Order Number 9235903

Object-oriented modeling for integrated computer aided process engineering: A software reuse approach

Mehta, Jaimin A., D.Sc.

Washington University, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY

OBJECT-ORIENTED MODELING FOR INTEGRATED COMPUTER AIDED PROCESS ENGINEERING: A SOFTWARE REUSE APPROACH

by

Jaimin A. Mehta

Prepared under the direction of Professor R. L. Motard

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

DOCTOR OF SCIENCE

May, 1992

Saint Louis, Missouri

GEN

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY

ABSTRACT

OBJECT-ORIENTED MODELING FOR INTEGRATED COMPUTER AIDED PROCESS ENGINEERING: A SOFTWARE REUSE APPROACH

by Jaimin A. Mehta

ADVISOR: Professor R. L. Motard

May, 1992

Saint Louis, Missouri

There are two major problems in realizing Integrated Computer Aided Process Engineering (ICAPE) systems and environments: object-oriented modeling of process engineering data, and integration of the existing stock of software for process engineering. This research investigates a novel approach based on software reuse to solve both problems.

The main contribution of this research is a new, software reuse approach to object-oriented modeling for integration, and a systematic software reuse methodology called "Reuse for object-orientation" or REO. The currently known object-oriented modeling methodologies prescribe development of a "universal" model for the application domain; thus they are practical only for new systems of limited

scope. The REO methodology, on the contrary, provides a short-cut for deriving object-oriented models from the existing stock of software. The past and current research in software integration have examined the black box approach, wherein the tool is interfaced with its input and output only, and the glass box approach, wherein the tool is interfaced with its internal symbols, but used in as-is condition in its entirety. The REO methodology, on the contrary, provides an approach wherein only parts of a tool are used in an object-oriented system. Presently, the REO methodology covers two software components: programming language descriptions and program descriptions.

The "experimental" subject includes parts of ASPEN, a chemical process modeling and simulation system, that is over a decade old and has over a quarter (1/4) million lines of program code. An object-oriented model is derived for this subject by following the REO methodology, and based on it a prototypical ICAPE system called "Icape-91" is designed and implemented in an experimental object-oriented system.

This research has identified and developed a novel approach to software integration and object-oriented modeling; an approach based on software reuse. Software reuse is a generalization of software integration. Software reuse can help in deriving object-oriented models from the existing stock of software. Software reuse can significantly assist software developers working in the field of ICAPE and ICAE in general. The successes of this research should motivate development, aided by REO, of large scale ICAPE systems or environments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Acknowledgements

I thank, foremost, my advisor Rudy Motard for providing me with generous support, advice, encouragement, and most importantly patience and an unparalleled environment for independent research. I also thank him for editing my rather incomprehensible drafts of papers, and for inordinately long loans of books. I extend my thanks to my doctoral examination committee members for their advice, suggestions, and stimulating discussions. The questions on design method from Bill Ball steered me to . . . the end of this dissertation. I also benefitted from discussions with Jim Schaaf. I am grateful to Alvin Larsen from Monsanto Chemical Company and Barry Flaschbart from McDonnell–Douglas Corporation for being generous with their time despite their busy schedules, and also for stimulating discussions. I also thank T. D. Kimura with the Department of Computer Science for my first introduction to the concepts of object–oriented programming. This work would have been incomplete but for the over–extended support from John Kardos, the Chairman of the Department of Chemical Engineering.

I also thank Yoshio Yamashita for assisting me in learning about the VSM system that he had developed. My gratitude extends to my colleagues—especially Dennis Patakas and Guillermo Simari, both of whom have graduated long since—at the Center for Computer Aided Process Engineering.

My thanks also extend to Bruce Hanna with Ontologic, Inc., for a mighty package of more than forty publications on object–oriented programming which he sent me, on his own accord, as a samaritan's gift. I also thank Mike Blaha with General Electric, Corporate R&D, for an introduction to the OMT notation; in retrospect, I believe the lesson was quite helpful in writing this dissertation.

I thank Andy Hazucha and Charu Malik with the Department of English Literature for prompt help in editing this dissertation.

Finally, I would like to thank all my family and friends for their encouragement, support and love. I am grateful to my Dad, Mom, and sibs for their love and encouragement. I am also thankful to my cousin Amita, her husband Deven, and my uncle Chandravadan for generous support for my studies in the U.S. To my friends from IIT Bombay--especially Ashutosh, Arun and Ajay--many a thanks for reinforcing and sustaining my interest in graduate school. My thanks also extend to many fellow students at Wash-U, especially Henry Erk and Kapil Talwar, for many things.

This work was supported partly in the past by the members of the Center for Computer Aided Process Engineering, the National Science Foundation under the grant number DMC-8619162 and CD12-8514513, and the Department of Chemical Engineering. This dissertation was prepared on superb software, Technical Publishing Software, from Interleaf, Inc.

1. Introduction

Computer Aided Engineering, hereafter referred to as CAE, is the practice of engineering and design predicated on computer aids. As a field of study, CAE is concerned with the development, management and use of computer aids for engineering and design. One of the unresolved problems of CAE is the "integration" of a huge stock of software and data. This problem is caused by the growing complexity, heterogeneity, and size of both software that are developed and managed, and data that are generated and managed. The study pertinent to this problem will be called Integrated Computer Aided Engineering, hereafter referred to as ICAE. The desired systems will be called ICAE systems. CAE restricted to the domain of process engineering is called Computer Aided Process Engineering, hereafter referred to as CAPE. Similarly, ICAE restricted to the domain of process engineering is called Integrated Computer Aided Process Engineering, hereafter referred to as ICAPE.

