Introduction

"No scene from prehistory is quite so vivid as that of the mortal struggles of the great beasts in the tar pits.... The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skillful but that he ultimately sinks".

Frederick P. Brooks, Manager IBM OS/360 Project, The Mythical Manmonth [BRO75].

1.0. Overview

Somewhere between the concrete and the abstract lies the software domain: intricate, complex, intangible and unique. The software revolution - a dream of having catalogs of reliable, interchangeable and reusable components may be on the horizon, but there are still many irreconcilable shortcomings in established paradigms for software development. Escalating costs, coupled with the ever increasing demand for software, its poor reliability, and the eventual inability of managers to meet project goals or schedules are factors that have led to the software crisis, where projects slip imperceptibly but inexorably out of control\footnote{These problems can have far reaching consequences especially in view of the fact that software technology has now become a major economic force with the cost of in-house software in the United States alone amounting to $150 - $200 billion. In fact it has been estimated that by 1995 a 20\% improvement in software productivity alone will be worth US $90 billion worldwide [RAM84] [BOE87] [SHA90].} [COX90] [YOU82].

In an attempt to obviate these difficulties, in 1968, a group of eminent computer scientists gathered in Garmisch, Germany, to
participate in a now famous software engineering conference sponsored by NATO\textsuperscript{2}. Software engineering was formally defined to represent a means for "the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines" [NAU69]. Many people have opposed this view, calling it a fabrication that took away the uniqueness and magic of programming. As a result, great debates over whether the creation of software was an art, a science, or an engineering discipline, have been on for over twenty years [DJI89] [HOA84] [SHA90].

Nevertheless, the term software engineering has gradually gained acceptance over the decades throughout the software community, and has since been defined in many different ways as:

- the systematic approach to the development, operation, maintenance and retirement of software [IEE83];

- the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed within time and cost estimates [FAI85];

- the practical application of scientific knowledge in the design, construction and maintenance of computer programs and the associated documentation required to develop, operate, and maintain the delivered software system [BOE76];

\textsuperscript{2} In the midst of their discussions, Dr. Edward David remarked, "There is no theory which enables us to calculate limits on size, performance or complexity of software. There is, in many instances, no way even to specify in a logically tight way what the software product is supposed to do, or how it is supposed to do it" [NAU69].
The study of principles and methodologies for developing and maintaining software systems [ZEL78].

The implication of these definitions is that different techniques and methods can be organized, integrated and managed in different ways resulting in various process models or software development methodologies. Here, engineering refers to the selection, refinement and integration of these techniques and methods into a process model that is appropriate for the problem, application, and environment to which the methodology is applied. That is, project characteristics help us decide how to engineer the process to achieve appropriate product characteristics. Software engineering may thus be seen to encompass three key elements - methods, tools, and procedures - that enable the manager to control the process of software development and provide the practitioner with a foundation for building high quality software in a productive manner [PRE87].

Hardware and software engineering fall under the broader umbrella of systems engineering. Systems engineering concepts have been in use for decades for the design of computer hardware and have reached a state of relative maturity. Hardware design techniques are relatively well established, manufacturing methods are continually improved, and reliability is now a realistic expectation rather than a modest hope.

However, engineering techniques for software production are less well established and have only recently gained widespread acceptance. Numerous ideas have been borrowed from the general systems area, but there still remains considerable scope for the application of systems engineering concepts to various problems encountered in software development.
The primary focus of this thesis is on "front-end" software engineering tools, specifically for software engineering requirements analysis and design, and on software project resource estimation, management and control. Borrowing ideas such as Interpretive Structural Modelling from systems engineering, potential applications in the area of software requirements analysis and software design have been investigated [KUM89].

System Dynamics is another powerful systems engineering simulation tool which gives substantial insight into the dynamics of the system at hand and the use of this tool in conjunction with an "embedded" neural network has been researched for its potential application to "intelligent" software project resource estimation and control, a study based on the Putnam Resource estimation model [PUT78].

System Dynamics also has potential applications in linear and non-linear circuit simulation, which could have considerably important implications for the design and development of integrated CAD based software engineering tools for specific application domains. Investigations into the development of dynamic simulation models have been researched for non-degenerate and degenerate systems.

Continuing along the same lines, sensitivity studies appear to have important applications in the design of a class of networks that falls under the broader umbrella of connectionism, i.e., neural networks. The state space dynamics and time domain evolution of sensitivity state vectors of recurrent networks called Hopfield networks have been studied using system dynamics as a simulation tool. The case for minimal sensitivity design of these networks has also been investigated.
1.1. Software Life-cycle Paradigms
1.1.0. Generic Phases in Sequential and Non-Sequential Life-Cycle Models

Software systems go through essentially three generic phases during their life cycle - definition, development, and maintenance, as shown in Fig.1.0. The definition phase begins when the need for the product is identified and ends when the user is satisfied with the system specifications laid down by the team. This involves systems analysis, software project planning, and software requirements analysis. This phase is followed by a system development phase which ends up in coding different functional modules that form components of the software system. These software modules are then integrated into the final software system and tested for correct operation and completeness in terms of the user requirements specification. The last phase of maintenance includes activities such as correction of errors discovered during operation, making performance enhancements, adapting the system to rapidly changing environments, and adding minor new features.

Software engineering life-cycle paradigms provide a rigorous and systematic procedure for the development of software with a quality assurance so that it performs its intended task correctly. The procedures laid down ensure certain deliverables at pre-specified milestones which provide a means of estimation of various system-specific parameters and ensure product quality.

1.1.1. Sequential Software Life-Cycle Models
1.1.1.0. The Classical Waterfall Life-cycle Model

To illustrate the generic phases of the software life-cycle, the waterfall cycle has proven convenient as shown in Fig.1.1. The first
FIG. 1.0 Software Development Paradigms: A Generic Perspective
FIG. 1.1 SOFTWARE ENGINEERING LIFE-CYCLE MODEL
stage in the process is the understanding of the problem in question and the user requirements. Requirements include the context within which the problem arose, functionality expected from the system, and system constraints. At this point the managers and software specialists decide whether or not it is feasible to build the system. During this stage the specialists try to understand the requirements and attempt to define the specifications that would meet those requirements. These specifications describe the external behavior of the system - what the system must do, and not how - and must be checked for suitability, omissions, inconsistencies, ambiguities, and overall completeness. Then system design begins.

Design describes how the system is to be implemented so that it can meet the specifications. Since the whole system may be complex, the main design objective is decomposition. The system is divided into modules and their interactions specified. The modules may then be further decomposed into sub-modules and procedures until each module is at a primitive level, capable of easy implementation.

During the implementation or coding stage, each module is coded in a suitable language and individual modules or units are tested for proper functionality. Having tested each module separately, the system is integrated and an acceptance test carried out to ensure proper module interaction. Most of the testing is done after the entire system is completed.

3 Requirements may have been given by the end user, or, if the software system is embedded within a larger system, they may derive from the system requirements.

4 A large percentage of errors discovered during testing originate during the requirements analysis and product design phases. Also, requirements and design errors are much more expensive to correct - typically, 100 times more expensive than implementation errors. Clearly then, more of the software life-cycle effort needs to be directed towards requirements specification and design.
Once in operation the system must be maintained, which includes the fixing of errors discovered during operation, adapting the system to a particular environment and tuning it for better performance. Major changes in this phase may actually cause the system to "evolve"\(^5\).

The conventional waterfall model includes amongst its drawbacks, the requirement of a considerable amount of time for the development of system specifications. It delays the writing of code and permits very little "end-user" feedback until the coding stage which is rather late in the life-cycle, because "nothing's done until its all done". Late verification of user requirements can prove to be costly in most cases. During testing periods, the most trivial errors are found first, and the most serious ones last. "De-bugging" thus tends to be extremely difficult during the final stages of system testing and the requirement for computer test time thus increases exponentially. The model also fails to recognize the possibility that it may often be easier to change an existing system than to re-develop the specifications and design a new system. This concept of reusability plays an important role in the development of large software systems and needs to be incorporated into the conventional life-cycle model.

1.1.2. Non-Sequential Software Development Life-Cycle Models
1.1.2.0. Semi-structured and Structured Life-cycles

Not every organization uses the Classic project life-cycle. There is a growing recognition that techniques such as structured design, structured programming and top-down implementation should be incorporated into the life-cycle. The semi-structured life-cycle

\(^5\) The boundary between maintenance and evolution is fuzzy because what exactly constitutes a major change is a matter of opinion. Maintenance absorbs a large fraction of the cost incurred over the complete life-cycle - primarily due to misunderstanding of user requirements, erroneous debugging operations and poor documentation of specifications, design, code, and testing before release.
FIG. 1.2 STRUCTURED LIFECYCLE
uses top-down implementation strategies along with structured programming to build a software system. In a more subtle sense, top-down implementation actually implies that some design and coding go on in parallel, as do coding and testing. There is also some feedback between the user and the implementor on the specifications of the system which is critical to the successful design of a system.

