that have been developed before. The solution for this is to reuse old code. The keys to the reuse of old code or processes are some method of keeping track of the tools and programs developed and used by an organization, and a set of well structured and complete requirements definitions. The latter is provided by a structured requirements definition language and one way of providing the former is the library reference guide and program library.

The use of a library of programming tools and routines can help reduce programming redundancy and increase systems development efficiency. The effective use of this library requires knowledge of the library's contents and how to use those contents. This library should also be easy to use. A Library Reference Guide (LRG) can help attain these goals.

A LRG can consist of an on-line query program, a traditional manual or a combination of the two. If the LRG is up to date and well designed it will be easy to use and will help programmers save time. The LRG has two functions: (1) help system developers find and (2) use a routine. It should be indexed by the tool or routine's name, and the task or function performed. Each entry should include a functional description of the routine. This functional description should include:

what it does

setup required if any

inputs and outputs

data altered or destroyed

data routine entered or last time altered. [47]

Each routine's description should include all that the system developers need to know about that routine and should be complete in and of itself. By providing this information in the LRG the systems designer can use it to identify those functions or processes defined in the requirements definition document that have already been developed and incorporate them into the systems design.

The guide should be organized according to the functions performed by the routines (group search routines, stack routines etc., together). All entries should be cross referenced by name, function(s) performed, and subject (file, flag, stack, etc.) [47]. A search method that is fast, flexible and easy to use should be provided. Facilities for searching on the entries function(s), name, and subject should be included at the very least. It is desirable to have the ability to search on more than one key word at a time.

The library guide should be task oriented. For complex routines give a general description first then detail each option separately. The guide should be strictly functional. If explanations on how to use a routine are needed the programmer should be referred to a users guide that will explain the routines in more detail. It should be easy to update and add new routine descriptions to the LRG.

As the program library is dynamic, the LRG should also be dynamic. This will require a clear line of communication between the LRG manager and users if entries are to be kept up to date. When the LRG is updated any user manual affected should be updated at the same time.

The Library Reference Guide, as described, gives systems developers quick access to the information they need thus helping increase their efficiency and at the same time reduce programming redundancy. The guide can be a valuable software development tool when properly implemented and can help reduce the cost of software development.

Related to the reuse of code are two tools called adaptable applications software and foundation software. Adaptable applications software consists of a set of software that can be installed and customized to provide a complete system that when finished meets an organizations unique requirements [51]. Foundation software consists of an integrated environment of standard packages and custom modules that provides common services to the development and operation of applications software [33].

Often the use of off-the-shelf software requires major undesirable and traumatic business and operational concessions. The use of adaptable applications software can alleviate these problems by providing low risk, low cost, fully functional and customized software. As its name implies this software should be adaptive and easy to work with allowing all desired functions, features, and requirements to be accommodated. Some of the benefits of using adaptive software are as follows:

1. The need to reinvent the wheel is eliminated.
2. The organization's unique requirements can be met.
3. The risks are low.
4. The costs are a fraction of what they otherwise would be.
5. The time required is similarly reduced. [51]

Applications software as well as adaptable software is provided a structured, standardized, and simplified view of the outside world by foundation software [33]. The economics of development, operation, and maintenance of large systems is dramatically improved and the risk of development is reduced by the use of foundation software.

Foundation software isolates applications software from changes in the computer systems technical environment and makes the technical components easier to use, thus increasing productivity throughout the application softwares life cycle. Besides the technical environment Curtis [33] divides large scale systems into two other architectural levels, applications software and foundation software. The technical environment is at the core of the system. Around it is the foundation software which is between the technical environment and the applications software. The system can be thought of as an onion with the technical environment at the core and the applications software as its outer-most layer.

The technical environment consists of the operating system, DBMS, network architecture, etc. The application software level, which includes adaptable applications software, contains all the specific substantive functions that relate to the application problem at hand and is the functional core of the applications system. An interface, between the detailed considerations required by each component of the technical environment and the applications software, is provided by the foundation software level. The control, communications, and standard programming services of the technical environment are directly used by the foundation software.

Foundation software consists of three types of components: custom interface modules which provide efficient standardized utilization of the technical environment's more complex components, packaged software such as report generators, enquiry packages, etc., which are usually integrated into the foundation software with custom modules, and common modules which are processing functions required frequently throughout the applications software.

The functions to be provided by the foundation software should be determined within the context of the application design characteristics, the organizational environment of the system development effort, and the constraints of the technical environment. Some of the functions and features provided by the foundation software are: on-line user management, input and output management, application data management, network management, system management, office automation facilities, and technical-environment enhancements [33].

Foundation software can provide benefits throughout the system development process. The major benefits begin with system design and extend through system support. Within system design foundation software can be viewed as an extension of structured design methodology as it results in a high level of modularization. Standard, reuseable software provides many functions and common functions are designed only once.

During system development and implementation foundation software allows senior technical staff to concentrate their efforts in high-payoff areas and allows less sophisticated staff to develop technically sophisticated applications. In addition foundation software isolates application programmers from changes in the technical environment.

Using foundation software, applications programs are isolated, during system support, from the effects of rapid changes in technology. Because new applications functions can be added with minimal affect on the existing system, the application can be changed to meet evolving user requirements more easily and cost effectively.

Through the use of adaptable applications software, for the application specific functions, and foundation software, for the interface to the technical environment and to provide common tasks, system development risk, time, and cost could be greatly reduced. Furthermore, as stated the use of these techniques enhances the use of structured design and implementation methodologies and can aid in system maintenance.

Some other techniques or tools that aid in the implementation and testing of structured or modular programs are structured programming languages, preprocessors and as mentioned earlier modular design.

Structured languages, such as Pascal and Modula 2 that provide strong data typing and subprograms provide the means to implement an application design as a set of modules. These modules are often referred to as procedures, subroutines or functions. Being able to implement each module individually allows the system developers to test each module first individually then in related groups. When the design phase of the development process uses structured or modular techniques the use of structured languages is further enhanced and systems implementation is made easier.

Preprocessors are programs designed to aid in systems testing and implementation (an example is described in 49). They allow the programmer to do such tasks as check a program module's syntax, or support data encapsulation and modular system development before actually compiling or interpreting, or even coding the program or its individual modules.

Some basic software development tools and methodologies have been presented here. In addition, two methods for developing software development tools can be added to them. The two methods are: 'A Forms Based Approach to Human Engineering Methodologies' [39] and 'Specification Meta Systems' [34].

Software engineering methodologies can be improved through the use of a forms based interface. By using a forms based interface, standardized methodologies may be enforced and software quality increased. The advantages of using a forms based interface for a software engineering environment can be shown by focusing on the design of forms, on the impact of forms on the software engineering process, and the improved tool support facilitated by the standardization achieved by forms.

By using blank forms and help forms software development methodologies can be precisely described. The syntax concepts of a methodology are enforced by the blank forms and the help forms provide the programmer with information on how to fill in the forms. Forms standardize the documentation of design decisions and the placement of documentation. Using forms to standardize the enforcement of a methodology facilitates the design of semiautomatic tool support. Finally the design of a customizable environment which evolves as the programmer's needs to continually provide optimum support for themselves evolves is facilitated by the forms based approach as well.

Forms have certain general properties. A form usually contains headings that describe what is to be entered in the blanks. Underneath each blank are annotations which provide information on how to fill in the blank. These annotations provide help information and if they get too long a separate help form may be provided.

The notation that must be used to fill in a blank form may be formal or informal, programmer or programming language oriented. In addition graphical symbols and a text other than English text may be used.

A variety of headings, some requiring machine oriented detail in some programming language, may be contained on a form.

Two general heading types are the programming oriented heading or p-heading, mentioned above, and the documentation or d-heading which provides for headings where human oriented design decisions may be recorded.

The proper designing of forms for enforcing a methodology requires that the designer consider the content and placement of the d-headings and p-headings, and the contents of the help information. The rationale for the design of forms is to provide a layered collection of modules, make relevant documentation visible and hide irrelevant detail, highlight relevant documentation, and provide for the design of data abstractions [39].

Tools and methodologies are two complementary ways for supporting the software life cycle [39]. Tools can be developed to support a methodology as it becomes precise. Syntactic concepts which may be presented to the programmer by means of forms are present in methodologies. These forms serve to precisely identify the techniques for filling in the blank forms, thus using the syntactic concepts of the methodology. The early identification of module properties that are externally visible is emphasized by forms. Through the use of specification techniques forms provide support for the design of data abstractions, and forms provide a techniques for programming in the large by enforcing a layered architecture.

The form-based approach can be used in all phases of the software life cycle ranging from the requirements phase to the coding phase [39]. The uniformity that results from the use of forms in recording software related information facilitates more powerful tool support.

The actual use of forms has produced two major conclusions. These conclusions are that forms standardize documentation and secondly that forms encourage the use of modules [39].

A number of conclusions can be drawn about the use of forms. First, an interface based on forms facilitates the standardized enforcement of methodologies. Second, by carefully designing these forms, a programmer's behavior may be altered in desirable ways. Lastly, better tool support can be provided by organizing software related information using forms.

Specification meta systems are a practical method for the construction of general purpose computer aids for specification, analysis, and documentation in software systems development [34]. Meta systems consist of two levels, these are the designer's level, called the meta level, which requires special training to use, and the target level or user level which is intended to be usable by an untrained individual. Two level meta systems give the effectiveness and power of a narrow target domain (level one) while providing flexibility through the abstraction and manipulation of the target level (level two).

Meta approaches are general purpose methods with language definition facilities that have become feasible with respect to computer aided specification systems. Problem oriented specification languages are effective only in fairly restricted areas and specification tools tend to have very similar structures and underlying principles, thus giving impetus toward meta driven systems.

In applying a given computer aided methodology, it is often hard to match the predefined concepts of a particular language with those that derive from the problem to be solved. However since it turns out that the various software tools supporting particular descriptive languages have essentially the same structure, it is possible to construct a set of computer aided tools, independently of the application language, that can then be supplemented by mechanisms that aid language definition [34].

The same capabilities as provided by conventional computer aided methodologies can be built into meta systems in this way. Meta systems, being free of the language restrictions of a particular descriptive language, allow the user to designate his own conceptual world and language terms. The language independent set of tools should be built upon a general scheme onto which each potential descriptive language can be mapped.

The second main component of a meta system is the meta interpreter. The generation of the meta database that controls the operation of the language independent part during the consumption of specifications is its major function [34]. It can also have additional functions such as generating user manuals describing the language given at the meta level.

The foundation of meta systems have been well developed recently in both semantic processing and data abstractions. The foundations of two level meta systems are theoretically much deeper than those of problem oriented languages [34].

This chapter has examined a number of computer science topics that are pertinent to the goal of this paper. These topics included:

Artificial Intelligence. a description of expert systems, and knowledge bases, and a look at some generic expert tasks,

Database management systems. a brief description of what a database management system is, a look at the three major database data models and what data definition and manipulation language are, and a description of the data dictionary and it benefits,

Software engineering. a description of the software development process and a survey of some methodologies and tools used in software development.

The next chapter will discuss the similarities of these three main areas and the ways in which various methodologies and tools from the different areas can be used to enhance methodologies and tools in the other areas.

Chapter 3
An Overview of AI, DBMS, and SE Similarities and Cross Applications

All three of the areas under discussion have certain characteristics in common. They are all involved in systems development, and all three deal with the processing of data (information or knowledge). In fact all of data processing and computer science is concerned with the manipulation of data in one way or another. What is intended here is to show how the techniques or methodologies of one area could be or are being used in the other two areas to support or enhance systems development and the processing of data.

