Interpretive Structural Modelling in Software Engineering Analysis & Design

"I know you believe you understood what you think I said, but I'm not sure you realize that what you heard is not what I meant”

Anonymous Software Customer

2.0. Introduction

Over the past few decades, in an attempt to alleviate the software crisis, numerous development paradigms have been suggested, and in more recent years, the "art" of software development is gradually becoming a "science" as formal rules and structures are adopted in an attempt to create a methodical engineering discipline. Development techniques are becoming increasingly structured and more attention is being given towards building a sound mathematical foundation for software engineering. Apart from this, the software engineering "experience" has caused the primary focus of the software development effort to shift from coding to design to requirements analysis and specification. The classic "waterfall" model of software development has been modified radically through use of modern concepts of rapid prototyping and reusability [MUR90].

More recently, with the ever increasing complexity and associated exacting reliability requirements of software systems, the object-
oriented development paradigm has gained considerable importance. Object-oriented programming focuses on the structure and intrinsic functions of objects that are manipulated, rather than on the steps that are taken to manipulate those objects, as conventional programming techniques do. The object-oriented paradigm supports both front-end software analysis and design by describing a cognitive process for capturing, organizing, and communicating the essential knowledge of the systems "problem-space", while providing guidance on using techniques to map this "problem space" to a "solution-space" model.

Object-oriented analysis (OOA) is usually introduced at the analysis step of the development process in order to perform suitable identification of key information items in the information domain, and key functional blocks in the function domain. However, OOA can also be considered a part of the object-oriented design (OOD) phase, where the identification of key objects from a suitable preliminary informal strategy is a natural precursor to the design and development procedure.

In the process of moving from a conceptual model of the "problem space" to the design concept of the "solution space" graphical representations have been found to handle the communication of mental impressions very well. However, as already mentioned, correct designs result only from correct requirements specifications. The primary focus of the development process thus falls on specifying these user requirements.

Knowing that requirements analysis is a communication intensive activity, most modern design methods start out by attempting to

---

9 The unique nature of the object-oriented paradigm lies in its ability to build upon three important software design concepts: abstraction, information hiding, and modularity.
capture a conceptual model of the problem/solution space in the form of structured charts. During the past ten years, a number of graphical "modelling" approaches have become popular in the analysis/design field. These include methods which emphasize functional abstraction such as the Jackson, Warnier-Orr, Gane-Sarson, Yourdon-DeMarco-Constantine methods, and others which emphasize data abstraction such as object-oriented development. These methods incorporate different ways of producing "engineering drawings" representing conceptual dependencies in a software system - like flowcharts - only much more powerful [YOU89a] [YOU89b]. For example, in OOA/OOD, Booch diagrams are used to represent program components [BOO82] [PRE87].

At present, however, transformation of the conceptual model into a more structured representation such as Booch diagrams is done using one's own intuition, domain-specific knowledge and experience. Formal structural modelling methods are not commonly employed to effect this transformation inspite of the fact that structural aspects of systems are dealt with implicitly. In fact, this structure is not consciously separable from substantive knowledge to an extent where fundamental questions about how basic knowledge elements are related may be asked.

Interpretive Structural Modelling (ISM) [WAR74] [WAR76] is one such method using which we can move from a conceptual model of the problem or solution space to a structured chart representation in a formal way. In fact, ISM also provides a rational basis for partitioning the graphical diagram into a hierarchy of levels ranging from axiological objectives (mission goals) at the top to policies (activities) at the bottom, and various realities in between as shown in Fig.2.0 [SAG77]. This chapter discusses the possibility of formalizing the process of diagrammatic model
FIG. 2.0 HIERARCHICAL STRUCTURING OF PROGRAM PLANNING LINKAGES.
building through the use of ISM and its associated model exchange isomorphism's (MEI's). With an overview of the problem of requirements analysis, interpretive structural modelling, and object-oriented techniques, the potential for application of ISM to OOA/OOD is brought out through an example of the preliminary design of a numeric control software system.