Fig.1.2 shows the structured life-cycle [YOU89b], which is one step ahead of the semi-structured life-cycle. Here, there is no implication that all of activity N must finish before activity N+1. On the contrary, the network of dataflows connecting activities strongly implies that some activities may be going on in parallel. The structured life-cycle may be radical - if some activities proceed in parallel with others, or conservative - if activities proceed strictly according to sequence. Both these approaches are generally avoided, and a tradeoff between these extremes is adopted as the practical project implementation strategy.

1.1.2.1. Rapid Prototyping

Amongst the alternatives that have been suggested, one model is based on the concept of rapid prototyping as shown in Fig.1.3. In this context, prototyping implies the design of a new application while the feasibility of the design remains in question. Rapid prototyping may thus be viewed as a feasibility study that serves to demonstrate system aspects that are critical to the user.

The prototype does not usually meet all the requirements of the user. Although it implements only the most important aspects of the system, a critical feedback path from user to designer is provided before the final product is designed. In contrast, the classical
FIG. 1.3 THE PROTOTYPING LIFE-CYCLE

1. Identify basic needs
2. Develop working model
3. Demo in context, get refinements, etc.
4. Implement revisions
5. Yes, impact on prototype?
   NO, rigorous specification components
   YES, prototyping done
6. Detail components needed
   NO, rigorous specification components
   YES, clean up prototype and document
7. Preliminary design
8. Engineered product

Feasibility
- Guidelines
- Good candidate
- No
- Rigorous specification approach

No rigorous specification approach
- Implement revisions
- Yes, impact on prototype?
life-cycle provides this feedback only after implementation of the design, a problem which is obviated to a large extent in the semi-structured life-cycle. Normally it may be expected that systems designed with the prototyping approach would have superior interfaces and respect implementation limits better than those developed with the classical approach.

Rapid prototyping encourages the concept of reusability. In the reusability approach, software designers reuse as much existing software in the form of fundamental functional modules as is possible and then integrate these modules to form a working system. Reusability reduces software development costs, accelerates the development process and reduces testing needs. However, new applications for which little reusable software exists, precludes reusability and one must revert to conventional development approaches [JON84].

Conventional rapid prototyping is normally carried out with the expectation that the prototype is a requirements gathering tool and is to be discarded when system design begins. More recently, the concept of "evolutionary prototyping" has been proposed [CON89]. This software development process model starts out with a set of objectives that specify cost and quality associated with achieving goals [GIL85]. Big phases of the linear life-cycle are broken down into smaller tasks. "Critical success" features are selected because without them, the system will not be useful. The partial system is then built based on a design that is easy to change and adapt. Subsequent iterations add new features to the system, but the changes are never too large. Advantages in this development model stem from the fact that before such new features are added, an evaluation step affords the user repeated input into the development process and the danger of developing a system past the user is
SET GLOBAL OBJECTIVES
DEFINE BASIC OPEN STRUCTURE APPROACH
DEFINE PRELIMINARY EVOLUTIONARY PLAN

ENGINEER STEP
IMPLEMENT PLANNED STEP
DELIVER TO USER
EVALUATE PERFORMANCE

FIG. 1.4 EVOLUTIONARY LIFE-CYCLE

FIG. 1.5 SPIRAL MODEL OF THE SOFTWARE PRODUCT
minimized. Developers also receive frequent feedback as shown in Fig.1.4.

1.1.2.2. The Spiral Model: A Risk Driven Approach

Fig.1.5 shows a framework for the development of large software projects based on the Spiral model which is a risk driven approach [BOE88]. In Fig.1.5 the radial dimension represents the cumulative cost incurred to date, and the angular dimension the progress in completing each cycle of the spiral. Each cycle begins with the identification of the objectives of the portion of the product being elaborated, the alternative means of implementing this portion of the product, and the constraints imposed on the application of alternatives.

The next step is to evaluate the alternatives with respect to the objectives and the constraints thereby identifying areas of uncertainty which are significant sources of project risk. If so, the next step should involve a formulation of a cost-effective strategy for resolving the sources of risk such as prototyping, simulation, administering user questionnaires, analytic modelling, or combinations of these and other risk solution techniques.

If performance or user-interface risks strongly dominate program development risks, the next step may be an evolutionary development step. On the other hand if previous prototypes have already resolved performance or user-interface issues, and program development risks dominate, development follows the basic waterfall model modified suitably to incorporate incremental development [BOE88].

6 "Identifying and dealing with risks during the earlier stages of development may eventually help keep the cost of software down and help prevent software disasters" [BOE91]. Software risk management has now emerged as a new and powerful paradigm for the implementation of software projects.
1.1.2.3. Fourth Generation Techniques

Where application functionality is extremely well understood, approaches referred to as fourth generation techniques (4GT's) or the transform approach may be used. Fig.1.6. depicts this approach which emphasizes the use of tools that allow the developer to specify some characteristic of the software at a very high level. Ideally, such specifications would be supplied in "natural language", or in non-procedural fourth generation languages (4GL's). Subsequently, such high level programs may be translated interactively using user decision and rationale to lower level specifications following which the resulting specifications may be automatically compiled to object form. The final programs may require some fine tuning before an acceptable product standard is achieved. The purpose of these tools is to accept certain parameters to generate an application program. Such parameters could describe screen interaction and definition, data manipulation or other environment-dependent factors. Using 4GT's, programmers can generate programs with very few errors much faster than conventional approaches [BAL85].

However, unless functionality is well understood, 4GT's may prove difficult to use. Automated application generators typically employ fourth generation languages that are very expressive and simple to use for writing program functions. Numerous examples of application generators supporting database management, report writing, database query etc., have been cited in the literature [HOR85]. Notable amongst contemporary 4GL/4GT based systems is Oracle, a relational database management system [PER89].

(16)
Fig. 1.6 Fourth Generation Techniques

Fig. 1.7 The Expert System Life-Cycle
1.1.2.5 The AI/Expert System Based Model

Attempts to get the computer to shoulder more of the software engineering burden intelligently, are probably as old as programming itself. Such efforts relate to AI programming environments (which are the experimental apparatus necessary to support the science of how to bring about an effective interaction between different kinds of knowledge), studies of the software design process (which deals with discovering the kinds of knowledge used in the task of software design) and knowledge based software assistants (which incorporate the useful knowledge into practical systems).

The AI or expert system based paradigm [HAY83] shown in Fig.1.7 is a life-cycle concept which deals with the problem of not knowing exactly before hand what will go into designing and implementing a new system in a specific area. Solving such problems requires frequent feedback from all the development steps.

The first step is called problem area and scope identification which is done by the expert and the knowledge engineer. Resource needs, development goals and representative sub-problems are identified and the knowledge acquisition process is initiated. Subsequently, the knowledge is put into a conceptual framework. Representative concepts are formulated and information flows which take place during the problem solving process, are understood. This is followed by the identification of formal languages and tools onto which such concepts can be mapped. Rules, control and information structures formulated by the knowledge engineer representing formalized knowledge are implemented yielding a prototype program, a step which involves design, coding and testing. Next, the expert evaluates the performance of this prototype and provides the knowledge engineer with feedback for revision.
Knowledge based models using AI techniques are still in the development-cum-research stage. These are based on a combination of the above paradigms, and the work done in the AI expert systems realm. The computer acts as a partner in the system development, guiding the developer through the development process. The key is the separation of the software engineering knowledge domain from the application knowledge domain and the capturing of this information along with the process paradigm in the database. Rules are formulated based upon accepted software engineering techniques and subsequently encoded into expert systems. These systems are then used in conjunction with other expert systems containing application knowledge rules to form a specific instance of a software system [CHA87] [FRE85] [BAL83]. Future systems will focus on knowledge intensive tasks rather than on labor-intensive processes to spur programming productivity. Such systems could take as long as fifteen years to build, but contributions to such fifth generation techniques (5GT’s) could be significant.

In passing, it is worth noting that there are several important differences between traditional software engineering life-cycles and the life-cycle for knowledge based expert systems. Most software engineering projects assume that the problem is one of implementation rather than that of design. The rigid specifications and modularization imposed generally do not prove useful for projects using knowledge-based systems. The project should be thought of as a problem related to design rather than to implementation [WAT86]. The functional specifications of expert systems change as a wider body of test cases and field problems are covered by the systems behavior as a result of which they are difficult to detail out. Besides this, the style of program implementation and development is different for expert systems, which are grown incrementally rather than programmed. The program is interactive by nature and new knowledge units are formulated as the expert uses the program and applies it to new test cases. This contrasts with simply implementing code to meet a functional specification prepared in advance of implementation. Knowledge engineering tools more often require adaptation and evolve during the knowledge acquisition and implementation process. However, although rigid specifications may be difficult to impose, a preliminary functional specification of any AI based software system is normally in order, so that it may serve as a platform for the development of the system. This specification may be subject to change as the development proceeds and user feedback is obtained.