3.1 AI in DBMS

The application of artificial intelligence to database management has been done in three ways:

AI techniques employed to improve the user interface,

AI techniques have made it possible to increase the efficiency of the DBMS,

Similarities between AI and DBMS may be exploited to extend the capabilities of the DBMS allowing it to answer different kinds of questions about the data and about itself [2].

Improvements in the user interface provide for greater data independence. It is not necessary for the user to be aware of the actual organization of the database when interacting with the system. The data manipulation or query language for accessing the database is the dominate feature of the user interface. By developing natural language interfaces for query languages the usefulness of a DBMS to users can be enhanced.

Natural language processing is a major subarea of AI. Its use in DBMS introduces new problems. Possible problems include:

Users of DBMS tend to use abbreviated often ungrammatical requests for information as they become familiar with the system.

Misspelling of words by users cause problems.

As the information in the database is constantly changing the DBMS's parser may not have a complete lexicon of the words the user might use in his queries.

The user is often unaware of the structure of the database and its connection to the structure of the application domain [2]. (In a properly designed system the user should not need to know about the database's structure, or its connection to the structure of his application domain. This knowledge should reside within the scope of the systems responsibilities, i.e., the system is responsible for keeping track of the connection between the structure of the user's application domain and the structure of the database.)

Some natural language systems for database access that have been implemented are, LADDER (Sacerdoti, 1977), PLANES (Waltz, 1978), ROBOT (Harris, 1977), and TQA (Damerau,1979) [2].

Some of the features explored by these systems include, ability to handle elliptic input, correct spelling errors, handle pronouns, and use the database to look up unknown terms in the query. These systems function in different ways. However all of the systems use a grammar of the particular language and a parser to process the input in terms of the grammar. (See Handbook of AI vol. 1 chap. VII pg. 163-73 for further information.)

Automatic derivation of a natural language:

DBMSs have been designed to explore the usefulness of specific NL features in the natural language front end. In one such system, TED, developed by Hendrix and Lewis, the semiautomatically derived NL front end makes use of the database schema and a dialogue it conducts with the user to produce the NL interface rather than requiring extensive programming. This process results in a grammar for the particular language.

Questions about the English expression of concepts from the database, connections between files in the database, ranges of certain attributes, and so forth, may be included in the dialogue. This process produces a language capability that is in general slightly more restricted than a manually created one.

Cooperative responses:

Natural language queries that are answered using a direct literal interpretation of the query may not result in the correct answer and could be misleading. If the system could infer from the users query that the state of the database violates the user's presumptions the system could formulate a corrective, indirect response that is more informative than the formally correct response. This is called cooperative response. The COOP system (Kaplan, 1979), was designed to provide such a response.

In incremental query formulation the system helps the user to formulate his query by asking directed questions of the user that help the system to generate an appropriate final form for the user's initial query. A system accomplishes this task by accepting a partially specified query in somewhat rough form which it then attempts to complete by (1) entering into a clarification dialogue with the user to clarify ambiguities in the query and (2) attempts to fill in "gaps" in the initial query specification by initiating a dialogue with the user.

More sophisticated interfaces:

The DBMS user should be able to interact with the system not only in terms of the concepts and terminology covered by the database per se but also in terms of the enterprise being modeled. The user should be able to ask question about the database, such as what kinds of information are stored in it. This system could provide an interface between the user's needs and the resources of the computer system. The KLAUS system, currently under development at SRI International, (Haas and Hendrix 1980) is an example of such a system (Handbook of AI vol 1 pg 169-70 for more info.).

Query Optimization:

Two methods to improve the execution of queries are syntactic query optimization and semantic query optimization. Syntactic query optimization uses heuristics referring only to the structure of the database while semantic query optimization uses information (such as value constraints on attributes or limitations on entity relationships) about the actual contents of the database. (Handbook of AI vol. 1 pg 170-71)

Artificial Intelligence and Database Management are both concerned with the representation, retrieval and use of information. Extensions of the data model, which describe the structure of the database in the database scheme, and the use of formal logic to reveal restrictions on the ability of database systems to represent certain kinds of information or to perform certain operations are of particular interest.

Data models capable of representing aspects such as semantic constraints, generalized inference for query processing, and inheritance of properties between classes of objects in DBMSs have been developed based on the semantic network knowledge representation formalism. Specifying the data model and stating queries can be done using the network notation. An example of this is Sowa's conceptual graph model.

Two languages for designing databases that were discussed in the previous chapter, TAXIS and SDM, also make use of techniques for the specification of semantic data. TAXIS (see chap. 2) uses the semantic net representation formalism and also implements the principles of exception handling and data abstraction from programming language research. SDM, (semantic data model) was intended to facilitate the representation of information about the domain of application as well as database information. In particular relationships of abstraction aggregation and restriction could be used to characterize the relationships between the entities in a domain.

There are a number of restrictions present in DBMSs as compared to AI systems. Two of these restrictions are that current databases are typically incapable of representing many kinds of quantified information and disjunctive information, and second databases usually do not have extensive capabilities for inference. Quantified and disjunctive information can be handled in the query language but they can not be represented in the database. Using constraints and views certain forms of deduction can be performed but general inferencing capabilities are not available in current databases although there have been proposals to incorporate this capability for such purposes as query processing [2].

The purpose of the database data dictionary is to provide information to the DBMS users about the characteristics and relationships of the the data stored in the DBMS's database(s). The typical data dictionary contains information only about the entities in the different database schemes, such as what the data element's name is, and nothing about what the relationships between these entities are, or what processes use the different entities. By incorporating this missing information into the data dictionary and providing an easily used interface capable of making use of the information provided, the data dictionary can make the DBMS easier to maintain and allow greater and more efficient use of the information it contains. One possible way in which this might be done is by implementing the DD database as a knowledge base. This could be done using, for example, frames and scripts.

DBMS usually have the ability to use virtual data, that is data that is not physically present in the database and must be calculated or derived. Frames allow the attachment of a procedure to a frame slot for the purpose of calculating the value of that slot. Additionally frames can inherit information about their parent frames and show relationships between frames.

A great many possibilities could be opened up by using frames in implementing a DD database. Making the DDDB a knowledge base could increase the efficiency and power of the DML or query language. The generalization levels possible using frames make possible easier division of the knowledge base into various user views, having varying levels of details and still showing the relationships between the different user views. The use of frames could make the data dictionary a knowledge representation system, and the basis of a DBM expert system. Using this method of implementing the data dictionary would go a long way towards overcoming the restrictions, such as inferencing capabilities, that currently hamper existing DBMSs.

3.2 AI in Software Engineering

The main focus of software engineering is the development of software systems or applications programs. One of the main tasks of artificial intelligence is the development of expert systems. An expert system is, of course, first a program, a software system. The software development process, of software engineering, is not concerned with developing a particular type of system (i.e., a business application such as accounts receivable), it is concerned with providing a standard methodology through which any software system can be developed and maintained. It seeks to help the computer scientist, programmer or analyst plan, design, construct, test, operate and maintain the system that the individual is trying to develop. Obviously software engineering plays or should play a large role in The development of AI systems. At the same time AI can have a role in the software development process.

What kind of role can AI have in SE? Artificial Intelligence can be used in SE in many of the same ways it is used in DBMSs. This includes the use of AI techniques to improve the user interface to development tools, and increase the efficiency or ease-of-use of the tools and the development process.

Natural language interfaces could be developed to allow incremental specification development. The tool user could enter information in English and the system would interrogate the user about the the information it has been given asking such questions as what is this information for, how is it used, etcetera in order to clarify the user's intentions.

In the last chapter a number of generic expert tasks were discussed. These tasks consist of interpretation, diagnosis, monitoring, prediction, planning, and design. Looking back at those tasks, it is easy to see that some of them are contained in the software development process. It is only a short step to envisioning how these tasks could be used in software development.

One of the generic tasks listed above is planning, which includes the task of prediction. The first step of systems development is requirements analysis or planning. Also mentioned above is the use of natural language interfaces, which require the interpretation of user input. Analyzing an organization's operations and goals should be the first step in systems planning. The use of a natural language interface to interrogate an organizations employees would increase the efficiency of the planning process. By implementing the planning process as an expert system, efficiency could be further enhanced and more information could be made readily and easily available to systems developers.

The second phase of systems development, requirements definition, can be incorporated into the planning phase. It is properly the last half of what is usually considered the planning process and produces the end results of the planning process the requirements definitions. An expert system could be developed to process the information produced from the interrogation of employees of the organization using a well defined, proven requirements definition language as a base. The interface to the planning process would provide a means of checking the requirements produced by the system and of filling in or changing missing or erroneous data.

Design, as a task, has many of the same requirements as planning. By implementing the design process as an expert system the burden of coping with the complexity of the system being developed, and of coping with the interactions between systems modules is removed from the developer's responsibility. The expert system could be able to consider different design possibilities, and avoid points in the design space that are only locally optimal when modifying the design. The expert design system could also help assess the consequences of design decisions, explore different design possibilities and help in the integration of systems constraints with design choices.

With the ability to develop a workable, functional set of requirements definitions and a correct, operational design, the next step is to construct or implement the system. There already exist automatic program generators that work for narrow problem domains. With more work in understanding the first three phases of the software development process and corresponding work in developing expert systems to handle these tasks the scope of these program generators can be widened.

The last phases of the development process require diagnosis and monitoring. A monitoring system should be developed to analyze the results of a system's use and report any errors or faults in the system and aid in their correction. In fact Orr states that a systems methodology is really a system to build systems. As such, it contains all the characteristics of any other system. It should therefore be monitored and evaluated periodically just like any other system. This means, of course, that the development process should be monitored, and the results of this process put through a diagnosis procedure to uncover any problems and correct them. This procedure could be implemented as an expert system, both interactive and automatic, to allow systems developers to check specific systems functions or run complete systems checks as the system is being used.

3.3 Database Management in Software Engineering

In software engineering you have requirements definition languages that define a proposed systems data and programming (i.e. functions, activities, procedures) requirements. In database management systems you have data definition languages, that define the database scheme and subschemes (data models).

The major difference here is that data definition languages do not usually define information about the processes that manipulate the data they define, while requirements definition languages do. By recognizing this similarity, and including this information in the database schemes a great deal of knowledge previously unavailable to the database user can be made available. At the same time, this added information can give the database administrator better control over the database by giving added information on what processes use a given entity type, which one produces it, etcetera.

In the discussion of software engineering tools the library reference guide was discussed. If the tasks and functions of this guide are compared with those of the data dictionary it becomes apparent that there is a considerable overlap in the functions that theses two tools preform.

Data definition languages provide the inputs to the data dictionary. These inputs define only the systems data. Requirements definition languages can provide inputs for the LRG. That is, the RDL can provide the process or program module descriptions that the LRG uses as entries. The RDL also provides data definitions. In the discussion of data dictionaries, it was mentioned that the data dictionary could be used as a repository for program descriptions, as well as data definitions. Quite clearly the DDL and the RDL perform the same basic task for essentially the same purpose. That is, to define the data and program modules to the system, whether it be a software development system and its LRG or a DBMS.

Further similarities between these two tools is that they both: can aid in reducing redundancy (both programming and data), are a communications tool providing a central information pool, can help train new employees, provide central control of data and program module definitions thus ensuring consistency of use and inhibiting redundancy, and can help in system documentation by improving its reliability. This list of similar functions performed by these tools shows that the main advantage of both tools is that they both assist in the discipline of systems development.