2.1. The Problem of Requirements Analysis and Specification

In recent years a number of software user requirements analysis and specification methods have been developed, all of which are related by a set of fundamental principles: the information domain as well as the function domain of a problem must be well represented and understood; the problem must be suitably partitioned in a manner that uncovers detail in a layered or hierarchical fashion; and the logical and physical representations of the system should be developed.

Specification, regardless of the mode through which we accomplish it may be viewed as a representation process. Requirements are represented in a manner which ultimately leads to a successful

10 The information domain contains three different views of data processed by computer programs. Information flow (which represents the manner in which data changes as it moves through a system); information content (representing the individual data items that compose some larger item of information); and information structure (which represents the logical organization of various data items).

11 Partitioning techniques are applied to handle problem complexity. In essence, partitioning decomposes a problem into its constituent parts. Conceptually, one establishes a hierarchical representation of function or information and then commences partitioning by: either exposing increasing detail by moving vertically in the hierarchy; or functionally decomposing the problem by moving horizontally in the hierarchy. As the problem is partitioned, interfaces between functions are derived. Data items that move across an interface should be restricted to inputs required to perform the stated function, and outputs by other functions or system elements.

12 A logical view of the software requirements presents the functions to be accomplished, and information to be processed without regard to implementation details. The physical view of software requirements presents real world manifestations of processing functions and information structures.
software implementation. Balzer and Goldman [BAL79], propose eight principles of good software specification:

- separate functionality from implementation;
- a process oriented system specification language is required;
- a specification must encompass the system of which the software is a component;
- a specification must encompass the environment in which the system operates;
- a system specification must be a cognitive model;
- a specification must be operational;
- the system specification must be tolerant of incompleteness and be augmentable; and
- a specification must be localized and loosely coupled.

The complete specification of software requirements is essential to the success of the software development effort. Requirements analysis is a software engineering task that bridges the gap between system level software allocation (based on functionality), and software design. It enables the system engineer to specify software function and performance, indicate software interfaces with other system elements and establish design constraints that the software must meet. It allows the software engineer to refine this allocation and represent the information domain that will be treated by the software. Requirements analysis also provides the software engineer with a representation of the information and function that can be translated into data, architectural and procedural design. In a more subtle fashion requirements also help the developer and the customer to assess software quality once the software is built. Fig. 2.1 portrays the software analysis task diagrammatically.
FIG. 2.1 OVERLAP OF THE ANALYSIS TASK
System specifications are generally written in a formal (normally non-executable) specification language, following which design and manual coding are carried out. In more modern environments, specifications are written in a computer executable language allowing them to be rigorously evaluated and validated using formal methods, code is generated automatically in a process that incorporates a design stage, and testing is unnecessary because specifications have been validated and the transformations that translate the specifications into code have been checked for correctness. Maintenance is usually carried out by re-writing requirements to generate updated code rather than patching existing code.

Requirements analysis is a communication intensive activity. Major problems encountered with analysis are concerned primarily with acquiring pertinent information, handling problem complexity, and accommodating changes that will occur during and after analysis.

Eliciting the required information from one or more human "experts" (clients, software consultants or users) for the purpose of building a knowledge base for software engineering analysis and design can be frustrating and time consuming. From the inherent nature of knowledge, it is subconscious and may be approximate, incomplete and inconsistent. Representation mismatch appears because of the difference between the way a human expert normally states knowledge and the way it must be represented in a computer program. If the software engineer elicits only that "expertise" as can be encoded in a given representation system, (such as logic programming, where facts are encoded as object-attribute-value triples and inferences as if-then rules), then significant expertise (visualized in a variety of knowledge structures such as lists, tables, hierarchies, flow-diagrams, networks, physical spaces, and physical models) will
be missed [OLS87]. This creates the problem of validation of the captured knowledge. Unless the interdependencies among pieces of knowledge are well understood, revisions in the knowledge base, in an attempt to correct a program bug (which may have caused an incorrect solution), can introduce subtle but far-reaching errors.