An AI based development methodology would probably support automatic application program generation where functionality or specifications would be provided to the system using natural language. Such environments would have to have extremely vast databases if they are to have widespread and general application, primarily because intelligent performance requires efficient access to large amounts of knowledge. The development of such databases is an extremely difficult task, and therefore such environments would necessarily have to be domain-specific [BAR85], where the database would contain knowledge about a particular application area. For example, an AI based application generator for VLSI software may have a sufficiently comprehensive database about the physics of silicon systems, VLSI design concepts, VLSI technology, and circuit theory. An application program for automatic routing in CMOS may then be developed for a specific technology by the AI based application generator.
1.1.3. A Perspective for Future Paradigms

The implications of artificial intelligence for software engineering are numerous. Although the ultimate goals of such research would be to be able to move from a natural language specification to a running program automatically, a middle of the run goal would probably be to be able to move from a set of formal verifiable specifications to the final program.

There are, however, more reasonable intermediate targets. Firstly, having better man-machine interactive systems. Considerable advances have been made in this area and there are numerous examples of so called "user friendly" designs. Some basic human factors for design would be: to make it difficult for the user to inadvertently cause disaster; modularize applications in terms of user tasks rather than for programming convenience; limit interactive access to when it is required; and use simple metaphors to explain task organization [NIC90].

Secondly, one should attempt the development of wholly formal specification languages which can completely describe the state of a system in concrete detail. Formal methods [WIN90] [GER90] [LEV90], have the advantage of concise accuracy, allowing little room for ambiguity and vagueness.

Thirdly, natural language research should move ahead to be able to develop front end systems that can completely understand languages the way humans do.

Fourth, a database of data representations will have to be built up to find a way of structuring domain knowledge so that the program
can analyze properties of alternate representations and can select representations for particular tasks and for particular needs.

Finally, there must be a database of strategies. An expert system that is going to do any important part of the software engineering job is going to have to know all the strategies for sorting and searching, and their advantages and disadvantages to be able to select the best strategy for a particular job [SIM86].

Various critical views of programs and their life-cycles and software design and development methodologies appear in the literature [LEH80] [BER81] [MAY90]. In fact, the software engineering paradigms discussed above are generally viewed as alternative approaches rather than complementary approaches. In most cases, however, the paradigms should be combined so that the strengths of each can be utilized on a single project. Fig.1.8 shows how one can integrate different paradigms during a single software development effort.

From a more pragmatic point of view, there is therefore perhaps the need for a combined paradigm based on the composite concept of "expert systems" and "systems modelling and analysis". Automated systems which would support such a paradigm would comprise the following four elements:

- an intelligent user interface such as a natural language processor;
- an information system or knowledge database;
- a model system comprising conventional CAD models; and
- a decision support system or an inference engine.
FIG. 1.8 COMBINING LIFE-CYCLE METHODOLOGIES
1.2. Software Engineering Research

Software engineering may be a relatively new discipline, and although practice is still about ten years behind research in the area [ZEL84], significant strides have been made since its conception.

1.2.0. Software Requirements Specification

One of the most fundamental problems faced by system programmers is simply understanding what the user wants the software to do. Although Parnas pointed out almost two decades ago in his now classic paper [PAR72a] that early recognition of proper functional specifications is important, it is only in recent years that software user requirements specification has emerged as an independent area of study in computer science [YEH80] [YEH84] [GEH86]. Parnas also pointed out the crucial difference between the specification of a system and the implementation of the specifications of the system, a concept which has now come to be known as information hiding: any specification scheme should provide only that much information that is necessary to program the system correctly, and nothing more [PAR72b] [PAR85].

Specifications are a means to formally define the behavior of a system or a system component [JAL89]. The fundamental problem of requirements definition is in fact the capturing of our minds' view and transposing it to a medium that can be communicated [CHA87]. With a poor, "fuzzy" statement of user requirements, even the best code and best testing will not produce a software system with fewer errors. Interestingly, 50% of the errors and 80% of the cost of error removal can be traced back to errors or misunderstandings.
between the user and the systems analyst during the requirements definition phase of the project [YOU89a] [YOU89b].

Although informal requirements specifications provided by the user are easy to read and understand, they are often incomplete, vague and ambiguous, and cannot be used to scientifically verify that a system constructed actually conforms to the specifications. In an effort to obviate these difficulties, formal specifications have been promulgated as a means for making communication about the functionality of a system more precise and less prone to errors due to misinterpretation [LIS75]. Formal mathematical methods provide a means of defining notions like consistency and completeness, and more relevantly, specification, implementation and correctness [WIN90]. Such specifications can be checked to some degree for completeness, redundancy and ambiguity, and can be used in program verification.

Amongst the formal notations which have been used to describe the syntax of program specifications are the Backus-Naur form (BNF) [NAU63] and its variations (Extended-BNF), the Vienna definition language [JON80], and syntax graphs such as the ones used to describe PASCAL's syntax [JEN75]. Other specification languages which have been reported in the literature are CLEAR [BUR86], OBJ [GOG86b], GYPSY [AMB86], the Program Development System (PDS) which uses Extensible Language 1 (EL1) [KLA86], and PAISLey, an executable specification language, accompanied by specification methods.

Completeness is a desirable property for specifications, although it is a hard problem and almost always difficult to guarantee, especially in the context of user requirements analysis [BAL86]. In this regard, one could probably check the consistency much better. One basic approach to check the completeness of an axiomatic specification would be to generate automatically from the specifications, an implementation that faithfully translates the specifications into an executable program. If the specifications are not complete, the implementation is not complete, and the behavior of all the sequences of valid operations on the data type are not defined [JAL89]. For the verification of programs, more or less standard paradigms exist in which the system description language is some standard programming language based on first order logic. A verification condition generator is then used to provide the assertions which if actually proved, will suffice to verify the desired specification properties [GOG86a].
analysis techniques and software tools [ZAV91]. More recently, automated tools supporting the requirements acquisition process have also been reported in the literature [REU91].

Formal specifications do not render informal specifications obsolete or irrelevant. Informal specifications are still necessary as an aid to the understanding of the system being designed, especially because formal specifications are hard to read and understand. In fact, formal and informal specifications complement each other [LIS86].

In this context, it has been argued that while we cannot afford to use natural language specifications, we also cannot manage to do without natural language explanations. Any formal structure is a hollow shell to most of us without a description of its intended interpretation. Formal specifications are meaningless without natural language descriptions of the intended usage of the various functions and parameters. On the other hand, once the formal specifications are understood completely, the use of natural language should be avoided to answer any further questions about the behavior of the program.

1.2.1. Software Design Methodologies

Over the past decade or so, numerous software analysis and design methods have evolved, most of which provide techniques for information domain analysis, problem partitioning and logical and physical representation of the system under development. More notable amongst these are the Data Flow Diagram (DFD) [PRE87] [YOU89b], Data Structured System Development (DSSD) [PRE87] [ORR77] [ORR81], Jackson System Development (JSD) [CAM86], Structured Analysis and Design Technique (SADT) [PRE87] [ROS77], and more recently
Object-oriented Analysis (OOA)/Object-oriented Design (OOD) [BOO86] [PRE87].

Data Flow techniques assume that information is transformed as it flows through the computer system. Such transforms may comprise a single logical comparison, a complex numerical algorithm, or even the rule-inference approach of an expert system. Bubbles are used to represent the transformations, and arrows the data flows. External entities may also interact with the system by acting as both sources and sinks of information. Information within the system is stored in data repositories referred to as data stores. Data flow diagrams may be abstracted or refined into a number of levels, each level indicating a certain degree of detail about the system in question. In this manner, data flow techniques can be applied to any computer based system regardless of size or complexity.

Like their data-flow oriented counterparts, data-structure oriented analysis methods lay the foundation for software design. In every case an analysis method may be extended to encompass architectural and procedural design for software.

Data Structured Systems Development (DSSD), also referred to as the Warnier-Orr methodology, has evolved to its present status from work originally done by Warnier [WAR81]. Warnier developed a notation for representing information hierarchy using three constructs of sequence, selection, and repetition, and demonstrated that software structure could be derived directly from the data structure. Orr [ORR81] extended Warnier's work to encompass a somewhat broader view of the information domain that has evolved into DSSD, which considers information flow and functional characteristics, as well as data hierarchy.
Jackson System Development (JSD) [JAC83] is a technique based on information domain analysis and its relationship to program and system design. Similar to DSSD, JSD focuses on the "real world information domain". After identifying the entities and their related actions in the application domain, actions that affect entities are ordered in time, and represented with Jackson Diagrams. A process model which represents entities and actions is then developed. Functions corresponding to defined actions are then specified, process scheduling characteristics assessed, and hardware and software specified as a design.