These similarities alone suggest that it could be advantageous to provide one tool to fulfill both tasks. There are however other reason for doing this. The data dictionary provides additional functions. Among these are a entry validation process, through which new entries can be evaluated to determine whether they are new or redundant, a data manipulation language, to provide insert, modify, update, and search facilities, and report generators to provide reports on the information in the dictionary. All of these tasks are useful to the system development process. They provide tools that are of use to or needed, not just in the LRG, but also throughout the system development process.

In the discussion of the planning phase of software engineering, in the previous chapter, the use of an enterprise analysis methodology, called BICS, to enhance the development process was explored. This same topic has been examined in DBMS, and computer assistance to a methodology for enterprise analysis called business systems planning, (BSP) has been implemented, through the use of IBM's DB/DC Data Dictionary [22]. The data dictionary offers an integrated approach to the management of activities from enterprise analysis on through requirements definition, database administration, systems design, development, and maintenance. BSP shows a way in which a data dictionary could be used to enhance the software development process, starting with the first phase. What is needed is further research into the use of the data dictionary in software development.

Chapter 4
A Tool for Managing Databases and Supporting Software Development Systems

4.1 Introduction

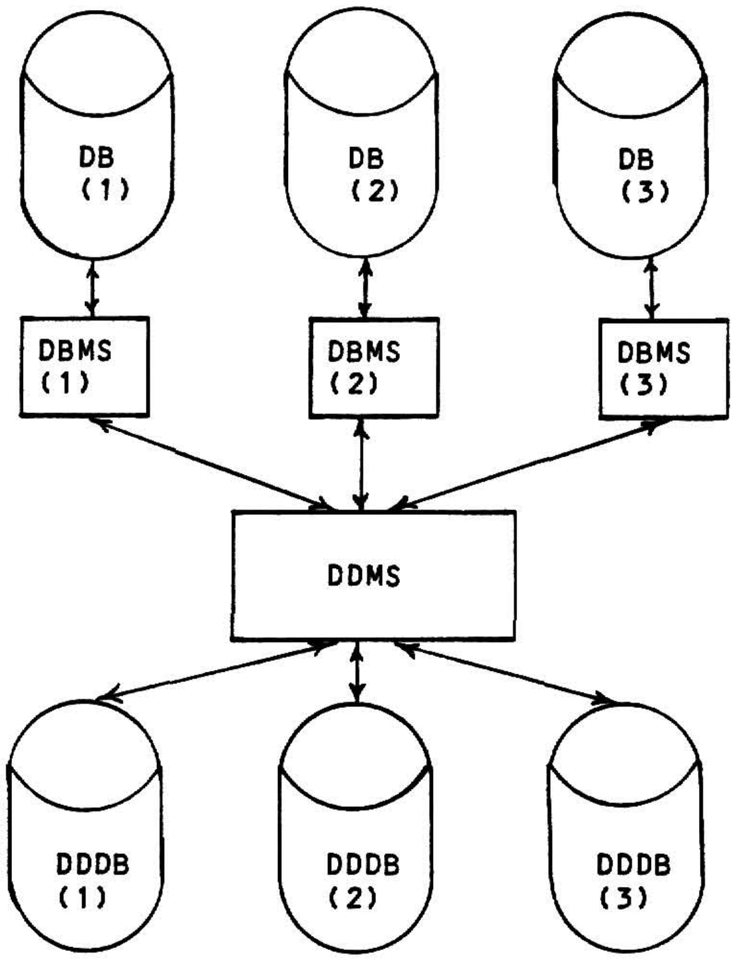
Having taken a look at some of the developments and theories from AI, DBMS and SE, examined the similarities between the disciplines, and seen how they might be used to compliment one another, it is possible to turn attention to the primary purpose of this paper, the proposal of a tool for the support of database management and software development. It has been said that there is a need for a system that incorporates all the good aspects of current data dictionaries, while providing some additions and modifications that would create a paradigmatic form of a data dictionary [20]. R. Phillips has attempted to do just that with a proposal for a data dictionary that he calls the dynamic data dictionary [20]. It is the intent of this paper to propose a modified version of this dictionary that will fulfill the goals set forth by Phillips, and at the same time show how this new dictionary, called the System Resource Dictionary, (SRD) could be used to provide support for the software development process, through a system that incorporates the SRD, which shall be referred to as the Resource Management System (RMS).

4.2 The Dynamic Data Dictionary

The Dynamic Data Dictionary System, (DDDS) as proposed by Philips, [20] should have the following characteristics: it must be flexible, able to support a variety of DBMS that support different data models; it must be a primary free-standing software system. The basic structure of this system is based upon one suggested by F.S. Zahran [28].

Zahran's proposal separates the software that manages the data dictionary and is used by the DD users from the Data Dictionary Database (DDDB). This software he calls the Data Dictionary Management System (DDMS). It is intended that this separation lead to an architecture that allows one DDMS to drive and control several DDDBs (see figure 7). The objectives of this structure are: to reach independent of the associated database's structure a clearer specification of the facilities and functions of the dictionary management software. The DDMS has a role in three main areas, they are: the support of different data management facilities allowing for the management, control and updating of the different data dictionary databases it is associated with; the support of interactions with the various dictionary users; and the support of interfaces between the DDMS and various standard software packages to provide facilities that the dictionary users may need.

Within the DDDS the DDMS should have functions that provide: data management facilities, similar to the facilities provided by an ordinary DBMS, for managing the data stored in the DDDBs but more sophisticated and versatile; dictionary language support to provide interactive or batch facilities for the definition of



DDMS - Data dictionary Management System
DBMS - Database Management System
(may be mixed DBMSs)
DB - Database
(may be hierarchical, network or relational)
DDDB - Data Dictionary Database

Figure 7: Zahran's Data Dictionary System.

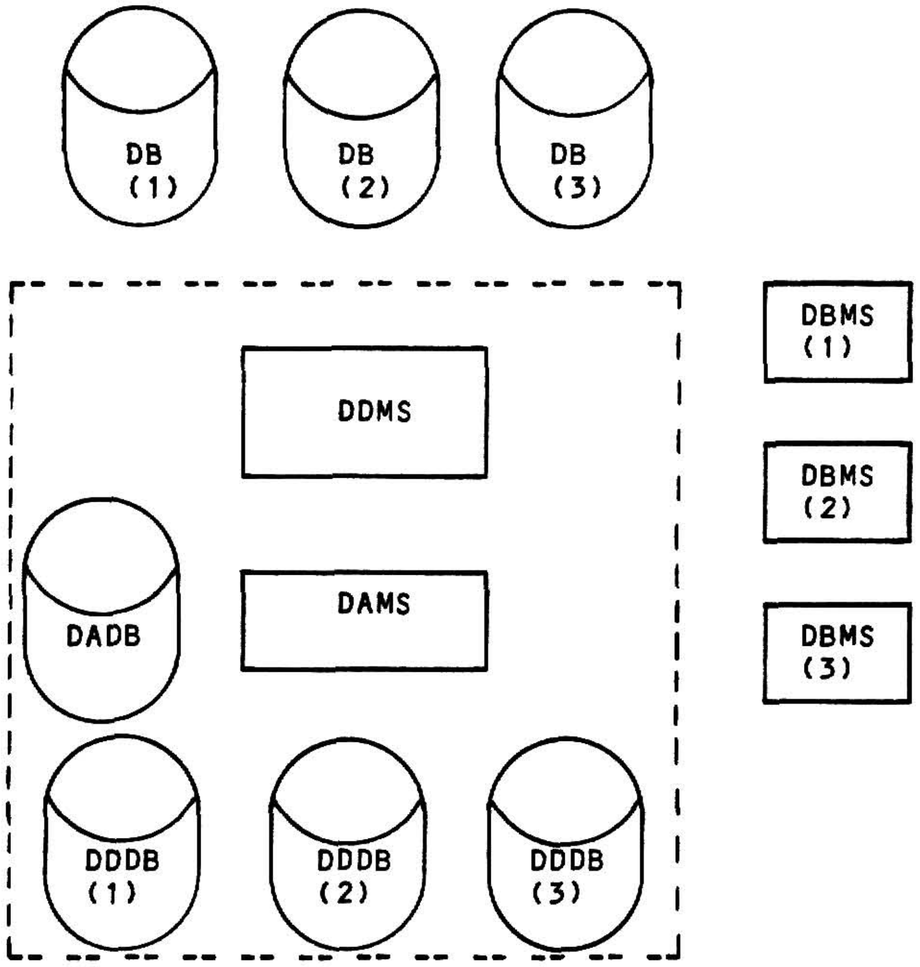
dictionary entities, the interrogation of dictionary contents, and the generation of data and procedure definitions; and an interface to other software systems such as report generators, query language processors... etc., for use on the dictionary data, and DBMSs to provide access control, database descriptions,...etc., to these DBMSs when needed.

At this point Philips suggests some modifications and additions to Zahran's model to arrive at his DDDS. Philips' changes include: multiple relational DDDBs that contain elementary and complex data objects; a new Data Abstraction Management System (DAMS) and associated Data Abstraction Database (DADB) which will be placed between the DDMS and its DDDBs; and DBMS access to a particular database through the DDDS only (see figure 8).

Philips describes a data object as an abstraction of some real world entity which should theoretically be general enough to represent any entity including programs, and hardware and software systems. He goes on to define a data object, using a definition provided by Unger [27], as a five tuple with the following components:

* name or set of names
* attribute or set of attributes describing its characteristics (e.g., real function)
* representation (e.g., string, C source code)
* corporality (indication of the number of copies, location, integrity, and security constraints)
* value

This data object is used by the data dictionary to represent data



DDMS - Data dictionary Management System
DAMS - Data Abstraction Management System
DADB - Data abstraction Database
DDDB - Data Dictionary Database
DBMS - Database Management System
(may be mixed DBMSs)
DB - Database
(may be hierarchical, network or relational)

Figure 8: Philip's Dynamic Data Dictionary System

entities in the database. It provides security, integrity, location and replication information. There are two types of data objects, real, and virtual or complex. Real data objects are present in atomic or structural atomic form, while for virtual data objects, the representation and value are provided by a process that is executed in order to compute its value.

The responsibility for access security controls falls on the DDMS in cases of data retrieval involving the satisfying ot a request for an element from a DB (see 20 for a description of this process). Changes made to the DDDBs, such as insertion, deletion, or modification, are monitored by the DAMS, to allow for accounting for and adjusting for these changes within the DADB, when they involve a virtual element. The DADB contains the names and aliases of all virtual data elements represented in the databases and their conversion routines.

Phillips introduced a module layered between the DDDS and its DDDBs to handle virtual data elements within the DBs, and has called it the Data Abstraction Management System (DAMS). This DAMS supports its own database, as mentioned above, called the data abstraction database. The DAMS would be responsible for processing all queries from the DDMS concerning the contents of the DDDBs, and passing to the DDMS the information necessary to convert virtual objects.

This configuration eliminates direct access or query of the DDDBs by the DDMS, and gives the DAMS the responsibility of monitoring the deletion, insertion, retrieval or perusal of data objects within the DDDBs.

This insertion of the DDDS between the DBMSs and their databases should, according to Phillips [20], help realize the following goals of database management: data independence, sharability, nonredundancy of data, reliability, integrity, access flexibility, privacy and security control, performance and efficiency, and administration and control.