The aforementioned problems of knowledge acquisition can be avoided to a great extent if the software engineer attempts to generate the various knowledge structures in a generic form using the Interpretive Structural Modelling (ISM) technique (which has been developed in systems modelling as a technique to structure a complex system so as to help in communication of knowledge and planning) in conjunction with relevant contextual relation(s). The contextual relation(s) used makes(make) assumptions about the format of the underlying representations (which can be visualized by interviewing or interrogating the "experts"). Since the ISM technique can successfully generate a variety of "expert" knowledge structures it appears to be a viable and effective medium for knowledge acquisition and validation, transforming an unclear, poorly articulated mental model (or a set of interrelationships among elements) of a particular system into a visible, well defined model that should potentially help the software engineer considerably. By way of an illustration of such knowledge acquisition through ISM (as a generic technique), Appendix I provides an example of extraction of expertise inherent in a hierarchical knowledge structure.

2.2. Interpretive Structural Modelling

Interpretive Structural Modelling (ISM) exploits the fact that most complex systems have some form of structure associated with them. Beginning with an individual or collectively held conceptual model of a poorly defined object system, a particular transitive
contextual relation among the set of objects or system elements is embedded in a binary matrix model, possibly with computer assistance. The matrix model can be partitioned in a suitable way to enable the extraction of a digraph taking advantage of the various ways in which the information in the matrix can be partitioned. Such a digraph can then be augmented for the sake of interpretation by replacing the numerical representations of elements of the object system by their relevant features.

It is the interpretation of the embedded object or representation system in terms of the object system that results in an interpretive structural model. ISM involves the systematic iterative application of graph theory notions such that there results a directed graph representation of complex patterns of a particular contextual relationship among a set of elements. It thus helps transform unclear poorly articulated mental models of systems into well defined models [SAG77]. Interpretive Structural Modelling has its basis in mathematics, particularly in graph theory, set theory, mathematical logic, and matrix theory, essentials of which have been introduced in Appendix II.

The ISM process involves the identification of an element set (from the object system), a contextual relation set and certain binary relations. Using these, ISM obtains the structured form of a mental model through a series of transformations. Each stage of the methodology may be viewed as transforming a model from one format to another. The transformation of a model from one form to another form has been referred to as a model exchange isomorphism (MEI) [SAG77].

Prior to the application of the 5 MEI's, a necessary step is to establish pre-conditioning guidelines for interpretive structural
modelling. Table II.1 presents a convenient format for the pre-conditioning guidelines for interpretive structural modelling. These guidelines consist of the purpose or objective associated with developing an interpretive structural model, the perspective or role position which the modeler assumes, the mode the modeler will use in the activity, and a set of primitives with one appropriate relationship.

Fig. 2.2 outlines a sequence of MEI's and the intermediate models associated with each model exchange isomorphism. These five MEI's are:

1. Development of a structured-self interaction matrix (SSIM) which depicts clearly the contextual relationships that exist between the various objects. This is obtained by taking the group consensus regarding the existence of the said relation between each pair of elements. The logic defined in Table II.2 is used to effect this translation.

2. Transformation of the SSIM into the reachability matrix format is accomplished by transforming information in each entry of the SSIM into 1's and 0's in the reachability

Table II.1 Preconditioning guidelines for Interpretive Structural Modelling

<table>
<thead>
<tr>
<th>Theme</th>
<th>(Purpose or Objectives)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perspective</td>
<td>(View from what role position)</td>
</tr>
<tr>
<td>Mode</td>
<td>(Usually descriptive or perspective)</td>
</tr>
<tr>
<td>Primitives</td>
<td>P: (The element set)</td>
</tr>
<tr>
<td></td>
<td>Q: (The relation for ISM of P)</td>
</tr>
</tbody>
</table>
FIG. 2.2 MODEL EXCHANGE ISOMORPHISM
matrix. The logic defined in Table II.3 is used to make the matrix entries.

3. Transformation of the reachability matrix into lower triangular canonical form is accomplished by elementary row and column transformations\(^{13}\).

When explicit lower triangularization is not possible, an alternate formulation for the interpretive structural model of a given set of elements and contextual relations can be based on several partitions [SAG77] of the reachability matrix created in step 2, viz.,

(i) the relation partition,
(ii) the level partition,
(iii) the separate parts partition,
(iv) the disjoint and strong subsets partition of each level identified in (ii), and
(v) the strongly connected subsets partition as per identification in (iv).