Amongst the automated tools that exist for requirements analysis, SADT is a structured analysis and design technique that has been widely used for system definition, requirements analysis, and system/software design [ROS77]. It consists of: procedures that allow the analyst to decompose software system functions; a graphical notation, the SADT actigram and datagram, that communicates the relationships between information and function within software; and project control guidelines for applying the methodology.

Software Requirements Engineering Methodology (SREM) [ALF85] is an automated requirements analysis tool that makes use of a requirements statement language (RSL) to describe "elements, attributes, relationships and structures". Elements comprise a set of objects and concepts used to develop a requirements specification. Relationships between objects are specified as part of the RSL, and attributes used to qualify the elements. Structures are used to describe the information flow.

The Program Statement Language / Program Statement Analyzer (PSL/PSA) was developed at the University of Michigan [TEI77], and
is a part of a larger system called Computer Aided Design and Specification Analysis Tool, CADSAT. PSL/PSA provide the analyst with capabilities such as description of information systems regardless of application area, creation of database for the information system, database addition and deletion, and production of formatted documentation and reporting on the specification.

1.2.2. Software Quality Assurance

Software quality assurance (SQA) has become an indispensable dimension of software development, designed to guarantee that the productivity requirements are fulfilled. It is "...the mapping of managerial precepts and design disciplines of quality assurance into the applicable managerial and technological space of software engineering" [DUN82]. SQA's scope must then be concerned with productivity (cost and schedule), process quality (which includes all development and maintenance activities), and product quality. SQA is often characterized by the four "W" questions:

What must we assure?
We must identify the quality characteristics of interest, state the relationships and weights amongst those characteristics, and define the degree to which they must be assured. (The fulfillment of qualitative characteristics such as reliability and performance cannot be guaranteed unless they are quantifiable).

When must we assure?
We must define the project milestones at which the quality characteristics of interest should be controlled, to assure that characteristics of interest can be met, or discover that they cannot be met, or to take early corrective action.
Which methods and tools must we use?

We must define which methods and tools are most adequate to gather information that support sound SQA activities.

Who must do the assurance?

We must decide what kind of professionals are most suited to do an effective quality assurance job, and how they fit into the overall project organization.

The effectiveness of SQA models and supporting methods depends on whether they are tailored to the specific needs and characteristics of a project and on whether both the development and maintenance processes and the resulting products are tractable. In this context, measurement is a very powerful mechanism for defining and analyzing software product and process quality. However, measurement must be goal oriented: it must be driven by the overall objectives of SQA [BAS87a] [BAS87b].

1.2.3. Software Measurement Techniques

Measurements on software can be broadly classified into two major categories: direct measures and indirect measures. Direct measures such as cost, effort, source lines of code (SLOC), speed, memory utilization, and size, are made objectively. Indirect measures are more subjective in nature in the sense that they are based on an individual's estimation or on compromised group consensus. Typical indirect measures would include function, quality, complexity (although some direct measures are available for this), efficiency, reliability, and maintainability.

Other broad classifications of software metrics have been proposed in the literature. For example, one could classify metrics into
those pertaining to the technicality, productivity, or quality of software. Alternatively, one could have size-oriented metrics based on SLOC estimates, and metrics which bypass SLOC. Much debate has ensued over the latter classification, and two schools of thought have emerged: one attempting to come to grips with the "lines of code" problem, and the other to bypass lines of code completely by either analyzing sub-elements of lines or by abandoning lines of code completely and switching to functions.

1.2.3.0. Size Oriented Metrics

Size oriented metrics are controversial and not universally accepted as the best way to measure software development productivity [JON78]. Problems faced by SLOC based models include whether one should consider "source lines of code", or "object lines of code". Such models become dependent to a large extent on the programming language in question, tend to penalize well designed shorter programs, cannot accommodate non-procedural languages, and their use in estimation may require a level of detail difficult to achieve.

Some of the more popular size-oriented models are macroscopic in nature: they deal with estimation techniques for software project resources such as cost and effort. These models are empirical and are dependent to a large extent on historical data. Empirical resource models could be static single-variable, static multi-variable, dynamic multi-variable or theoretical [BAS80].

More notable examples of empirical models are the Constructive Cost Model (COCOMO) [BOE81] [BOE84] for project cost estimation, and the Putnam Estimation Model [PUT78] [PUT89] for project effort estimation. In COCOMO there are three levels of models: the basic model, which is a static single-valued model that computes software
development effort and cost as a function of the SLOC; the intermediate model, which computes effort and cost as a function of both SLOC and a set of "cost drivers" that include subjective assessments of product, hardware, personnel and project attributes; and the advanced model which incorporates all characteristics of the intermediate version with an assessment of the cost drivers impact on each stage of the software life-cycle.

The Putnam Estimation Model is a dynamic multi-variable model that assumes a specific distribution of effort over the development life-cycle. The model has been derived from manpower distribution encountered on large projects (with efforts greater than 30 person-years, 1-10,000 KLOC). Extrapolation to smaller projects is also possible. Assuming a Rayleigh-Norden manpower loading over the development life-cycle, a fundamental equation called the software equation has been derived, which relates the effort, development time (time to full operational capability (FOC) of the software), and size of the project, using an overall productivity index (PI) for the organization based on historical data. Estimates for the size of the current project are typically done using delphi polling of experts in the application domain. The software equation is used in conjunction with another equation called the manpower buildup equation which, too, relates size, effort and development time, using a manpower buildup index (MBI) which is dependent upon the manpower loading profile that the project manager has selected. Using these equations, estimates for the effort and the minimum development time can be made with sufficient accuracy.

The Putnam Model can also be used for schedule risk planning. Using Monte-Carlo analysis and the uncertainties in the SLOC, PI, and MBI, risk profiles for the development time and effort can be generated.
which can help define a "joint probability plan" or delivery schedule for the software system.

1.2.3.1. Non-SLOC Oriented Metrics

Metrics in the second category (which do not use SLOC as a basis for modelling) first emerged with the work of Halstead [HAL77] [FIT78], and models based on these theories are microscopic in their viewpoint. In Halstead's software science, estimates for program length and volume are made from the sub-elements of lines of code by analyzing them separately. Lines of code are divided into separate data portions called operands, and functional parts called operators. Halstead uses these primitive program measures to develop expressions for overall program length, potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), and other features such as development effort, development time, and even the number of projected faults in the software.

The other "non-SLOC" model, due to Albrecht [ALB79], attempts to quantify the functions of programming. Function points are derived using an empirical relationship based on countable measures of the software's information domain and subjective estimates of software complexity. Information domain values are defined on the basis of the number of user inputs, number of user outputs, number of user inquiries, number of files, and number of external interfaces. Function points are calculated using complexity values associated with each of the above, and then applying complexity adjustment factors. Opponents of function points claim that the method requires "sleight of hand" as the computations are rather subjective in nature and that function points are just numbers with no direct physical significance.
A number of studies have attempted to reconcile the SLOC based and function point based models and both of them can be used accurately for software project planning [ALB83].

Research on software complexity measures based on graph theoretic approaches has also been carried out by McCabe [MCC76]. A program graph is used to depict program control flow and a complexity measure defined based on the "cyclomatic complexity" of a program graph for a module. One technique which can be used to find out the cyclomatic complexity metric is to determine the number of regions in a planar graph. This is done by counting the number of bounded regions and the outer unbounded region. Since the number of regions increases with the number of decision paths and loops, the McCabe metric provides a quantitative measure of testing difficulty and an indication of ultimate reliability.

1.2.4. Software Reliability

There is no doubt that the reliability of a program is an important element of the overall quality. If a program repeatedly fails to perform its intended task, it matters little whether other software quality factors are acceptable. Software reliability, unlike other factors can be measured using historical and developmental data. Software reliability has been defined in statistical terms as "... the probability of failure free operation of a computer program in a specified environment for a specified time" [MUS87] [MUS80].

In order to model software reliability, factors that affect the reliability itself must first be considered: fault generation, fault removal and the environment. Fault generation depends primarily on the characteristics of the developed code such as size and development process characteristics such as software engineering
methodologies and tools used, level of experience of personnel, etc.
In this context, code can be developed to add features or remove
faults. Fault removal depends on time, the operational profile and
the quality of the repair activity. The environment depends on the
operational profile. Since some of the foregoing factors are
probabilistic in nature and operate over time, software reliability
models generally employ random processes for estimation.

After about fifteen years of theoretical research, measures for
software reliability and models for characterizing it are gradually
moving into practice [MUS89]. Of the numerous models reported, more
notable ones are the basic execution time model which characterizes
the failures as a non-homogeneous Poisson distribution in execution
time, and the Musa-Okumoto logarithmic Poisson execution-time model
also based on a non-homogeneous Poisson process.