4.3 The Systems Resource Dictionary

The System Resource Dictionary can be defined as an ordered collection of data about a database or information management system, containing information about the systems data, as well as information about the system and its utilities/programs. This dictionary is part of a software system that henceforth shall be termed the Resource Management System or RMS (see figure 9). The RMS should provide the following facilities:

* support for a software development process
* data management facilities
* dictionary language support

  to define/describe the program modules/activities

  to describe the relationship between the data and

  the programs that use and generate the data

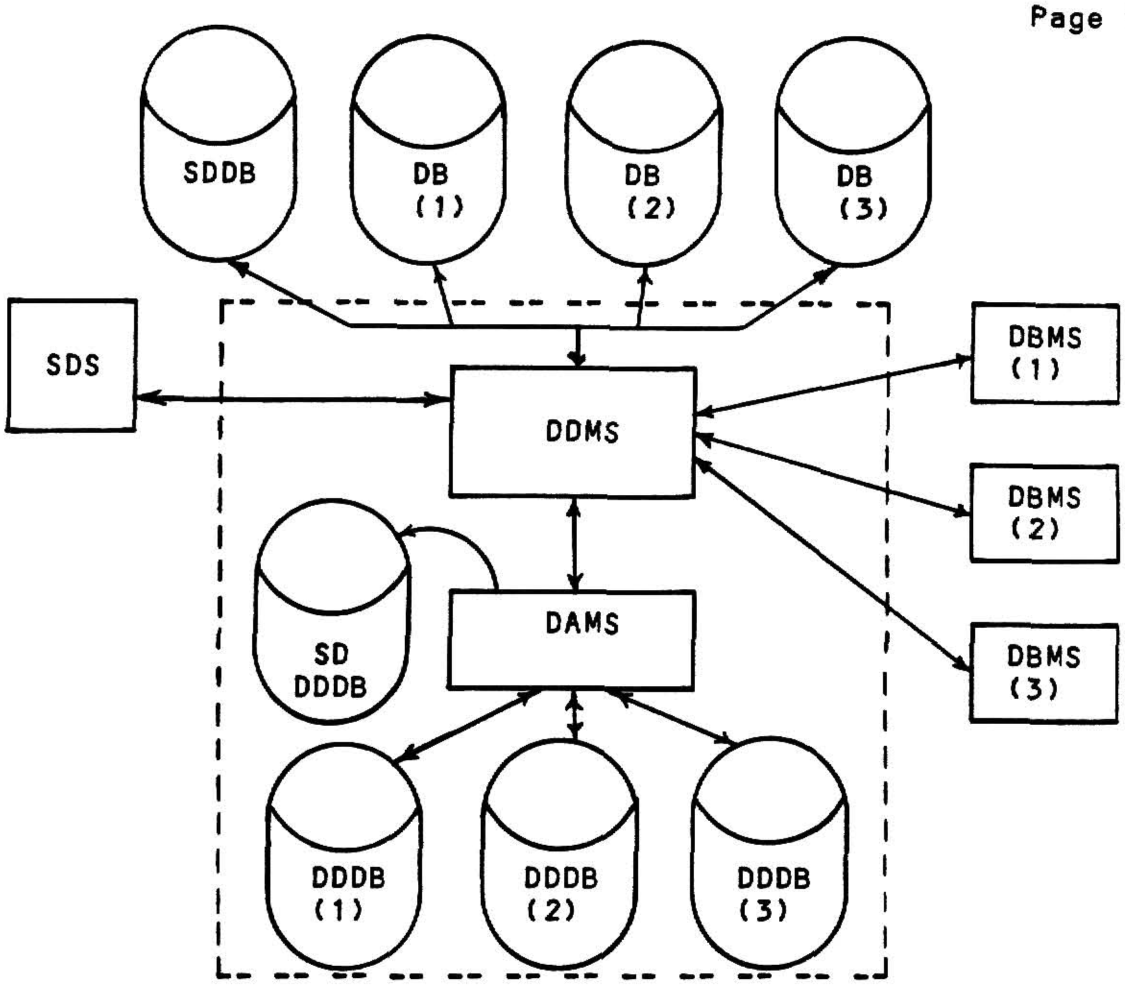
  to identifies real vs. virtual data

  Real Data: data explicitly contained in the database

  Virtual Data: data derived from the real data

  to allow ad hoc queries of the DDDB
* interfaces to other systems

The purpose of the RMS is the integration of software engineering with database/information system management for total



Systems Resource Dictionary:
  - DDMS - Data dictionary Management System
  - DAMS - Data Abstraction Management System
  - DDDB - Data Dictionary Database
  - SDDDDB - Software Development Data Dictionary Database

DBMS - Database Management System
(may be mixed DBMSs)

DB - Database
(may be hierarchical, network or relational)

SDS - Software Development System

SDDB - Software Development Database

Figure 9: Resource Management System.

system development, in order to attain the goals of nonredundancy in program development and data acquisition/retention; data and program independence; sharability; reliability; access flexibility; data integrity; performance and efficiency; and administrative privacy and security control. In addition we have the purpose of providing inferencing capabilities to database management systems and support to multiple databases and DBMSs. With the above purposes in mind some of the requirements that the SRD must fulfill include:

* ease of use
* provide system and data security
* provide an entity comparison function
* structured requirements definitions as inputs
* provide support for a structured software development process from planning through maintenance
* provide DML and query languages
* provide report generators
* structured design and construction (the system itself must be modular for ease of modification)
* allow multiple views of the dictionary data at varying levels of generalization
* provide interfaces for many different DBMSs
* provide support for different data models

Looking over this list it can be seen that Philips' dictionary system would meet most of these goals and requirements as provided.

The basic structure of Philips' dynamic data dictionary is therefore retained in what shall be termed the System Resource

Dictionary. To achieve the integration of software development with information management, make the dictionary easier to use, and provide inferencing capabilities to the system, a few modifications and additions will be introduced:

* To eliminate a duplication of facilities the requirements definitions language of the software development process will be integrated with or replace the data definition language of the data dictionary.

* The data dictionary databases will be implemented using frames and scripts, from artificial intelligence, to provide inferencing capabilities and provide attached procedures for the computation of virtual element values.

4.3.1 Using Frames in the SRD

Frames, and an associated concept scripts, are an artificial intelligence concept. There are a number of ways in which the use of frames could be useful in the Resource Management System (RMS).

To begin with, frames can provide the necessary means for implementing Unger's [27] five tuple data object discussed earlier. Frames provide slots which can be used to describe the different components of the data objects. Slots for the name and its synonyms, the attributes, and representation are possible. An example of how the basic data object could be implemented, as a frame, is shown in figure 10. The corporality component, which provides administrative information, could be implemented as a special slot that is actually a subframe to the data object frame. A way in which this could be done is shown in figure 11.

```
DATA_OBJECT Frame:
     SYNONYM(S): (Entity, Element...)
     ATTRIBUTES:
          (*Integer, Real, Character, Boolean, Hexadecimal,
          Binary, Record, String, Array, Set, Frame, Script,
          Procedure, Function, Activity, User, Terminal,
          Specialization_of: range: a Data Object Frame,
          Restriction_on: range: a Partial Data Object Frame,
          Default: range: a specified Attribute Slot value,
          If_Needed: range: a specified script or procedure*)
     REPRESENTATION: a data_object's system image
          (Frame, Script, Record....) Default: Frame
     CORPORALITY: a Record of an Objects Corporality
          Default: D_P_CORPORALITY
     VALUE: (atomic_value_reference, structure_of_atomic-
     _value_references, structure_of_objects)
```

Figure 10: A frame representation of the data dictionary data object

```
RECORD_OF_CORPORALITY Frame
     Synonym(s): (nil)
     Attributes:
          Specialization_of: DATA_OBJECT
          #_of_Copies: an integer Default: 1
          Integrity:
          Location(s): System Address(s) (*data dictionary*)
          Status: range: (In_Use, Proposed, archived)
               Default: Proposed
          Security_Constraints:
               Owned_by: range: (ADM, ACCTING, MKTING, MFGING,
               PRSNL, D_PING)  Default: D_PING
               Used_by: range: (ADM, ACCTING, MKTING, MFGING,
               PRSNL, D_PING)  Default: D_PING
               Modified_by: a D_P_EMPLOYEE.NAME
               Modified_on: a DATE
     Representation: a frame
     Corporality: D_P_CORPORALITY
     Value:
```

Figure 11: A frame representation for data object corporality

The fifth component, the object value, could also be implemented as a subframe, providing for the attachment of a procedure or script to calculate or derive the object's value.

This implementation could eliminate the need for Philips' data abstraction database, (DADB) but it would not, of course, eliminate the need for most of the functions provided by his data abstraction management system. When a virtual data object was referenced, rather than having to access the DADB for virtual data object information, the system would simply access the object as it is stored in the system resource dictionary database. The information necessary for its use would be acquired from the value slot, which would tell the system whether the data object was real or virtual, and if it is virtual how to derive its value from the database(s).

Frames can be allowed to inherit the attributes of their parents, restrict the value of the inherited attributes, and show relationships between frames. The inheritance property of frames opens the way for the definition of different levels of data objects having common characteristics, but not being necessarily fully equivalent. Being able to show relationships between frames allows the inclusion of more information about object relationships than do traditional database models.

The inheritance property of frames allows the definition of objects to start at a generalized level of definition and proceed to more restricted or detailed definitions. In other words you attempt to have all "like" elements have a homogeneous structure or syntax. For example, the data object date could be defined on a general level as a triplet consisting of the month (mm), day (dd), and year (yy), mm-dd-yy, with mm defined as a set of months where months is an integer between 1 and 12, dd is defined as an integer between the numbers 1 and 31, and yy is defined as an integer (the remainder of the information for the five tuple is unimportant for this example). All child occurrences of date by any name in the dictionary would use this basic generic definition and could be said to be derived from the same set of dates for a given year or time period. They could however restrict its values or attributes. The months could be changed to be a set of 12 character strings, (January, February, ..., December) for example.

The specific meaning of a given child date would be contained in the relationship the specific date subtype was involved in. The difference in meaning in the use of a specific calendar date, say August 5, 1985, can only be seen in the context within which it is used by a relationship. This can be shown by using two records or data objects, called assignment and loan. Both records have three attributes, employee, department, and date. We can say that the underlying structure and general meaning behind the two uses of date shown are equivalent. It is only within the context of the relationships given that the two uses of date have a different meaning, which are assignment.date and loan.date, or the date of an assignment and the date of a loan. The relationship the general attribute date is a part of gives that instance of date its unique meaning. Of course, if you define a restricted data object with a different name using date as its parent frame this problem is avoided. With frames and their ability to show relationships and inherit attributes this may not be necessary.

The use of frames, together with other AI techniques, could make it possible for the RMS to become an expert system on the different data models used in database development. The RMS would have all the facilities of a DBMS. The use of these techniques also raise the possibility of eliminating the duplicate facilities existing across the various DBMSs incorporated into the RMS, leaving only their user interfaces. This would allow the various users to continue interacting with the system as they did before the integration of their DBMS with the RMS.

A great many possibilities could be opened up using frames to implement the data dictionary's database. By so doing, the database becomes more of an AI knowledge base containing a great deal of information about an organization and the system of which it is a part. Their use could greatly increase the efficiency and inferencing capabilities of the DMLs and query languages ot the system, since frames can have inferencing capabilities built in.

In the next section, a methodology for the use of the software development process in the system resource dictionary will be suggested and the above concepts discussed.