After inducing the partitions described above, we may rearrange the reachability matrix to obtain a standard or canonic form (say, the lower block triangular form) ready for further transformations akin to steps 4 and 5.

---

\(^{13}\) In order to obtain the lower triangular form, those elements which belong to the highest level are entered as the starting elements of the lower triangular reachability matrix. Those elements which have already been entered into the lower triangular reachability matrix are ignored while scanning for the next highest level. This process is continued until all the elements are exhausted. Any element corresponding to which there is a single 1 entry is an element in the highest level.
4. Transformation of the lower triangular reachability matrix into a minimum edge adjacency matrix\(^{14}\). (In this transformation all possible redundant relationships are eliminated.) In this step the cycles are identified by entries of 1, if any, to the right of the main diagonal and are condensed into a single representative element.

5. Transformation of the minimum edge adjacency matrix into a structured chart representation. We may determine the structural model from the connectivity information in the minimum edge adjacency matrix. Finally, the subjective contextual relation is superimposed on the structural model and the Interpretive Structural Model is obtained, an example of which is shown in Appendix I.

### Table II.2 SSIM Data Matrix Entry Logic

<table>
<thead>
<tr>
<th>Condition</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (i R j) and (j R' i)</td>
<td>V</td>
</tr>
<tr>
<td>if (j R i) and (i R' j)</td>
<td>A</td>
</tr>
<tr>
<td>if (i R' j) and (j R' i)</td>
<td>0</td>
</tr>
<tr>
<td>if (i R j) and (j R i)</td>
<td>X</td>
</tr>
</tbody>
</table>

\(R\) : is related to  
\(R'\) : is not related to

\(^{14}\) The following two steps are followed to convert a lower triangular reachability matrix to a minimum edge adjacency matrix:

1. Make \(e_{ii} = 0\) for all \(i\).

2. Observe each row. As each \(e_{ij}\) entry of 1 is identified, the corresponding \(i\)th column is searched for entries of 1. For all entries \(e_{ki} = 1\), where \(k > i\), the corresponding entries \(e_{kj}\) are constrained to be 0.

Note: Transitivities between non-adjacent levels may be eliminated by a simple extension of this algorithm.
As the developed structures are tested against the conceptual model, added insight is obtained into the inherent structure of the system to the extent that certain aspects of the earlier model may need to be corrected. These aspects, such as entries in the reachability matrix may get illuminated to a point where a modeler is able to recognize an error in the group's concept of system structure. Consequently, ISM is an iterative procedure which allows the model to be continually refined. Satisfactory achievement of an interpretive model which accurately portrays all aspects of the original conceptual model as perceived by an individual or group responsible for creating the system, may be judged using group consensus or Delphi polling of experts.

2.3. Object Oriented Development

Object-oriented engineering techniques [BOO86] [EVB86] [COA90] [LOY90], have generated widespread interest over the past few years. In object oriented development, software function is accomplished when a data structure (of varying levels of complexity) is acted upon by one or more processes according to a procedure defined by a static algorithm or dynamic commands. To accomplish object oriented design, one needs to establish a mechanism for: 

(77)
(i) the representation of the data structure \((\text{objects}^{15})\); (ii) the specification of processes \((\text{operations}^{16})\); and (iii) the invocation of procedures \((\text{messages}^{17})\).

All objects are members of a larger class and inherit the private data structure and operations that have been defined for that class. Stated another way, a class is a set of objects having the same characteristics. An individual object is therefore an instance of a class.

Object-oriented approaches for problem/solution definition and partitioning are well worth applying as part of the software requirements analysis/preliminary design. In fact the definition of objects and operations is an excellent way to begin the analysis of both the functional and information domain.

In the context of this discussion, an object may be viewed as an information item, and an operation as a process or function that is applied to one or more objects. Object-oriented analysis/design provides us with a simple, yet powerful technique for identifying objects and operations. As will be seen, ISM lends itself naturally to a more formalized as well as structured object-oriented analysis (OOA) and object-oriented design (OOD).