In the application of these models, a generic approach must be
adopted. Firstly, it must be determined whether software reliability
methods can help for the application in question. Secondly, it must
be decided what the user considers as a failure of the system.
Thirdly, engineers must answer some basic questions about the
hardware and software components, program evolution, and running
software at different speeds. Fourthly, they must determine the
operational profile for the proposed software system. The model for
estimating the reliability may then be chosen [MUS89].

1.2.5. Modern Structured Analysis and Computer Aided Software
Engineering (CASE)

Software engineering tools provide automated or semi-automated
support for methods that encompass a broad array of tasks such as
project planning and estimation, system and software requirements
analysis, design of data structures, program architecture and algorithm procedures, coding, testing and maintenance. When tools are integrated such that information created by one tool can be used by another, a system for the support of software development called computer aided software engineering (CASE) is established. CASE tools combine software, hardware and a software engineering database to create a software engineering environment.

Computer Aided Software Engineering (CASE) has undergone a series of evolutionary changes during the past 20 years. Structured programming and structured design were subjects of considerable interest in systems development organizations throughout the early and mid-1970's for bringing improvements in the software engineering field. Modern structured analysis has currently emerged at the cutting edge of CASE technology which will require integration with structured design and structured programming tools in the 1990's [YOU89b].

Modern structured analysis models the system at hand using a three fold modelling procedure.

Data-flow diagrams (DFD's) are used to model operational system aspects, primarily where functions of the system are considerably more complex than the data that the system manipulates. These diagrams model the processes, data flows, data stores, and any interactions with external entities that the system may have. In order to handle complexity, DFD's are usually "levelled", where each higher level bubble may be refined into a group of interacting lower level bubbles. Real time modelling is incorporated through control flows and control processes in the DFD.
Information support for DFD's is provided by data dictionaries (DD's) which are used to describe the meaning of the flows and stores in the DFD's, the composition of integrated packets of data, the composition of packets in stores, and the details of relationships between the stores of data. When bubbles in the DFD cannot be refined any further, their internal processes are laid down in the form of a structured representation called the process specification (PS). Process specifications are usually expressed in a form that can be easily verified by both the user and the systems analyst and which can be communicated to various audiences involved.

The second modelling tool is the entity relationship diagram (ERD) which is a network model that describes the stored data layout of a system at a high level of abstraction. ERD's are composed of objects, relationships, associative object type indicators and supertype/subtype indicators. The data stores of the DFD are modelled in the ERD. In the process of development of a system model it is not very important whether the DFD or the ERD is developed first. The choice is normally dependent upon whether the system is data-rich or function-rich, and sometimes both the models may be developed concurrently.

Often, an important part of specifying a system is the description of what happens when, and a third modelling tool - the state transition diagram (STD) - is used to highlight this time-dependent behavior of the system. In fact it is the real-time control flows and control processes that are modelled using the STD. The STD may also be used to model the human interface portion of on-line systems.

As we have seen, each of these tools focusses on a critical aspect of the system being modelled. Because the underlying system has so
many different dimensions of complexity, DFD's should focus only on functionality, ERD's on data relationships and STD's on timing characteristics, each without interfering with the other. However, in the end, the primary focus must be on model consistency, so that errors in implementation are minimized. This process is typically referred to as balancing of the models and CASE support exists here in a rather advanced way.

The methodology of modern structured analysis involves the building of what are now referred to as the essential model and the implementation model. The essential model is also referred to as the "perfect technology" model which assumes implementation using infinitely fast CPU's, zero energy consumption, zero down time, zero data transmission/retrieval delay.

As opposed to the Classical structured analysis life-cycle, where current physical and current logical model development is followed by the new logical and new physical modelling, modern structured analysis focusses on the development of the essential model comprising an environmental model, followed by the development of a behavioral model. The environmental model focusses on external system interactions with the environment in which the system must operate and is depicted by a context diagram (a single system bubble and external terminators) and an event list (which represents the "stimuli" or transactions to which the system is required to respond). The behavioral model shows the required behavior of the system regardless of how it is implemented. This is done through ERD's, DFD's, STD's, DD's, and PS's, using an "event partitioning" approach to generate the "first-cut" DFD, where every bubble has a function to respond to a single event on the event list. Bubbles dealing with common data are then levelled "upwards", and those that
can be refined, are levelled "downwards", leading to a concept of modelling called the middle out approach.

The user implementation model is the first part of the boundary between analysis and design and tackles such issues as identification of the "automation boundary", defining a man-machine interface and other operational constraints. Design issues deal with the development of a processor model, a task model and a code organization model, where the real challenge is figuring out how to map the essential model onto a real world architecture within the constraints defined by the user implementation model.

1.2.6. Object-Oriented Systems

More recently, object-oriented languages and object-oriented development environments are gaining popularity because of the fact that they support reusability to a high degree [MEY87]. Object-oriented development is a partial life-cycle software development method in which the decomposition of a system is dependent on the concept of an object. This method is fundamentally different from traditional functional approaches to design and serves to help manage the myriad of complexities encountered in large software-intensive systems [BO086] [ROB81a] [REN82] [SEI86].

Object-oriented programming techniques support the building of "software IC's" [LED85], which are reusable software components. The notion of objects communicating by means of messages is a key concept. Objects include data and key procedures called methods which describe a single kind of manipulation of an object's information. Objects can access the data directly and define a selection mechanism that can translate a message into a call to one of these procedures. The important thing is that methods cannot be
separated from objects. Sending a message to an object is similar to calling a function to operate on a data structure with one crucial difference - function calls specify not what should be accomplished, but how. The function name identifies certain specific code to be executed. Messages, by contrast, specify what is to be accomplished and leave it to the object to decide how to do it. The messages an object can respond to are known as its protocol, which is the external view of the object.

One of the key concepts supported by object-oriented systems is that of intelligence encapsulation: view the object from the outside to provide a natural metaphor of intrinsic behavior [REN82]. Encapsulation defines a data structure and a group of procedures for accessing it. Users access the data structure only through a set of carefully documented, controlled and standardized interfaces. Also, it is through messaging that the loose coupling of components is achieved and the division of responsibility between the user and the supplier defined and enforced. Message sending serves as the uniform metaphor for communication in the same way as the objects serve as the metaphor for processing capability and synthesis. Objects react to messages sent to accomplish processing by sending other messages to accomplish processing.

Apart from this, the concept of inheritance allows code to be stockpiled and thereafter reused as many times as is required. This is done by building a general class of objects, whose methods support the superset of operations expected to be used by special instances of that class. Special instances of the class may differ from other instances by having their own private data fields or behavior, in addition to the shared class variables, thereby representing an inheritance hierarchy.
Object-oriented languages employ a data or object-centered approach to programming. To fully support object-oriented programming a language must exhibit the following four characteristics: information hiding, data abstraction, dynamic binding and inheritance [PAS86]. Some languages which support such characteristics such as SIMULA 67 [GRA73] and ADA [ICH85] [BUZ85], and environments such as SmallTalk-80 [XER81], have been developed. Object-oriented extensions to structured languages have also been developed, notably Turbo C++ from Borland International.

1.3. Future Development Environments

The environment in which software is being developed is changing at an unprecedented rate, a dynamic change which also signals a present danger - the field already has many unsolved existing problems without having to consider the new ones brought by the change. In this respect, it is imperative to examine the environment and its possible restructuring, in order to be able to shape the changes intelligently rather than to merely respond to them [MUS85].

Most software engineering methods and automated support tools are adhoc and are usually available for a limited number of development languages and host computer systems. Collections of these tools on a common host system typify software development environments today. Each tool independently supports some phase or phases of the life-cycle. While collections of tools can support some consistent development methodology, the lack of architectural integration presents operational problems. Such collections of tools duplicate functions and are difficult to use and maintain. Also, because of the inconsistency and incompatibility of different tools in terms of their inputs and outputs, it becomes difficult to establish
interfaces between them, resulting in a severe fragmentation of data.

As applications become more complex, two forces will improve decision making and tool integration. The first force is the drive towards greater standardization in software development of large projects. The second force is the ever increasing application and sophistication of the software quality assurance (SQA) through more rigorous quality evaluation techniques. Software quality will increase as more sophisticated software quality technology blends with information control mechanisms [CAV87].

Future software development environments will integrate software tools and appear to the user as a single cohesive system of hardware and software that supports a particular life-cycle. In such environments, tools will work together synergistically to support activities in each phase and make transitions between phases smooth. The environments will have a single operational style, shared data, communication and interfaces. They will also support automated quality specification, quality analysis, traceability and change-effect analysis, automated documentation and project management [CAV87].

1.4. Potential Applications of Systems Engineering Tools

Concepts of systems engineering have long been used for modelling hardware systems, and although more recently some applications have been found in the realm of software engineering, considerable scope still remains for the application of the spectrum of system modelling tools that are available. Amongst these, Interpretive Structural Modelling [WAR74] and System Dynamics modelling [GOO74]
[RIC81] are two powerful concepts which have considerable potential for application to software engineering problems.