4.3.2 A Methodology for Software Development Support

This is not a proposal for a particular method of software development. What is being proposed is a method for supporting and enhancing any software development methodology, such as mentioned in the section on software engineering in Chapter 2, using facilities that the proposed SRD would supply. The specific manner in which, and degree to which this integration of software development and data management, using the SRD, would occur would of course depend on the software development methodology chosen and the manner in which the SRD is implemented.

In Chapter 2, two methods of using enterprise analysis in systems development were discussed. The first of these was Business Systems Planning (BSP) [22], which used the IBM DB/DC Data Dictionary to provide information management support for the planning process. The second was the Business Information Control Study or BICS [38], which uses an extension called REQGEN (requirements generator) to generate the requirements specification for an organization's data processing needs.

These two methodologies show that it is feasible to push the requirements analysis or planning phase of systems development back into the area of business management analysis. By so doing, the systems developer can be provided with a great deal of useful information on the organization's data processing requirements, at an earlier time in the business planning process than was possible before. In addition, the whole business planning effort benefits from the increased information processing support that can be provided by the SRD.

The SRD as proposed could provide these same facilities, and in the future provide even greater support to the planning phase. This would be accomplished through the built in inferencing capabilities that could be provided by the use of frames and scripts in implementing the dictionaries database(s).

In the requirements definition phase of software development, the SRD would provide a separate dictionary database for each new project's requirement definitions. The language facilities for the specifications of the systems requirements could, as discussed in the last chapter, be provided by the SRD through a data or requirements definition language. Properly developed either one of these tools can provide the necessary information for both the software development process and the data management process, as implemented using the proposed Systems Resource Dictionary.

The data object, as defined by Unger [27] and used by Phillips, [20] shows the feasibility of this approach. The components included in Unger's data object represent the kinds of information necessary for the processes carried out by both systems development and systems management, which should be no surprise as both tasks are part of the software life cycle.

The use of the SRD in system design represents an opportunity to combine the functions of data and program management. In the last chapter, the similarities between managing the program library through a library resource guide and managing the systems data through the data dictionary were discussed. The conclusion was reached that the functions of the two were so similar that it would be feasible, and even advantageous to combine these two tasks. This is one of the points of overlap between the DBMS and Software Engineering disciplines of computer science.

By having access to one facility such as the SRD that contains all a systems resource definitions (data, program, functions,... etc.) the design phase of software development can be greatly enhanced. This enhancement takes the form of a means for reducing programming and data redundancy. The dictionary is used to help the designers of a system reduce redundancy, by providing facilities for the comparison of existing data and process (program, function, procedure) definitions with the definitions resulting from the requirements definition phase of the systems development process. Such a process would be carried out something like this: requirements definitions would be compared with existing activities and data items/entities in the SRD to determine their uniqueness or redundancy (look for existence of equivalence between data items and between activities).

(a) Data item - If it exists in the system you need only add the data item's name if it is different from the systems name for that data item and is not a listed synonym. If it can be derived from existing data as a virtual data item it is added to the dictionary, along with the procedure for deriving it from the existing data. If the data item doesn't exist, and can not be derived then it is added to the system.

(b) Activity - If a program module already exists to perform the needed task then it should be used. If a suitable module does not exist then it should be ascertained if there exists a module that can be modified to perform the needed function. If such a module exists then it should be used only if the needed modifications are not extensive. If no equivalent or easily modified module exists then and only then do you design, code and test a new module. The new or modified module should be added to the SRD, and system usage for each module used should be updated to reflect its usage in the new application.

This process requires a facility that provides entity comparison capabilities. It would have to be capable of comparisons on multiple keys, such as an entity's name and its synonyms, or process name and function(s).

One way of comparing data objects would be to start by comparing data object names and synonyms. If a match is found in any of the name or synonym comparisons then their representations would be checked to determine if they matched. The next step would be to check their attributes to determine if they are a complete or partial match, i.e., whether or not the names, types, and functions of the attributes present in the data objects match. If an attribute has a complex structure it would go through the comparison process in the same manner as the data object of which it is a member. It is not necessary that the number of attributes match, although it would make things easier if they did, since even a partial match of a new data object with one present in the data dictionary can provide a useful base on which to build the new data object. This would hold true whether the new data object was less or more complex than the data object that has been located as a possible match and would be true for both data items and activities (processes), provided structured programming techniques are used in the construction of all activities present in the SRD.

The SRD could allow for the creation of separate databases for the definition and testing of each individual development project, and provide the facilities for the integration of the definitions in the dictionary database of the development systems and the SRD's operational database(s), as well as provide the facilities for comparing dictionary entries with the requirements definitions.

In system construction and testing, the next phase, the SRD will of course aid in reducing the amount of code to be written through the use of the facilities described above. In addition, it would provide the facilities to allow the different programmers to use program modules developed, by others, for a given project, by providing one place (the project database) for the storage of the new implementation. Through the maintenance of the security and integrity of the data and process definitions the facilities of the SRD would be able to prevent code and data definition changes by unauthorized personnel, and help to maintain the program module interfaces, by preventing the proliferation of different versions of a program module, and identifying all entries affected by any change to an entry.

Testing of the new application could be aided by the ability of the SRD to store and produce reports (using report generators) on the tests carried out to prove the program modules as they are completed and integrated. Using frames, each data object in the dictionary would be able to contain information on its usage. By developing a facility to step through applications code and test each branch and note each use of a variable and place that information in the corresponding SRD data object it is possible that program module correctness and data usage could be tested.

Once a new application was installed in the system, this same facility could be used to provide data management information on the usage of that application, and all other systems resources over their lifetimes. The SRD placed, as Phillips DDDS would be, in between the DBMSs and their databases, and also in the middle of system development process makes it an ideal candidate for monitoring the development, operation, and maintenance of all systems functions.

There are many reason for the integration of the resource management system and a software development system. First, the integration of different DBMSs into the RMS would be facilitated by their integration.

This integration could help in identifying duplicate facilities, and aid in the production of the interfaces necessary to incorporate the DBMSs various facilities into the RMS. Integration would be accomplished by developing, if they are not already developed, a set of requirement definitions for the new DBMS's resources, and putting them through the comparison process used in the design phase of systems development described earlier. If a set of requirement definitions already exists for the DBMS then they would of course be converted to be compatible with the SRD's definition language and comparison facilities and used in the integration process.

Further reason for this integration are enumerated here as a means of showing the benefits that could be derived from this combined system:

* Help in assuring quality of the system

* Successful integration requires modern software design methods, such as structured programming and structured requirements definition

* Standardizes module interface descriptions - aids not only testing but also reuse of program modules for other applications

* Cuts application development time

* Aids in program maintenance

* Aids in identifying data and programming redundancy

* Identifies new data elements and new program modules

* System security aid for both data and programs

* Helps provide information management for the software engineering process

* Provide inferencing capabilities to the system

* Advocates the use of a data dictionary data base system as the logical method of DB support for software development. (this allows a reduction of or avoidance of redundancy in system software for software development support)

* Aids in software and DBMS documentation

The system resource dictionary becomes an integrated part of the software development function, leading to the natural documentation of the system as it is developed and as it evolves. In addition, the RMS could help in familiarizing new personnel with system resources, and of course, provide all the facilities and functions of Philips DDDS and any other data dictionary.

4.3.3 An Example of the Use of Frames in Software Development

In order to show an example of the use of frames in software development the design phase of a structured development process, developed by Warnier and Orr, [42] is used to demonstrate the conversion of Warnier/Orr diagrams into frames for a Systems Resource Dictionary database. The example used here is a portion of an example of a sales report generator used in a text, by David Higgins [37], which discusses the use of Warnier/Orr diagrams in designing structured programs. The diagrams and frames for this discussion are contained in the appendix at the end of this report and will be referred to throughout this discussion.

For this example two assumptions are made:

(1) that the invoices are ordered on invoice number within customer number within salesman number.

(2) that the relationship between customer and salesman is many-to-one, i.e., many customer to one salesman, salesman.sales is the total sales per salesman, and in each case customer.sales is the total sales per customer

The Warnier/Orr diagrams can be divided into two basic types of diagrams for the purposes of this example. The two types consist of data structure diagrams (figures 12-15) and process structure diagrams (figure 16). This is a some what loose translation, but it will be sufficient for the purposes of this example. Each bracket, of a Warnier/Orr diagram, represents a successively lower level of hierarchy moving left to right in a diagram (refer to figures 12-16). Each bracket or level of hierarchy is associated with a word or phrase identifying it, which may be associated with the identifier for the next higher level. *A period, at the beginning of an identifier,* is used to show association with the identifier of the next higher level. Each level can be thought of as an element, and the identifier for that level as the element's name. For example, figure 13, which shows the minimal data (input) needs of the program, has levels named Company, Salesman, Customer and Invoice. It also shows elements identified as .Name on the second and third levels. These are examples of element identifiers that would be concatenated with the identifier of the next higher level, and would be referenced as Salesman Name and Customer Name.

In the case of data diagrams, each element listed on a given hierarchy level is either an atomic or *simple attribute*, or a complex attribute which has a lower level of hierarchy associated with it. Figure 14, in the appendix, shows examples of both simple attributes (Salesman Name) and complex attributes (Invoice).

The diagrams representing processes are somewhat more complex. Each level of hierarchy represents a lower level process. *However, only the individual* elements on a given level associated with a lower level of hierarchy are process identifiers and attributes of the next higher level of hierarchy. The remaining elements of a given level are process statements for that level. In figure 16, of the appendix, Company Begin, Process Salesman, and Process Customer are examples of processes. Other elements, such as Set Company Sales Total to Zero, are examples of process statements.

Process elements are either sequential, repeating, or alternating. Repeating elements are represented through the specification of the number of element repetitions placed in parentheses under the element's name. In figure 16, of the appendix, Process Salesman is one example of a repeating element showing its number of repetitions ([1,S]).

Those elements that are of the last two types, repeating and alternating, are associated with a physical test which will be used in the finished program to control branching. These tests are defined in footnotes on the process diagrams (see figure 16) and are noted in the diagram by a question mark (?) followed by the reference number of the test. For example, in figure 16, ?1 following the Process Salesman repetition specification refers to test ?1 - No More Salesman = True.

In order to convert the Warnier/Orr diagrams into frames, a number of decisions had to be made. These decisions were somewhat arbitrary, being based solely on the desire to stress clarity, ease of understanding, and readability in the frame implementation. Most of these decisions were general in nature, that is, necessary to the process of determining the frame constructs for the representation of the output of any software development system. The remaining decisions were specific to the conversion of Warnier/Orr diagrams.

The frames are based, as described earlier in this chapter, on a five tuple data object (figure 10). This data object frame provides the basic structure around which all the frames of this example are built.

Two slot types, Specialization_of and Restriction_on, are used to show the inheritance by one frame of another frame's attributes. The slot Specialization_of is used to show the inheritance by a frame of all the attributes of its parent frame. The slot Restriction_on shows the inheritance by a frame of selected frame attributes of another frame. Both slots, used in this manner, can show the relationships that exist between the data objects in a data dictionary. Figure 17, pages 4 and 5, shows examples of these two constructs. By referring to figure 18 the relationship between the Salesman, Customer, and Invoice frames and the Employee frame can be seen. Also shown, by these two figures, is the relationship that exists between the "generic" date frame and the specific uses of that frame as a definition for the date attribute in various other frames.

Two additional concepts, discussed in the AI section of Chapter 2 dealing with frames, are default and range. Default and range can be thought of as attribute subslots, which can be used to show an attribute's default value, or a set of values that an attribute may be expected to have.