Booch [BOO83] proposes the following steps for OOD:

---

15 An object is a component of the real world that is mapped into the software domain. For example, in the context of a computer based system typical objects might be: machines, commands, files, displays, switches, signals, alphanumeric strings etc.

16 When an object is mapped into its software realization, it consists of a private data structure and processes, called operations or methods, that may legitimately transform the data structure.

17 Operations contain control and procedural constructs that may be invoked by a message—a request to the object to perform one of its operations.
i) Define the problem.

ii) Develop an informal strategy for the software realization of the real-world problem domain.

iii) Formalize the strategy using the following substeps:

   a. Identify objects and their attributes.
   b. Identify operations that may be applied to objects.
   c. Establish interfaces by showing the relationship between objects and operations.
   d. Decide on detailed design issues that will provide an implementation description for objects.

iv) Re-apply steps (ii), (iii) and (iv) recursively until a complete design is created\(^{18}\).

The object-oriented approach to requirements analysis or preliminary design may be undertaken briefly in the following manner:

i) The allocated software is described using an informal strategy, which is nothing more than an English language description of the problem to be solved\(^{19}\). It may be stated in the form of a single grammatically correct paragraph.

---

\(^{18}\) It should be noted that the first two steps are actually performed during software requirements analysis.

\(^{19}\) As a general rule, the informal strategy has the following characteristics: (i) it is written as a single straightforward paragraph; (ii) it is written at the same level of abstraction; (iii) it focuses on what must be done to solve the problem, and not how the solution is to be accomplished; (iv) it does not have to contain all information uncovered during requirements analysis.
ii) Objects are determined by underlining each noun or noun clause and entering it in a single table. Synonyms must be noted. The objects thus noted should be partitioned according to the space to which they belong: the problem space \( P \) for requirements analysis or the solution space \( S \) for preliminary design.

iii) Attributes of all objects are identified by underlining all adjectives and associating them with their respective objects.

iv) Operations are determined by underlining all verbs, verb phrases and predicates, relating each operation to the appropriate object.

v) Attributes of operations are identified by underlining all adverbs and associating them with their respective operations.

Following these guidelines, an object-operation table is derived. Further, the object-operation table is refined to ensure that redundant, extraneous and abstract objects are eliminated, and that each operation is attached to a single object. Synonymous operations are combined and names of operations are expanded to make them more descriptive of processing function. Each object listed in this table should have at least one operation that acts upon it.

However, there may be cases when an object appears to stand alone, there being no apparent operation that requires information about the objects' underlying implementation. Alternatively, we may encounter an operation that appears to apply to no object in the table. In such a case, either the informal strategy is incomplete and an important object has been omitted, or an object belonging to the "problem" space has been allocated to the "solution" space, or the operation requires knowledge of a "stand-alone" object but the relationship has gone unrecognized, or the informal strategy has been written at different levels of abstraction.
OOD then defines each object as a program component (module) that is itself linked to other program components to form the complete program, giving rise to an important aspect of design quality called modularity. Program components are design abstractions and are represented in the context of the programming language in which the design is to be implemented. Subsequently, data objects and corresponding operations are specified for each of the program components.

Once program components have been identified, the evolving design is assessed critically to identify necessary changes. It is likely that the "first-cut" definition of packages will result in modifications that will add new data objects or even require the designers to return to the informal strategy to assess the completeness of the operations that have been specified.

The program components are then represented graphically to help establish interface connections and provide easily recognized patterns for design representation. Booch [BOO83] has proposed Booch Diagrams, as shown in Fig. 2.7 which shows the use of such diagrams to represent dependencies between program components.21

A recursive definition of the solution strategy is essential to achieve a level of design and data abstraction from which implementation detail may be derived. When details of objects are required to be added, the first step is to define the interfaces for each of the operations attached to an object and to identify internal structure for data objects. This is followed by stepwise refinement of each operation associated with the package.