Individuals and groups often encounter complexities in their dealings with systems. The complexities exist because of the large number of elements and the interactions involved amongst them as well as the complex nature of these elements and interactions. Large scale system software also exhibits such complexity. However, a common characteristic of such complex systems is that each has some structure associated with it which may be explicit or implicit. Interpretive Structural Modelling transforms an unclear "object" system into a well defined "representation" system utilizing perceptions of the object system to develop a structured graphic model.

Most software engineering analysis (requirements specification) and design techniques employ graphical representations to communicate "mental models" of the system under consideration or development. However, the process of translation of these mental models into a structured chart representation is based mostly on one's intuition or heuristics and is done generally through informal group consensus. This thesis examines the application of ISM to the formalization of this process of translation (model exchange isomorphism). It indicates a procedure to approach the structured chart representation (digraph) of the system at hand, starting with the software scope, in the form of a primitive statement of need or an informal strategy for software realization, and refining it further in an attempt to reduce the level of abstraction and gradually introduce design details in an appropriate number of successive stages. In particular, it exemplifies the application of ISM to the object-oriented approach for the preliminary design of a numerical control (NC) software in two stages of refinement. This

(42)
yields a "minimal" structure for the analysis/design unlike the Booch diagram which fails to eliminate all relationships implied by transitivity. Potentially, ISM as a general modelling mechanism can also be used for deriving a variety of structured chart representations other than the Booch Diagram.

On the other hand, System Dynamics focusses on the structure and behavior of systems composed of interacting feedback paths. DYNAMO flow diagrams, and causal loop diagrams offer a convenient way to represent loop structures before the development of system equations. Flow diagrams consist of rates, levels and auxiliary elements organized into a consistent network. Causal loop diagrams identify the principal feedback loops without distinguishing between the nature of interconnected variables. DYNAMO equations provide a systematic format for the representation of simulation equations of the system that is under analysis or design.

Many static single-variable and dynamic multi-variable models are available for the estimation and management of software development under a wide variety of operational constraints, including cost and time. Putnam's model for software development is generally used for project resource estimation. A primary drawback of these models is the lack of insight into the dynamics of software project resource estimation and control. System Dynamics provides us with a simulation tool which offers insight into this dynamic behavior of software systems. Apart from this, in software project planning the estimation of certain variables eludes objectivity and as a result neural networks can play a major role in alleviating this problem. Investigations in this direction have led to the embedding of a neural network based decision support system within the system dynamics simulation model to provide a powerful support tool for software project planners. In the Indian context where software
development companies are growing in a big way, the application of this technique can prove to be very useful from both project planning as well as resource estimation and control point of view, helping managers understand the ever changing nature of software development.

Physical System Theory [KOE61] [SAG77] has long been used for systems modelling and analysis based on (i) domain independent graph theoretic representation of system interconnection structure, and (ii) the domain-specific characterization of component behavior (terminal equations). On the other hand, System Dynamics focusses on system behavior based on interacting feedback loops, and lends itself naturally to the simulation approach for the study of systems. It would stand to reason, therefore, to try and gain improved insight into the System Dynamics formulation and simulation by invoking the concepts and techniques of Linear/Non-linear System Theory. The thesis investigates the formulation of a Generalized Model for time domain simulation of non-degenerate and degenerate systems by bringing the modelling methodology of Physical System Theory to bear upon system dynamics simulation procedures. Neural networks have been used as a vehicle to exemplify the formulation of these models.

A classic example of a circuit where system dynamics can give insight is the Hopfield neural network. Hopfield networks are a class of neural network models where non-linear graded response neurons are organized into networks with effectively symmetric synaptic connections that are able to implement interesting algorithms, thereby introducing the concept of information storage in the stable states of dynamical systems. System dynamics provides a means to gain potentially useful insights into the behavior of such networks by studying the dynamics of the state-space trajectory
as well as time domain evolution of sensitivities of the states with respect to various circuit parameters eventually leading to the idea of minimal sensitivity design.

1.5. Neural Networks

Over the past three decades or so, the AI community has focussed its research on finding ways to solve problems that cannot be efficiently handled by digital means. Such problems include speech, character, text, equipment and human recognition tasks; financial analysis; database management; image and signal processing; dealing with fuzzy, chaotic or incomplete information, and some kinds of manufacturing process and quality control [OBE89]. Scientists are now beginning to think that the key to the solution of such problems possibly lies in biologically inspired neural networks, where an artificial neuron or processing element emulates the axons and dendrites of its biological counterpart with wires and emulates the synapses by using resistors and weighted values. We stand witnessing the birth of a new discipline.

The biological questions asked about neural computation have been primarily reductionist in nature, wherein it is tacitly assumed that if we understand in detail the operation of each molecule in a nerve membrane, we will understand the operation of the brain. In contrast, the complexity of a computational system derives not from the complexity of its component parts, but rather from the multitude of ways in which large collections of these "self-organized" components can interact and work in concert. Neural computation is an emergent collective property of the system, which is only vaguely evident in any single component element. Although a great deal of progress has been made in recent years, there is still no global
view of the principles and representations on the basis of which the nervous system has been organized.

Many people have proposed hypotheses about the way computation is performed in neural systems. To date, it has proved difficult if not impossible either to verify or disprove any given hypothesis concerning the operating principles of even simple neural systems. Major areas are so richly interconnected and computation within an area so intertwined, that there exists no good way of separating one function from another. As Carver Mead puts it, simple neural systems based on clear, obvious principles may once have existed, but they are buried in the sands of time. Billions of years of evolution have presented us with highly efficient, highly integrated, and impossibly opaque systems [MEA89].

1.5.0. Artificial Neural Network Research

Essentially, artificial neural networks are massively parallel interconnected networks of simple (usually adaptive) elements and their hierarchical organizations which are intended to interact with objects of the real world in the same way as biological systems do [KOH88], and which model some of the functionality of the human nervous system and attempt to capture some of its computational strengths [LIP87] [HOP88]. A neural network can be viewed as containing eight components as summarized in Table I.1 [WAH89]. Massive parallelism gives neural networks a high degree of fault tolerance, associative recall, and graceful degradation.

Amongst the fundamental objectives of neural network research, the first is to understand how the brain imparts abilities like perceptual interpretation, associative recall, common-sense reasoning and learning to humans. The second objective is to
Table I.1: Neural Network Components [WAH89]

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing units</td>
<td>Three types: input, output, &amp; hidden.</td>
</tr>
<tr>
<td>States of activation</td>
<td>Vector of the activation levels of the units in the system.</td>
</tr>
<tr>
<td>Output function</td>
<td>Function on the activation level of a unit which produces the units' output; may vary between units, but most systems are homogeneous.</td>
</tr>
<tr>
<td>Pattern of connectivity</td>
<td>Connections determine the performance and function of the system.</td>
</tr>
<tr>
<td>Propagation rule</td>
<td>A way of combining outputs of units and patterns of connectivity into an input for each unit; usually a weighted sum of the inputs and the excitatory (+) and inhibitory (-) connection strengths.</td>
</tr>
<tr>
<td>Activation rule</td>
<td>A function for determining the new activation level of a unit on the basis of current activation and inputs to the unit.</td>
</tr>
<tr>
<td>Learning rule</td>
<td>Three types: develop new connections, abandon old connections, modify weights; only the last has been pursued; almost all learning rules are based on the Hebbian learning rule.</td>
</tr>
<tr>
<td>Environment</td>
<td>In which the computing engine functions.</td>
</tr>
</tbody>
</table>

understand the subclass of neural network models that emphasize "computational power" rather than their biological fidelity. To achieve this objective, it is admissible to incorporate features within a model even if those features are not neurobiologically possible although there is some controversy over this narrow viewpoint [VEM88].

1.5.1. Historical overview

"Neural computers" have had a much longer history than generally believed. Brain and thought processes were first investigated by ancient Greek philosophers such as Plato (427-347 BC) and Aristotle
and later on by Descartes (1596-1650) and the eighteenth century empiricist philosophers. Heron the Alexandrian built hydraulic automata around 100 BC. From the claims of the 16th century Swiss physician Paracelsus to be able to create man, through to the demonstration of syllogisms by Stanhope (1753-1816) and the conception of the Analytical Engine of Charles Babbage, the centuries bristled with tales of automata that could move and talk like their human masters [KOH88].

Later on, according to Grossberg, although interdisciplinary studies did flourish for sometime, a schism of major scientific importance occurred towards the end of the nineteenth century. Interdisciplinary scientists were replaced by those who rarely had even a rudimentary knowledge of the other field. This schism was exacerbated by the explosion of scientific knowledge and its attendant requirement to specialize; and by the fact that available mathematical techniques were unable to support the non-local, non-linear and non-stationary phenomena that commonly characterized the psychobiological and related interdisciplinary sciences. This mismatch between the psychological phenomena and nineteenth century mathematics created an intellectual crisis for all theorists who might have wished to study the mind and brain [GRO88].