In this example, it was decided to have each frame represent the conversion of one level of a Warnier/Orr diagram. This shows the strong relationship that exists, in this instance, between the frame implementation and the Warnier/Orr diagrams. Each frame, in the example, takes its name from the identifier for the hierarchy level it represents. This is illustrated by a comparison of the diagrams and frames presented in the appendix. The division of the diagrams by hierarchy level leads to a naturally structured or modular representation of the diagrams, and encourages the structured development of applications.

There are two basic types of frames, data or input/output frames, and process frames. The two types of frames correspond to the two diagram types. As stated above, each level of a data or process structure diagram is implemented as a frame.

In keeping with the decision to have each frame represent one level of a diagram, each process frame is defined so as to indicate only those inputs, outputs, and processes used directly by the process being defined. This can be seen, in this example, by comparing each element of a diagram representing an input, output, or process with the corresponding frame's attributes. In those cases where the element is associated with a lower level of hierarchy it is also a frame identifier (name), and the elements of the lower level are its attributes. Those elements, in process structure diagrams, representing process statements are part of a frame's value slot which shows the processes implementation.

By looking at the Process Company frame (figure 17, page 1 of the appendix), and comparing it with the process structure (figure 16), the division of the elements of the first level of the process structure between statement elements and process elements can be seen. It can also be seen that no inputs or outputs are shown in this frame. It is logical to suppose, that since the example is that of a sales report generator, that some kind of output requiring a number of inputs would be required. This results from the decision to have each frame represent only one level of a diagram and nothing more.

For the purposes of this example, the Data Object Corporality is simply named. It would however follow the pattern of the example shown in figure 11 of this report.

In any attempt to implement some form of frame representation for a data dictionary database it will be necessary to answer a number of questions. Most of the answers to these questions turn out to be arbitrary ones, based on the preferences of the individuals involved in the projects implementation more than anything else. Never-the-less, it is important that these questions be answered so that a clear picture of the frame implementation may be attained. The following is a list of the questions that were asked and answered in some form for this example:

> What is the best way to show the relationship between an inherited frame attribute and the frame from which the attribute is inherited?
>
> Would it be best in cases where only 1, 2, or 3 elements are inherited to just redefine those elements in the new frame and use other means to show the relationship between two frames?
>
> Could these relationships be shown through attribute naming conventions where related frame attributes are given abbreviated or extended names based on the names of the frame attributes to which they are related?
>
> Would it be best to rename all attributes used in a frame that are inherited from other frames except where a whole frame is inherited intact, in which case only the inherited frame's name need be changed? Is it even

necessary to change the inherited frame's name? (The problem with this is that it clutters up the frame representation and reduces readability. Then again it can serve to make the relation between the two frames involved much clearer.)

How much detail should be included in frames that define processes? Should a "program" frame show all inputs and outputs of a program even if they are not used by the main program module, or should inputs and outputs be defined only in the process module frames where they are first used?

The frame representation used here is merely a suggestion of how frames could be implemented. Such frame slots as Specialization_of, and Restriction_on may or may not prove useful. The same is true for the manner in which a default slot value could be represented. The whole question, of the feasibility of using frames for representing data dictionary data objects, clearly needs a great deal more study which is beyond the scope of this paper. It remains to be proven whether or not this suggestion will provide an increase in the productivity and usability of large data dictionary systems.

Chapter 5
Conclusion

Recent developments in computer science have made it clear that there is a need for a synthesis of tools technology and methodologies in the various areas of computer science. This paper has presented a brief look at some of the various tools and methodologies of three areas of computer science, AI, DBMS, and SE, and attempted to show how they may be used together, in various ways, to enhance and avoid duplications of effort in the use of similar or complimentary tools and methodologies from the various areas.

The literature search done, in the areas of AI, DBMS, and SE, shows that there are many ways in which developments in each of these three areas can be used to enhance the capabilities of systems and methodologies developed or being developed in the other two. Some of the developments that have been enumerated within this paper include the use of natural language interfaces from artificial intelligence to enhance user interfaces in DBMS and software engineering, and the integration of software development in an enhanced data dictionary. It has also been illuminated that many of the phases in the software development process could be implemented as expert systems as they fall within a set of generic expert tasks outlined in Chapter 2.

The Theoretical System Resource Dictionary and Resource Management System, proposed in this work as an enhancement and modification to work done by Phillips [20], appear to provide the facilities and future opportunities necessary to a viable solution to the problems associated with integrating DBMSs and other systems, such as a software development process. It continues to provide all the benefits, and meet the goals set forth by Phillips and others for managing databases such as data security, independence, and nonredundancy. The RMS would allow the absorption by an organization of any existing DBMS environment while avoiding excessive costs, mismanagement, and loss of its valuable information and system resources.

There is a clear need for future research into dictionary driven systems such as the RMS and the DDDS. A beginning in this research would be the implementation of a system such as suggested in this paper or by Phillips. It is vital in this effort that the use of knowledge systems and knowledge bases be explored. Their use could provide great advances in information systems management by enhancing the inferencing capabilities of the RMS and associated DBMSs.

BIBLIOGRAPHY

ARTIFICIAL INTELLIGIENCE:

1) Barr, Avron and Edward A. Feigenbaum The Handbook of Artificial Intelligence Vol. 1, William Kaufman, Inc., 1981.

2) Barr, Avron and Edward A. Feigenbaum The Handbook of Artificial Intelligence Vol. 2, William Kaufman, Inc., 1981.

3) Brodie, Michael L. and Stephen N. Zilles, "What Should Be Modeled?", ACM SIGART-SIGMOD-SIGPLAN Workshop 1981 ACM p40-52.

4) Charniak, Eugene, "Organization and Inference in a Frame-Like System of Common Sense Knowledge.", TINLAP-1 MIT 1975, (ACM 1975), pp 46-55.

5) Deutsh Peter L., "Constraints: A Uniform Model For Data and Control.", ACM SIGART-SIGMOD-SIGPLAN Workshop 1981, Michael L. Brodie and Stephen N. Zilles ACM pp 118-20.

6) Goldstein, Ira P., and Bruce Roberts, "Using Frames in Scheduling.", AI: An MIT Perspective, D. H. Winston and R.H.Brown Editors, MIT Press, (1979) pp 253-84.

7) Hartley, R.T., "Knowledge Systems"; papers on Production Systems and Semantic Networks (lecture notes).

8) Hayes, Patrick J. and Gary G. Hendrix, "A Logical View of Types.", ACM SIGART-SIGMOD-SIGPLAN Workshop 1981, Michael L. Brodie and Stephen N. Zille, ACM p128-30.

9) Hayes-Roth, Frederick, "Knowledge-based Expert Systems.", Computer, October 1984, pp263-73.

10) Stefik, Mark, "An Examination of a Frame-Structured Representation System.", 7th IJCAI UBC Vancover 1981, IJCAI 1981 P845-52.

11) Stefik, Mark et al, "The Organization of Expert Systems,A Tutorial.", Artificial Intelligence, North-Holland 1982 p135-73.

DATA BASE MANAGEMENT :

12) Cardenas,A.F., Data Base Management Systems, Allyn and Bacon, Inc., 1979.

13) Durell, W., "Disorder to Disipline Via the Data Dictionary.", Journal of Systems Management (USA), vol. 34, no.5 p12-19 (May 1983).

14) Ewers, Jack E., "How to Evaluate a Data Dictionary.", Computer World, 1981.

15) Gallaire and Minker (Eds.), Logic and Databases, Plenum, New York, 1978.

16) Hammer, Michael, and D. McLeod, "The Semantic Data Model: A Modeling Mechanism for Database Applications.", ACM Transactions on Database Systems, ACM 1978, pp.26-36.

17) Hammer, Michael, and D. McLeod, "Database Description with SDM: A Semantic Database Model.", ACM Transactions on Database Systems, Vol. 6, no. 3, September 1981 p351-86.

18) Kent, W., "Consequences of Assuming a Universal Relation.", ACM Transactions on Database Systems, Vol 6, No.4 December 1981, pp. 539-556.

19) Mylopolous, B., and Wong, "A Language Facility for Designing Database-Intensive Applications.", ACM Transactions on Database Systems, Vol 5, No. 2 June 1980, pp 185-207.

20) Phillips, Robert W., Dynamic Data Dictionary, Masters Report, Kansas State University (1983).

21) Risch, Tore, "Production Program Generation in a Flexible Data Dictionary System.", IEEE, 1980, PP. 343-348.

22) Sakamato, J.G. and F.W. Ball, "Supporting Business Systems Planning Studies with the DB/DC Data Dictionary.", IBM Systems Journal, Vol. 21, No. 1 1982, pp. 54-80.

23) Snyders, Jan, "Data Dictionary: The Manager in DBMS.", Computer Decisions, Vol.13, October, 1981, pp. 36-46.

24) Snyders, J., "New Trends in DBMS.", Computer Decisions, February, 1982, pp. 100-133.

25) Sowa, J.F., "Conceptual Graphs for a Data Interface.", IBM Journal of Research and Development, Vol. 20 pp. 336-357.

26) Ullman, Jeffery D., Principles of Database Systems, Computer Science Press Inc., 1980.

27) Unger, E.A., and E.J. Schweppe, "A Concurrent Model: Fundementals.", 2nd International Conference on Parallel Computation, France, 1979.

28) Zahran, F.S., "A Basic Structure for Data Dictionary Systems.", ACM European Regional Conference (England), Proceeding... Systems Architecture, March 1981.

SOFTWARE ENGINEERING:

29) Bell, Thomas E., D.C. Bixler, and M.E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering.", <u>IEEE Transactions on Software Engineering</u>, Vol. SE-3, No. 1, January 1977, pp. 49-59.

30) Bloom, Naomi Lee, "Writing Less Code - An Approachable Ideal.", <u>National Computer Conference, Anaheim, Ca., USA, 16-19 May 1983</u> (Arlington, Va, USA: AFIPS Press 1983) p.3-9.

31) Boehm, B.W. "Software Engineering.", <u>IEEE Transactions on Computers</u>, December 1976 pp. 1226-41.

32) Branscomb, L.A., J.C. Thomas, "Ease of Use: A System Design Challenge.", <u>Information Processing 83, Proceedings of the IFIP 9th World Computer Congress</u>, Paris,France, (Ampsterdam, Netherlands: North-Holland 1983) pp. 431-38.

33) Curtis, G. A., "Fondation Software : A significantly improved approch to the development of large application systems." <u>National Computer Conference, Anaheim Ca., USA, 16-19 May 1983</u> (Arlington, Va., USA: AFIPS Press 1983) p11-19 also pgs 113-21 and p137-44. Joint Computer Conference AFIPS conference Proceedings.

34) Demetrovics, Janos, Elod Knuth, and Peter Rado, "Specification Meta Systems.", <u>Computer</u> vol. 28 no.5 p29-35 (May 1982).

35) Depree, Robert W., "Pattern Recognition in Software Engineering.", <u>Computer</u> (USA) vol. 29 no. 5 p247-50 IEEE (May1983).

36) Greenspan, Sol J., J. Mylopoulos and A. Borgida, "Capturing More World Knowledge In the Requirements Specification." 6th International Conference on Software Engineering, IEEE 1982 p225-34.

37) Higgins, David, Designing Structured Programs, Prentice-Hall, Inc., 1983.