---

21 The use of graphical notation for OOD may not be essential, but it does provide an indication of dependency among packages (objects and operations) that is lacking in program design languages.
2.4. On the Integration of ISM and OOD

In the earlier stages of object-oriented design, once the refined object-operation table is available, ISM may be applied by invoking each of the five model exchange isomorphisms mentioned earlier. Each information item and function shown in the ISM can be further partitioned, and the analysis steps applied iteratively. In the course of these iterations, interface details showing relationships between objects and operations are defined and decisions are taken on detailed design issues that provide implementation descriptions for objects.

The process delineated by the object-oriented approach could well be carried out with computer assistance. In the future, one might consider automating this process using knowledge-based systems, although interpretation using such systems would necessarily have to be domain specific.

A multi-purpose software package (the Interpretive Structural Modeler with an architecture as described in Appendix III) has been developed on the basis of the general algorithms described in this section and the preceding one.

The ISM procedure is explained in greater detail through an example of its application to the preliminary design stage of the software engineering development life-cycle.

2.5. Example

To explain the application of ISM to OOA/OOD we will use an example of preliminary design for numerical control (NC) software from Pressman [PRE87]. The first task for object-oriented development is
to write a one sentence statement of the problem with all noun and
noun phrases underlined:

Develop numerical control software for a machine tool that
reads NC blocks and operator commands and produces machine
servo-system control commands and a CRT display.

A preliminary OOA yields the object-operation table as shown in
Table II.4. Solution space objects can be separated out as shown in
Table II.5. In the table, although "NC software" may seem to be a
redundant object, its inclusion is necessary for the overall
completeness and understanding of the structured chart which results
from ISM. Before invoking the five MEI's of ISM, it is necessary to
define a basic contextual relationship. In this example it is:
"depends upon".

Table II.4 Preliminary Object-operation Table

<table>
<thead>
<tr>
<th>OBJECT</th>
<th>SPACE</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>NC software</td>
<td>S‡</td>
<td></td>
</tr>
<tr>
<td>machine tool</td>
<td>P‡</td>
<td></td>
</tr>
<tr>
<td>NC-blocks</td>
<td>S</td>
<td>read</td>
</tr>
<tr>
<td>operator commands</td>
<td>S</td>
<td>read</td>
</tr>
<tr>
<td>m/c servo-system commands</td>
<td>S</td>
<td>produce</td>
</tr>
<tr>
<td>CRT display</td>
<td>P</td>
<td></td>
</tr>
</tbody>
</table>

‡ S : solution space
‡ P : problem space

Fig. 2.3 shows the structured self-interaction matrix for the
preliminarily refined objects which is developed using the SSIM data
matrix entry logic as shown in Table II.2, and is actually the
FIG. 2.3 PRELIMINARY STRUCTURED SELF-INTERACTION MATRIX.

FIG. 2.4 PRELIMINARY STRUCTURAL MODEL
The result of the first model exchange isomorphism MEI 1. From the data matrix, the reachability matrix is derived using Table II.3. Equations (2.1), (2.2) and (2.3) show the reachability matrix, lower triangular form, and the minimum edge adjacency matrix respectively for the case in question, which are obtained using MEI's 2, 3, and 4. Fig. 2.4 shows the results of MEI 5 which yields the preliminary dependency diagram for the object modules that will form a part of the final software solution.

\[
\begin{bmatrix}
1 & 1 & 1 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\quad (1) \text{ NC software} \\
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
1 & 1 & 1
\end{bmatrix}
\quad (2) \text{ NC blocks} \\
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
1 & 1 & 1
\end{bmatrix}
\quad (3) \text{ commands}
\]
The preliminary design chart of Fig. 2.4 is at a high level of abstraction and each of the objects, control-words and commands can be refined further in an attempt to reduce the abstraction and gradually introduce design details.

As part of the informal strategy developed for NC-software a more detailed description of both these objects is provided in order to proceed with the refinement. Once again all noun and noun phrases are underlined.

Refined strategy for NC-blocks:

A central computer transmits NC blocks to NC software which reads each NC block and stores it in an NC program file. NC blocks are read from the NC program file and decomposed into control words for position and special control functions.