While most mind and brain experimentalist's ignored theory and most theorists looked for more hospitable frontiers, there arose a widespread tendency to interpret brain functions in terms of an ever expanding list of metaphors which included telegraph circuits, hydraulic systems, information processing channels, digital computers, linear control systems, catastrophes, holograms and spin glasses. All of these metaphors have been unable to explain a substantial database about the brain and behavior, as well as they
might, since none of them arose from a sustained analysis of behavioral and brain data [GRO88].

1.5.2. Neurophysiological Research

Over the past decade two physiological insights - that mental functions involve changes in synaptic connection strengths between neurons (first proposed by D.O. Hebb [HEB49]), and that the brain functional units are dispersed groups of cells - have underpinned numerous neural network models. However, none of these models explains how neurons come to be organized into functional groups in the first place [LEV88]. In an attempt to solve this problem Edelman proposes that fundamental cognitive processes such as perception, memory and learning are guided by epigenetic processes based on the Darwinian idea of variable populations and natural selection [EDE87]. "Neural Darwinism" is also the subject of a review by medical historian Israel Rosenfield [ROS88] [KIN88].

Numerous attempts have also been made to understand cognition by studying human system performance. For example, Anderson has discussed some of the hardware of real nervous systems and developed some models for cognition that try to work within the constraints of nature [AND83].

Investigations into memory storage and the molecular nature of associative memory in the marine snail *Hermisenda crassicornis* and in the rabbit, have recently been reported in the literature [ALK89]. Pavlovian conditioning is evident in the learning behavior of an enormous range of species. Inspite of the diversity of organisms, of behavior, and of stimuli that can become associated, the quantitative rules obeyed are surprisingly similar. This similarity of rules suggests that the mechanisms underlying
associative memory in neural systems have been conserved over the course of evolution.

1.5.3. Analytical Neural Network Models

At least three sources of neural network research approaches can be identified which have had a substantial influence on contemporary research: binary neural network models; linear models; and continuous and non-linear models. The stream of binary neural networks was initiated by the classical role of McCulloch and Pitts who were the first theorists to conceive the fundamentals of neural computing [CAR89] [MCU43] and also examine ideas in pattern recognition and the computation of invariants [PIT47]. They investigated threshold logic systems (with possible neuronal biases) where neuronal activity is the sum of the inputs that arrive via weighted path ways.

It was not long before learning and adaptation were added to the McCulloch-Pitts model resulting in Rosenblatt's Perceptron [ROS58] [ROS62]. Of these the most important is the "back-coupled perceptron", where weights are adapted according to whether the actual output matches a target output imposed on the system. The difference is treated as an error and used to adjust the weights according to some probabilistic law. This is referred to as back coupled correction. This class of models also introduced equations to change the weights through learning.

Mueller, Martin and Putzhath [MUE62] designed circuits which used McCulloch-Pitts logical operations and also extended their analysis to analog circuits for applications to acoustic pattern recognition.
Inspired by an interest in brain modeling, Widrow [WID62] developed his classical gradient descent adaline and madaline adaptive pattern recognition machines. Errors are fed back to adjust the weights using a Rosenblatt back coupled error correction rule [WID60]. Adaline and madaline have generated numerous technological spinoffs in more recent research on adaptive filtering and adaptive pattern recognition which have been summarized in [WID88] [WID90].

Anderson [AND68] initially described his intuitions about neural pattern recognition using a class of spatial cross-correlation functions. Kohonen made his transition from linear algebra concepts such as the Moore-Penrose pseudoinverse to more biologically motivated studies which he has summarized in his influential books [KOH77] [KOH84].

Continuous-nonlinear network laws typically arose from a direct analysis of behavioral or neural data. One distinguished modeling tradition can be traced directly to the influence of Mach [RAT65], which set out to model data taken from the lateral eye of the limulus or horseshoe crab.

The perceptron model has since found considerable application, especially in pattern recognition. Although two level perceptrons can sort linearly separable inputs into two classes, sorting more complex geometries requires multi layer perceptrons [LIP87] [ROS58] [MCL88]. The deficiencies in Rosenblatts "probabilistic" back propagation learning algorithm were modified and finally resulted in the later back propagation algorithm as we know it today [RUM86] [WER74].

More recent research in multi-layer perceptrons has focussed on studying the various hidden layer characteristics in the form of a
geometrical spatial separability problem [GOR88] [MIR89] [TOU89] [HUA91], and on different aspects of the back propagation training algorithm, its weaknesses, its strengths and possible modifications [BRA89] [HOR89].

One of the many models which followed the perceptron is the learning matrix [STE61] whose function is to sort, or partition a set of vector patterns into categories, and is generally regarded as a precursor to the concept of "competitive learning". A model comparative analysis of the learning matrix and madaline models can be found in a paper by Steinbuch and Widrow [STE65].

Apart from this, visual pattern recognition systems have been researched in detail by Fukushima et al [FUK88] [FUK83].

1.5.4. Learning and Memory Models

Sigmund Freud suggested that learning itself, through the increased electrical activity it prompts, modifies synapses, a hypothesis that was later formalized by D.O. Hebb [HEB49]. Hebb proposed that the strength of a synapse is increased by the simultaneous firing of cells on each side of it, and subsequently, an important development entailed adding a passive decay term to the Hebbian correlation term. These concepts led to the Hebbian learning algorithm which now forms the basis for a large number of contemporary neural network models [CAR89] [STA88].

However, recent working conceptual models arrived at through neurophysiological research investigations run somewhat counter to Hebb's assumptions about the nature of memory storage. Alkon and his colleagues [ALK89] have inferred from studies of the rabbit hippocampus that there is extensive local interaction between
post-synaptic sites. The spread of electrical and possibly chemical signals from one post-synaptic site to another — without activity or firing in the sites — seems to be critical for initiating memory storage.

The development of linear associative memory (LAM) models is a different line of research. Currently there are two opposing views concerning the encoding of information as a memory. Neurochemists think that information is stored in the neural cells in specific molecular patterns as in DNA. The second view says that memory works because of the functional and structural phenomena of networks. Signals do not affect every neuron internally. Rather, signals affect interconnections, namely synapses. Changes in synapses are the principal providers of memory. Some synapse changes may last for minutes or days (short term memory), while others may be permanent (long term memory).

Pioneering work has been done in the area of LAM's by Anderson [AND72], Kohonen [KOH72] and Nakano [NAK72]. Kohonen has also done pioneering work in distributed memory systems [KOH81], self organization in neural computing and topological feature maps [KOH82a] [KOH82b] [KOH87].

Hopfield's idea of storing information in the stable configurations of dynamical systems has been derived from concepts associated with a new class of materials called spin glasses [STE89]. The Hopfield net is standard of feedback networks and is made up of simple neurons in a heavily fed back interconnection matrix [LIP87] [HOP88] [HOP84] [HOP82]. They can be used as content addressable memories and optimization circuits [TAN86] [TAN87] [HOP85] [HOP86a] [HOP86b]. More recently, the information capacity of the Hopfield model [ABU85] and that of the Hopfield associative memory [MCE87] have
also been researched apart from other investigations into the performance of the Hopfield model [AIY90].

The Hopfield model is the exemplar dynamic neural network that has been chosen in this thesis to demonstrate the application of system dynamics simulation techniques to time domain voltage state vector and sensitivity state vector evolution studies, and minimal sensitivity design.

1.5.6. Real Time Models

Most of the models discussed so far require external control of system dynamics. In contrast, real-time models may or may not have an external teaching input, and learning may or may not be shut down after a finite time interval. Real-time modelling has characterized the work of Grossberg and his colleagues which is referred to as the theory of embedding fields [GRO64]. Two key components of embedding fields are instars and outstars [KOH80], which can result in heteroassociative memories or auto associative memories depending upon the kind of field coincidence.

A variety of learning algorithms for instars and outstars have also been studied. One of the most general of the outstar learning theorems is discussed in [GRO82].

Arbitrary space-time pattern learning and performance has been exhibited by multi-series outstar neural models called avalanches [GRO69]. These avalanche models find application in speech recognition and motor learning where space-time relations play an important role.
In more recent neural network models, "competitive learning" has brought the properties of learning into the real time setting. Investigators who have developed and analyzed the competitive learning paradigm include Steinbuch [STE61], Grossberg [GRO72][GRO76a][GRO76b] and Amari [AMA77]. The self organizing feature map [KOH84] and the counter propagation network [HEC87] are also examples of instar-outstar competitive learning models.

The analysis of the instability of feed forward instar-outstar systems led to the introduction of the Adaptive Resonance Theory (ART) due to Grossberg [GRO76b] with widespread applications in adaptive pattern recognition [CAR88]. ART Networks are designed to solve the stability plasticity dilemma - they are stable enough to preserve past learning, but nevertheless remain adaptable enough to incorporate new information, as and when it may appear. Two ART systems may be linked to form an associative memory.