38) Kerner, David V. and A. Malhotra, "Generating Requirements From Enterprise Analysis.", 1983 National Computer Conference, Anaheim, Ca., USA, 16-19 May 1983 (Arlington, Va., USA: AFIPS Press 1983 p255-60).

39) Kuo H. C., C. H. Li and J. Ramanathan, "A Form-Based Approch to Human Engineering Methodologies.", Sixth International Conference on Software Engineering, (IEEE 1982) p254-63.

40) Lauber, Rudolf J., "Development Support Systems.", Computer Vol.28, no.5, (IEEE, May 1982) p36-46.

41) Levene, A. A., G. P. Mullery, "An Investigation of Requirement Specification Languages: Theory and Practice.", Computer Vol.28 no.5, (IEEE, May 1982) p50-59.

42) Orr, Ken, Structured Requirements Definition, Ken Orr and Associates, Inc. 1981.

43) Ross, D.T., "Structured Analysis (SA): A Language for Communicating Ideas.", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 16-34.

44) Ross, D.T., J.B. Goodenough, and C.A. Irvine, "Software Engineering: Process Principles and Goals.", Computer, May 1975 pp. 17-27.

45) Smoliar, S.W., "Software Specifications, Databases and Knowledge Bases.", Information Processing 83. Proceedings of the IFIP 9th World Computer Congress, Paris, France (Amsterdam, Holland: North-Holland 1983) pp219-22.

46) Teichroew, D., and E.A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 41-48.

47) Thedens, Melinda, "Cataloging The Program Library", Datamation (USA) Vol.29, NO.5 p247-50 (May 1983).

48) Vanderiel, Kenneth, "Software Development Methodology and Practices.", GTE Network Systems Journal 3rd Quarter 1983 p76-82.

49) Warren, Sally, Bruce E. Martin and Charles Hoch, "Experience with A Module Package in Developing Productrion Quality PASCAL Programs.", Proceedings of the 6th International Conference on Software Engineering, Sept. 13-16 1982 IEEE Computer Society Press 1982 p246-53.

50) Wirth, Robert, "Future Direction of Software Technology.", GTE Network Systems Journal, Third Quarter 1983 p70-75.

51) Woodward, Mary, "A Case for Adaptable Aplications Software.", 1983 National Computer Conference, Anaheim, Ca., USA, 16-19 May, 1983, Arlington, Va., USA: AFIPS Press 1983 p21-28.

52) Zelkowitz, M.V., Perspectives on Software Engineering.", Computing Surveys, Vol 10, No 2, June 1978, pp197-216.

Appendix
An example of transforming Warnier/Orr Diagrams into frames for a System Resource Dictionary Database

The Warnier/Orr diagrams:

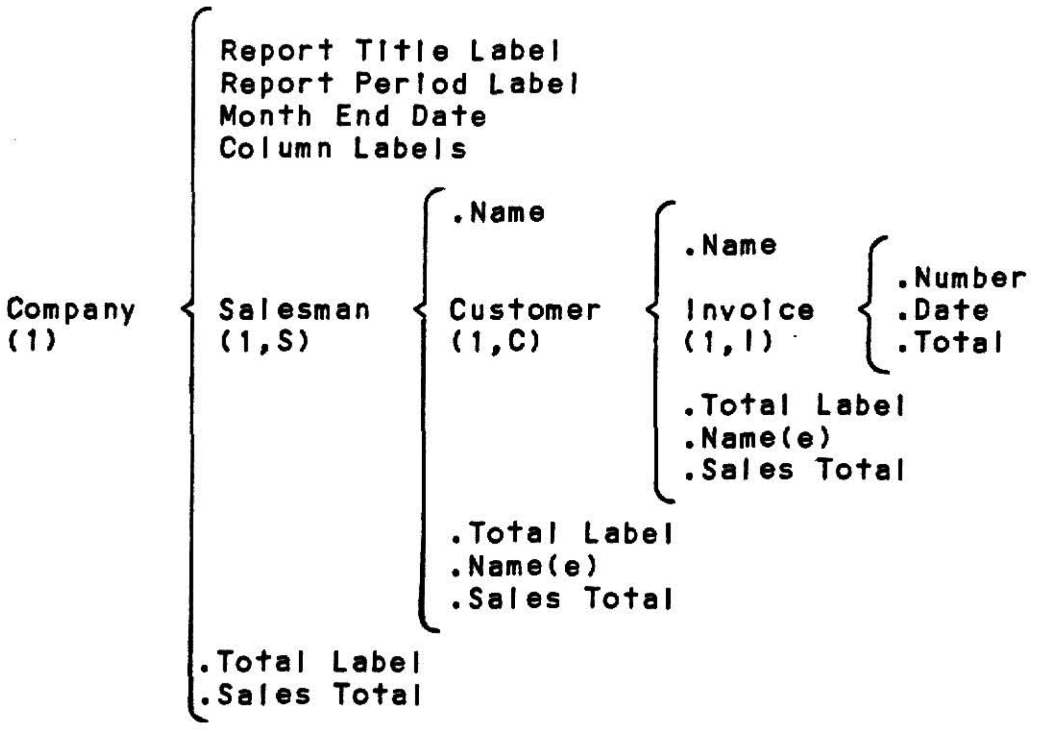


Figure 12: A logical output structure

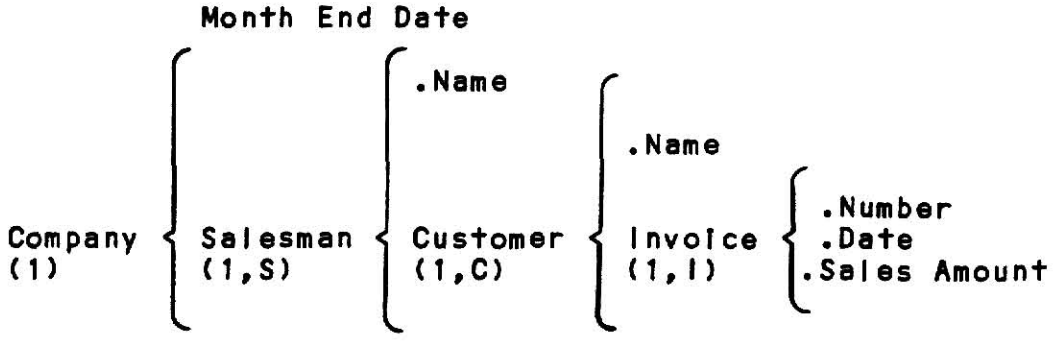


Figure 13: Logical data structure for the required output: represents the minimal data requirements.

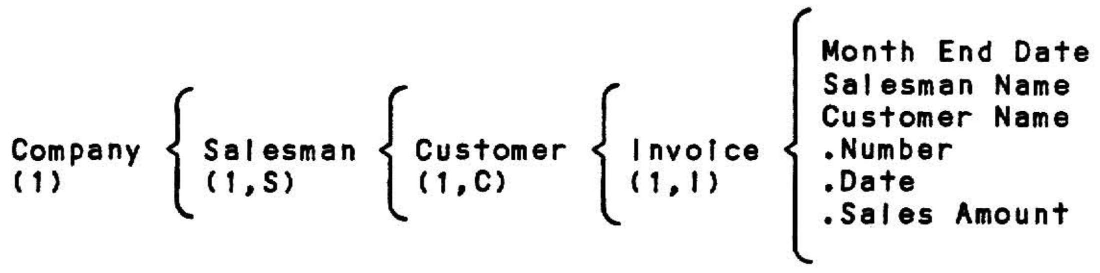


Figure 14: The ideal input file structure

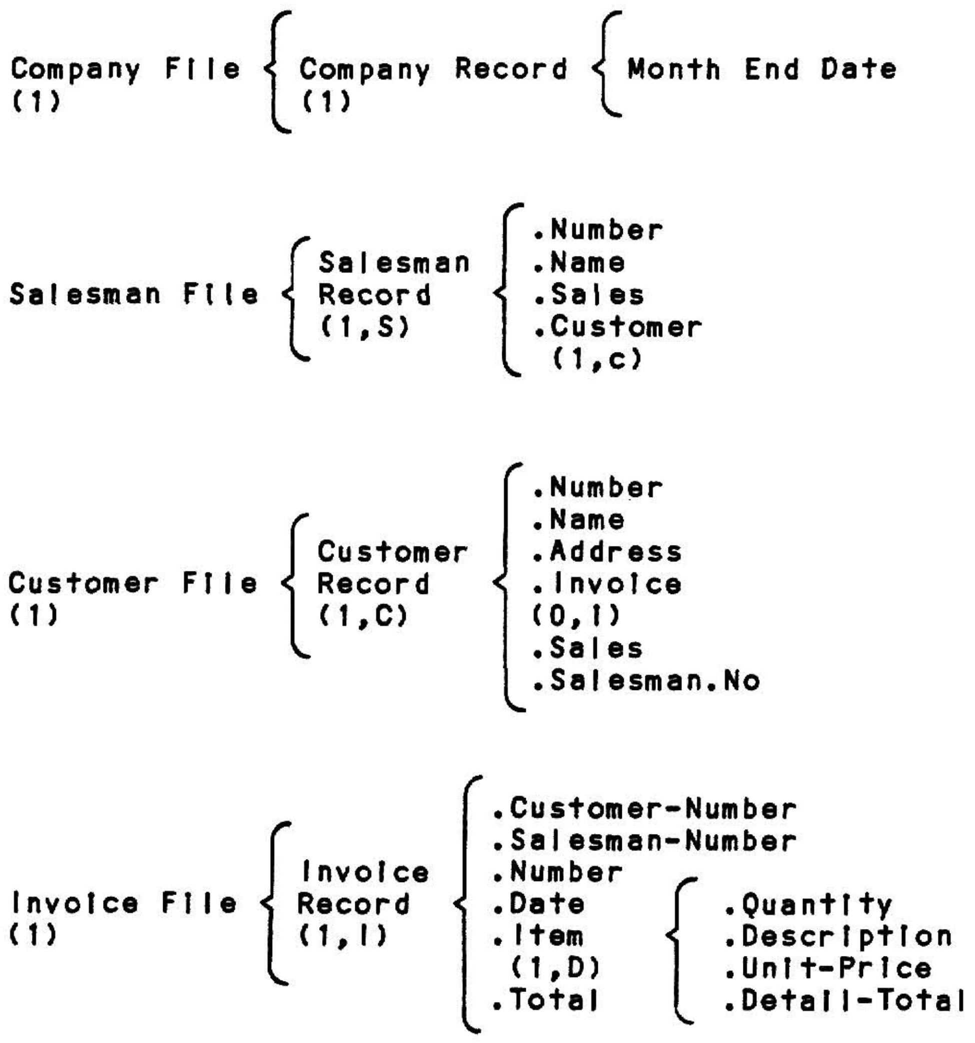


Figure 15: Actual Input Files

```
                   { .Begin { Get First Ideal Record   <...See below
                   {          { Print Company Heading Info.  *
                   {          { Initialize Company Sales Total
                   {
                   {           { .Begin { Print Salesman Heading *
                   {           {        { Initialize S-S-Total
                   {           {        { Set Salesman-Name(e) to
                   {           {                     Salesman-Name
                   {           {
                   {           {            { .Begin { Print Customer Heading *
                   {           {            {        { Initialize C-S-Total
                   {           {            {        { Set Customer-Name(e) to
                   {           {            {                  Customer-Name
                   {           {            {
Process            { Process   { Process    { Process { Print Invoice Info.  *
Company            { Salesman  { Customer   { Invoice { Add Sales to C-S-Total
(1)                { (1,S)?1   { (1,C)?2    { (1,I)?3 { Get Next Ideal Record *
                   {           {            {
                   {           {            {       { Print Cust. Total Info. *
                   {           {            { .End  {
                   {           {            {       { Add C-S-Total to S-S-Total
                   {           {
                   {           {       { Print Salesman Total Info. *
                   {           { .End  {
                   {           {       { Add S-S-Total to Company Grand Total
                   {
                   { .End { Print Company Total Info. *

Tests: ?1-No More Salesman = True
       ?2-Old Salesman Name not = Salesman Name or ?1
       ?3-Old Customer Name not = Customer Name or ?2

     S-S-Total = Salesman Sales Total
     C-S-Total = Customer Sales Total


                     { Open Input Files
                     { Set End Invoice File to False
Get Ideal Record     { Set Abnormal End to False
(1)                  { Moves Spaces to Ideal Record
                     { Get Invoice Record  *
                     { Get Ideal Record    *


     * Processes not shown
```