Refined strategy for commands:

Control words are processed and encoded into position control commands that are sent to the machine servo-system. Operator commands are input to the NC software via a keyboard interface. Operator commands enable the operator to insert an NC block into an existing NC program file, display an NC program file on a CRT, and execute an NC program.
After proceeding through the various steps of OOA/OOD and refining the object-operation table, the final objects selected for solution space implementation are as shown in Table II.6. With the help of Table II.2, MEI 1 is invoked for the selected objects of Table II.6, yielding the SSIM shown in Fig. 2.5.

Table II.6  Refined Object-operation Table

<table>
<thead>
<tr>
<th>OBJECT</th>
<th>SPACE</th>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>software</td>
<td>S^</td>
<td>abstract object</td>
</tr>
<tr>
<td>NC blocks</td>
<td>S</td>
<td>read-from-central-computer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read-from-NC-program-file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>decompose-into-control-words</td>
</tr>
<tr>
<td>NC program file</td>
<td>S</td>
<td>insert-into-existing-file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>stores-in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reads-from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>display-on-CRT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execute</td>
</tr>
<tr>
<td>control words</td>
<td>S</td>
<td>processed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>encoded</td>
</tr>
<tr>
<td>commands</td>
<td>S</td>
<td>send-to-servo-system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>input-via-keyboard</td>
</tr>
</tbody>
</table>

^ S: solution space

Using Table II.3, the reachability matrix of equation (2.4) is obtained from the SSIM of Fig. 2.5 (MEI 2). Equations (2.5) and (2.6) indicate the resulting lower triangular matrix and minimum edge adjacency matrix obtained using MEI 3 and MEI 4. In obtaining the minimum edge adjacency matrix, non-adjacent level transitivities have also been eliminated by a simple extension of the elimination procedure described in section 2.2.
<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th>O</th>
<th>O</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>A</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(1) NC - Software
(2) Commands
(3) Control-words
(4) NC - Program file
(5) NC - Blocks

FIG. 2.5 Refined Object SSIM Data Matrix
The resulting minimum edge digraph (ISM) from MEI 5 is shown in Fig. 2.6. The structure that results through ISM (Fig. 2.6), represents a minimal edge digraph unlike the Booch Diagram (reproduced in Fig. 2.7), in which transitive relationships are not removed completely. As per iterative application of steps 1-5 of ISM in section 2.2, each object (package body or program component) in Fig. 2.6 or Fig. 2.7 can be further partitioned in terms of data objects and corresponding operations.

2.6. Conclusions

Using the object-oriented approach as a vehicle, we have shown that ISM lends itself naturally to the concept of structured chart representation. The derivation of graphical representations can thus
FIG. 2.6 Refined Structural Model Depicting Object Module Dependencies
FIG. 2.7 NC SOFTWARE BOOCH DIAGRAM [PRE 87]
be made "algorithmic" in a broader sense to ensure more precise transformation of the "mental model" of the desired system. ISM is a generic form of knowledge structure which helps in extracting and validating the acquired knowledge in a variety of "expert" knowledge structures. It is a general modelling mechanism which could not only be used to derive structured chart representations such as Booch Diagrams through the object-oriented approach, but also be employed in the derivation of other structured representations such as those encountered in the context of JSD, DSSD, DFD, entity relationship diagram (ERD), or even state transition diagram (STD) for modern structured analysis/design [YOU89b].

The ISM procedure delineated in this chapter represents a structured means of approaching the diagrammatic representation of a mental concept. It thus presents software developers and users with an auditable track from the specification to the implementation level, i.e., changes at the specification level are automatically reflected in the diagrammatic representation of the conceptual model or the software scope. Configuration change control and management is thus inherently built into the procedure.

More generally, because nearly all graphical representations embody some underlying structure, it would be possible to apply ISM as a generic model to their derivation, irrespective of the life-cycle development stage under consideration. The alternate formulation of ISM referred to in section 2.2 is particularly useful for identifying hierarchical levels for modern structured analysis/design (with or without the OO Approach) in the context of large and complex systems and may contribute towards the further development of front-end or upper CASE tools while integrating them with the back end or lower CASE tools in the future.

(92)