Finally, in addition to probabilistic weight change laws, another class of such laws appears in more recent work under the name of simulated annealing [KIR83]. The link between simulated annealing and neural networks often surfaces in the Boltzmann machine [ACK85], which uses one such algorithm to update weights in a binary network similar to the model studied by Hopfield.

1.5.7. Hardware Implementations

Numerous attempts to implement either particular neural functions, or classes of neural nets as digital or analog LSI/VLSI have been made, and two distinct VLSI approaches have been reported [MUR88]. In the first, a high speed multiplier is used in a multiplexed mode to calculate the neuronal activity [GAR87]. Combined with a microprocessor, supporting memory and a custom communications chip
set, this forms an impressive neural hardware accelerator. A contrasting approach uses bitserial pipelined custom VLSI multipliers with low precision arithmetic to implement a similar hardware accelerator [MUR87]. Sub-threshold MOS device characteristics have been used to mimic the non-linearities of neural behavior in implementing Hopfield style nets [SIV86], associative memories [SIV85] and in auditory processing [MEA89].

Another major research group uses electron beam programmable resistive interconnects to represent synaptic weights between more conventional operational amplifier neurons [HUB86] [GRA86] [GRA88]. This work has also spanned an associative memory model in CMOS, which uses a digital memory to store weights and analog techniques to perform arithmetic [GRA87]. Other electronic implementations of associative memories are discussed in [MOO87].

An interesting development is the use of CCD/MNOS (metal-nitride-oxide-silicon) technology to store analog weights, and thus keep the entire neural system analog, and yet programmable [SAG86]. Other architectural and computational styles have been proposed using variously CCD technology [ARG87] and capacitor charge storage/switched capacitor techniques [SCH88] [TSI87].

Experimental VLSI chips of visual and motor subsystems, useful for artificial sensory systems have been designed and fabricated using CMOS VLSI technology [MAH89]. More recently, circuits improved by the architecture of the retina have been built by designers at Caltech [SIV87] [HUT88] [MEA89]. A recognizer for hand written digits performing computationally intensive tasks of live thinking and feature extraction has also been developed [DEN89].
Simple on-chip learning has been demonstrated in one design where digital circuitry was added to the interconnections to update the weights automatically based on local information [ALS87]. A learning chip (16 neurons and 112 interconnections) implements an algorithm developed by Kohonen [MAN88].

A new implementation of a VLSI fully interconnected Hopfield neural network with 2 binary memory points per synapse has been developed [VER89]. Digital architectures using stochastic logic have been used to simulate the behavior of Hopfield nets, with results similar to those found using standard analog neural networks or simulated annealing [VAN89].

Scalable architectures for the implementation of neural networks have also been designed and fabricated (46 neurons in 3 layers with 448 repeatedly programmable connections) as bidirectional associative memories in 3 micron bulk CMOS [BOA89].

Digital neuron type circuit elements based on logic NAND circuits for temporal and spatial summation, as well as thresholding have also been designed with multi-input multi-fanout capability [HAB89]. Apart from this, operational transconductance amplifier circuits suitable for modelling neurons with sigmoidal characteristics have also been reportedly developed [REE89].

In the future, practical limitations of VLSI technology may impose constraints on the density of interconnectivity, but advances in optical, opto-electronic, and holographic technologies may eventually help overcome this constraint. Two major approaches to realizing large scale opto-electronic neurocomputers have been researched: integrated opto-electronic neural chips with inter-chip optical interconnects that enable their clustering into large neural
networks; and nets with two-dimensional rather than one-dimensional arrangement of neurons and four-dimensional connectivity matrices for increased packing density with two-dimensional data [FAR89a] [FAR89b] [FAR87]. Optical implementations of associative memories and Hopfield nets have also been demonstrated [FAR85].

1.5.8. Neural Network Applications

Neural networks are being produced in the form of either neurocomputers (hardware that models the parallelism of neurons) or netware (software that emulates neurons and their interconnections on conventional serial computers). For example, NEC has developed a neurocomputer based on the back propagation learning algorithm. Another interesting example is the System for Nuclear On-line Observation of Potential Explosives (SNOOPE), which is a detection system that determines the existence of concealed plastic explosives in luggage and cargo, developed by Science Applications International, San Diego, California, USA.

Sensor processing and pattern recognition are among the many ways in which neural networks are being implemented. Applications include image processing, image compression, character recognition, and continuous speech recognition. These neural networks have been used for categorization of underwater sonar targets, and for handwriting character classification (NestorWriter from Nestor, Providence). They have also been used for robotics and autonomous vehicles, adaptive routing and switching.

Neural networks are also efficient at knowledge-processing tasks such as storage and retrieval of information in large databases, predictive modelling, and speech synthesis. This thesis demonstrates the potential application of neural networks for decision support in
the software project planning process where the estimation of
certain variables remains at best "fuzzy".

1.5.9. Future Research

Many elements about neural networks have yet to be worked into
place, not the least of which is how to model the human brain, where
our understanding still remains inadequate. As a result, no one has
really been able to come up with a "brain in a box", and the
widespread implementation and design of neural networks has faced
numerous problems.

Firstly, a neural network must be trained for a given application
and must be retrained when the system parameters change. There is no
systematic method to generalize a neural network trained for one
application and apply it for another application.

Secondly, much remains to be learnt about learning paradigms. All
the learning algorithms known today require an extensive amount of
training for good performance. Moreover, the learning speed depends
on the configuration of the neural network, which cannot be selected
systematically. Dedicated hardware to emulate various configurations
of neural networks and map the inner loop operations into analog
instead of digital circuits can improve learning speeds considerably. A million times improvement in speed has been
demonstrated using hardware emulation.

Thirdly, extremely large neural networks cannot be built with the
currently available technology, and consequently, it is unlikely
that a complete symbolic processor as intelligent as the human brain
can be built with neural networks alone in the near future. However,
the continuing evolution of technology could further extend the
available complexity level by another two orders of magnitude within the next decade. Silicon neural systems will be developed, but the design of these systems will be one of the greatest intellectual challenges of all time.

Lastly, with the limited size of neural networks, it is necessary to partition the problem so that part of it can be learnt by traditional methods and the other part by neural processing. However, the representation of knowledge in neural networks is drastically different from the procedural and declarative representation adopted in symbolic processors. Systematic methods to integrate them are still missing.

Some of the current major neural-network research programs focus on training/learning, scaling, improved network performance and identification of the network paradigm or pattern best suited to a specific application. There are dozens of paradigms and the number is steadily increasing. Other areas being researched include the development of networks that can choose their own architectures or configurations while focussing on issues such as optimal I/O format, training time, data pre-processing requirements, mathematical optimality, performance estimates, and debugging/diagnostics requirements.

However, some important questions which have yet to be answered by the research community are:

- How many layers and/or processing nodes are enough to solve a particular application problem?
- How creative should the system be?
- If it finds one good answer, should it continue to search for another?
What happens if neurocomputers base their conclusions on data other than what we use?

Once a neural network has reached a conclusion, what should it do about contradictory evidence? and

Can you achieve the same or better results with conventional technologies?

The information science describing neural network machines may have to change in order to provide a systematic framework to define how they will learn, pre-process and select input information. This would require the creation of a bridge between neurobiology and the information sciences to bring a much deeper view of computation as a physical process. It will also probably bring an entirely new view of information processing and of the awesome power of collective systems to solve the diversity of hard problems that are totally intractable by traditional computer techniques.

1.6. Outline of the Thesis

The primary focus of this thesis is on the demonstration of potential applications of structured modelling techniques, system dynamics simulation and neural networks to the area of software engineering.

Chapter 2 discusses the application of Interpretive Structural Modelling (which incorporates a formal knowledge acquisition methodology) to the formalization of the process of software requirements analysis, using the object-oriented approach as a vehicle. The potential of this approach in object-oriented design is demonstrated through an example on the preliminary design of an NC (numerical control) software system.
Chapter 3 discusses the dynamics inherent in software project resource estimation, management and control and demonstrates the capabilities of system dynamics and connectionist architectures to provide software project managers with comprehensive project planning support.

Chapter 4 looks at the area of system simulation for both non-degenerate and degenerate systems, and focuses on how we can bring the concepts of linear physical system theory and system dynamics modelling to bear upon each other. Non-linear systems are also discussed and comprehensive system dynamics simulation models developed.

Chapter 5 considers a classic circuit simulation example in the form of the Hopfield neural network and extensive simulation results are presented demonstrating the potential of system dynamics towards developing a better intuitive understanding of the dynamics inherent in strongly fed-back networks. Sensitivity studies of network states with respect to circuit parameters have been carried out and their potential application to minimal sensitivity design through optimization, of this class of networks discussed.

Chapter 6 draws out potential research avenues where the concepts embodied in this thesis could be carried further.