Figure 16: A partial logical process structure

```
PROCESS-COMPANY Frame:
     Synonym: ()
     Attributes:
          Processes: COMPANY-BEGIN
                     PROCESS-SALESMAN
                     COMPANY-END
     Representation: A program algorithm
     Corporality: PROGRAM_CORP
     Value:
     Begin (*program*)
          COMPANY-BEGIN;
          Repeat
            PROCESS-SALESMAN
          Until No-More-Salesman=True;
          COMPANY-END;
     End. (*program*)



PROCESS-SALESMAN Frame:
     Synonym: ()
     Attributes:
          Inputs: IDEAL-RECORD
          PROCESSES: SALESMAN-BEGIN
                     PROCESS-CUSTOMER
                     SALESMAN-END
     Representation: a process algorithm
     Corporality: PROCESS_SALESMAN_CORP
     Value:
     Begin (*process*)
          SALESMAN-BEGIN;
          Repeat
            PROCESS-CUSTOMER
          Until Old-Salesman-Name Not= SALESMAN-NAME or
                No-More-Salesman=True;
          SALESMAN-END;
     End; (*process*)
```

Figure 17: The frame conversion of the Warnier/Orr diagrams - Page 1

```
PROCESS-CUSTOMER Frame:
     Synonym: ()
     Attributes:
          Inputs: IDEAL-RECORD
          Outputs:
          Processes: CUSTOMER-BEGIN
                     PROCESS-INVOICE
                     CUSTOMER-END
     Representation: A process algorithm
     Corporality: PROCESS_CUSTOMER_CORP
     Value:
     Begin (*process*)
       CUSTOMER-BEGIN;
       Repeat
         PROCESS-INVOICE
       Until Old-Customer-Name Not= CUSTOMER-NAME or
             Old-Salesman-Name Not= SALESMAN-NAME or
             No-More-Salesman=True;
       CUSTOMER-END;
     End; (*process*)

COMPANY-BEGIN Frame:
     Synonyms: ()
     Attributes
          Processes: GET-FIRST-IDEAL-RECORD
                     PRINT-COMPANY-HEADING
     Representation: a process algorithm
     Corporality: COMPANY_B_CORP
     Value:
     Begin
       GET-FIRST-IDEAL-RECORD;
       PRINT-COMPANY-HEADING;
       Company-Sales-Total := Zero;
       No-More-Salesman := False;
     End;

SALESMAN-BEGIN Frame:
     Synonyms: ()
     Attributes:
          Inputs: IDEAL-RECORD
          processes: PRINT-SALESMAN-HEADING
     Representation: a process algorithm
     Corporality: S_B_CORP
     Value:
     Begin
       PRINT-SALESMAN-HEADING;
       Salesman-Sales-Total := Zero;
       Salesman-Name-E := SALESMAN-NAME;
       Current-Salesman := SALESMAN-NAME;
     End;
```

Figure 17: The frame conversion of the Warnier/Orr diagrams - Page 2

```
CUSTOMER-BEGIN Frame
     Synonyms: ()
     Attributes:
          Inputs: IDEAL-RECORD
          Processes: PRINT-CUSTOMER-HEADING
     Representation: a process algorithm
     Corporality: C_B_CORP
     Value:
     Begin
       PRINT-CUSTOMER-HEADING;
       Customer-Sales-Total := Zero;
       Customer-Name-E := CUSTOMER-NAME;
       Old-Customer-Name := CUSTOMER-NAME;
     End;



PROCESS-INVOICE Frame:
     Synonym: ()
     Attributes:
          Inputs: IDEAL-RECORD
          Processes: PRINT-INVOICE-INFO
                     GET-NEXT-IDEAL-RECORD
     Representation: a process algorithm
     Corporality: PROCESS_INVOICE_CORP
     Value:
     Begin (*process*)
          PRINT-INVOICE-INFO;
          Add INVOICE-TOTAL to CUSTOMER-SALES;
          GET-NEXT-IDEAL-RECORD; (*logical record*)
     End; (*process*)



CUSTOMER-END Frame:
     Synonyms: ()
     Attributes:
          Processes: PRINT-CUSTOMER-TOTAL-INFO
     Representation: a process algorithm
     Corporality: C_E_CORP
     Value:
     Begin
       PRINT-CUSTOMER-TOTAL-INFO;
       Add Customer-Sales-total to Salesman-Sales-Total;
     End;
```

Figure 17: The frame conversion of the Warnier/Orr diagrams - page 3

```
SALESMAN-END Frame:
     Synonyms: ()
     Attributes:
          Processes: PRINT-SALESMAN-TOTAL-INFO
     Representation: a process algorithm
     Corporality: S_E_CORP
     Value:
     Begin
       PRINT-SALESMAN-TOTAL-INFO;
       Add Salesman-Sales-Total to Compnay-Grand-Total;
     End;



COMPANY-END Frame:
     Synonyms: ()
     Attributes:
          Processes; PRINT-COMPANY-TOTAL-INFO
     Representation: a process algorithm
     Corporality: COMPANY_E_CORP
     Value:
     Begin
       PRINT-COMPANY-TOTAL-INFO;
     End;



GET-FIRST-IDEAL-RECORD Frame:
     Synonyms: ()
     Attributes:
          Inputs: COMPANY File
                  SALESMAN File
                  CUSTOMER File
                  INVOICE File
          Outputs: IDEAL RECORD
          Processes: GET-INVOICE-RECORD
                     GET-IDEAL-RECORD
     Representation: a process algorithm
     Corporality: G_F_I_R_CORP
     Value:
     Begin
       Open Input Files
       End-Invoice-Files := False
       Abnormal-End := False
       GET-INVOICE-RECORD
       GET-IDEAL-RECORD
     End
```

Figure 17: The frame conversion of the Warnier/Orr diagrams
- page 4

```
INVOICE Frame:
     Synonyms: (sales-invoice)
     Attributes:
          NUMBER: an integer
          CUSTOMER-NUMBER: Restriction_on: CUSTOMER
               Attributes: (Number)
          SALESMAN-NUMBER: Restriction_on: SALESMAN
               Attributes: (Number)
          TIME: Specialization_of: DATE
          ITEM: specialization_of: DETAIL (1,d) d<=20
          TOTAL: a real
     Representation: a record (1,I)
     Corporality: INVOICE_CORP
     Value:

DETAIL Frame:
     Synonyms: (Item)
     Attributes:
          QUANTITY: an integer >0
          DESCRIPTION: a string(40)
          UNIT-PRICE: a real
          DETAIL-TOTAL: a real
     Representation: a record
     Corporality: DETAIL_CORP
     Value:

CUSTOMER Frame:
     Synonyms: ()
     Attributes:
          NUMBER: an integer
          SALESMAN-NO: Restriction_on: SALESMAN
               Attributes: (Number)
          NAME: a string(30)
          ADDRESS
          SALES: a real
          INVOICE (0,I)
     Representation: a record (1,C)
     Corporality: CUSTOMER_CORP
     Value:

SALESMAN Frame:
     Synonyms: ()
     Attributes:
          Restriction_on: MRKTING-EMPLOYEE
               Attributes: (NUMBER, NAME,
                    JOB-TITLE: default = salesman)
          CUSTOMER: (1,c)
          SALES: a real >= 0
     Representation: a record (1,S)
     Corporality: SALESMAN_CORP
     Value:
```

Figure 17: The frame conversion of the Warnier/Orr diagrams - page 5

```
COMPANY Frame:
     Synonyms: ()
     Attributes:
          MONTH-END-DATE: Secialization-of: DATE
               DD: Range (28-31)
     Representation a record
     Corporality: COM_CORP
     Value:



IDEAL-RECORD Frame:
     Synonyms: ()
     Attributes:
          MONTH-END-DATE: Specialization_of: DATE
          SALESMAN-NAME: String[30]
          CUSTOMER-NAME: String[30]
          INVOICE-NUMBER: Integer
          INVOICE-DATE: Specialization_of: DATE
          INVOICE-TOTAL: Real
     Representation: a Record
     Corporality: I_REC_CORP
     Value:
```

Figure 17: The frame conversion of the Warnier/Orr diagrams - page 6

```
DATE Frame:
     Synonyms: ()
     Attributes:
          MM: an integer range (1-12)
          DD: an integer range (1-31)
          YY: an integer
     Representation: a record
     Corporality: D_P_CORP
     Value:



EMPLOYEE Frame:
     Synonyms: ()
     Attributes:
          NUMBER: an integer
          NAME: a string[30]
          E-ADDRESS: Specialization_of ADDRESS
          DEPARTMENT: a string[9]
               range (Admin, Prsnl, Mrkting, Mfging, D-Prcsing)
          SALARY: a real
          POSITION: a string[20]
     Representation: a record
     Corporality: PRSNL_CORP
     Value:



MKTING-EMPLOYEE Frame:
     Synonyms: (Employee)
     Attributes:
          Specialization_of: EMPLOYEE
               DEPARTMENT = Mkting
     Representation: a record
     Corporality: MKTING_CORP
     Value:
```

Figure 18: A sample of frames showing relationships to the frames in the conversion example.

THE SYSTEMS RESOURCE DICTIONARY
A SYNERGISM OF ARTIFICIAL INTELLIGENCE, DATABASE MANAGEMENT AND SOFTWARE ENGINEERING METHODOLOGIES

by

RANDALL N. SALBERG

B. S., Kansas State University, 1978

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

The Main thrust of this paper is to propose the means by which developments in the areas of Artificial Intelligence, Database Management, and Software Engineering can be brought together to enhance database management and software development. The integrating tool proposed is a data dictionary system based on the Dynamic Data Dictionary System (DDDS) as proposed by Robert Philips.

To accomplish this task a study was done of developments in tools and methodologies in the three areas mentioned above. It was determined that a great deal of similarity exists between the functions and support provided by program library guides and data dictionaries. Similar findings were made for requirements definition languages and data definition languages.

An improved DDDS, called the Resource Management System (RMS), that is based on a dynamic data dictionary called the Systems Resource Dictionary (SRD) is proposed. The purpose for and some of the requirements of the system are established, and a proposal is made for the use of frames in developing the SRD's database as a knowledge base. Some general methods for supporting the software development process, using the RMS and a number of generic expert tasks, are suggested. Reasons for this integration of methodologies and tools are examined. Lastly an example of the use of frames for data definition along with a discussion of necessary implementation decisions is given.

Further research is suggested to prove the feasibility of developing a software tool of the magnitude of the Resource Management System, and of using frames to implement the System's data dictionary database as a knowledge base.