The problems of integration in CAE have been around since the 1970's. A few research groups have studied or attempted to solve them. The attention these problems receive in mechanical, electrical, and VLSI or electronics engineering has no parallel in chemical engineering. Less than a few chemical engineering schools have studied them. The Integrated Program for Aerospace Vehicle Design project, hereafter referred to as the IPAD project, which was undertaken during 1971–84, is the first major attempt to solve these problems in the aerospace engineering domain [Fulton, 1987]. Almost all companies in the aerospace or CAD/CAM industry participated in the IPAD project; it indicates the significance and urgency of integration in ICAE.

A project similar to IPAD has not been undertaken in process engineering

and would therefore be of world-wide interest. This research project will be referred to as the "Proto-ICAPE Project." Though the project is limited in scope, the expectation underlying it is that the lessons learned will aid and motivate an undertaking of larger scale project.

1.1 Motivation for Research

The IPAD project resulted in impressive achievements in, and many contributions to, the field of CAE; the majority, however, are obviously overshadowed by the advances of the past decade in many fields of computer science, especially in object-oriented, database, and software reuse technologies.† Thus, a similar project of prototypical scale based on new techniques may show better approaches to development of ICAE systems.

Object-oriented programming during the last few years has been effecting a change of paradigm, in the Kuhnian sense,‡ in the software industry. Originally object-oriented programming was mainly an amusement, but it has come to influence the fields of database systems [Dittrich & Dayal, 1986], artificial intelligence programming [Stefik & Bobrow, 1986], programming languages [Saunders, 1989], operating systems [Jones, 1978], network systems, and lately computer systems and hardware [Pountain, 1988]. Almost all areas and applications of computer science are vigorously applying object-oriented technologies. One is thus led to the question as to how object-oriented programming might affect ICAE and ICAPE.

---

† The term *technology* refers to the scientific study of techniques, and scientific knowledge can help in improving the techniques. The term is not used in the sense, currently in vogue, of a specific commercial implementations or tools.

‡ A *paradigm* is defined by Kuhn [1970] as a fundamental world-view held by a community of scientists which is eventually replaced by another during the evolution of scientific knowledge. For example, the superceding of Newtonian mechanics by relativistic mechanics is a *paradigm shift* or a *paradigm change* in physics.

Database technology has progressed from simple files through hierarchical, network, and relational approaches to extended relational, and object-oriented approaches. The area of engineering data management is still being researched, now with greater interest than before. The complex requirements of engineering data management cannot be met by Codd's [1970] "pure" relational approach despite its simplicity. Thus, many researchers have been studying extended relational and object-oriented approaches. Of the three known and viable approaches--relational, extended relational, and object-oriented--object-oriented approach is widely believed to be the way [Dittrich & Dayal, 1986]. Object-oriented approach also shows some promise of acceptable performance for the domain of engineering (see the 001 benchmark [Cattell, 1991], popularly known as the Cattell benchmark). For either of the two post-relational database approaches, extended relational and object-oriented, the essential first step is the logical design of a database. In the domain of process engineering, more research is needed in object-oriented modeling for engineering databases.

Software reuse and reverse engineering technologies are relatively new developments that began in the 1980's. One of the more widely followed approaches involves treating the software or tool (engineering or design software is often popularly referred to as a *tool* to generate and manipulate data or information in different forms) of interest as a black box and reusing it in as-is condition. A diametrically opposite approach is to treat the software or tool of interest as a "glass box" and analyze the segments of its source (segments as small as an expression in the source language) to develop a new model of the tool [Bachman, 1990]. A more economical approach would be a mix of these antipodal approaches as will be discussed in Chapter 3. In any case, the current knowledge

base lacks in software reuse techniques specifically targeted for object-oriented environments.

In sum, there is a strong need for research in the application of object-oriented and software reuse technologies for ICAE and ICAPE; a need also exists to develop new techniques.

1.2 Problem

There are two major problems in ICAPE, and in general in ICAE: (1) the management of data for process engineering, and (2) the integration of software for process engineering. Both can be solved by developing a "common" model for the domain of process engineering (the phrase "model for domain" summarily means model of data, knowledge, and activities of the domain), and a model for the existing software for process engineering. The commonality should span not only different software that are used for process engineering, but also the domain of process engineering as well as other disciplines that might be involved in an engineering or design project.

As regards modeling for a domain, most modeling methodologies commonly prescribe that one should develop a "universal" model for the domain by listing all objects and events that occur in the domain. The task entailed, unfortunately, is practical only for systems of limited scope, and not for the domain as extensive as the engagements of a typical engineering and design office. Even for a domain of modest scope, the problem of developing an object-oriented model based on the consensus of domain experts can be exceedingly difficult.

As regards modeling for integration of software, which is typically stocked in a design office, the known object-oriented software development methodologies have no recommendations for a lack of experience in the matter. The known

approaches (these are ad hoc and opportunistic, rather than being based on software engineering) have many limitations that will be discussed in the next chapter.

In this research a new thesis is advanced: software reuse approach can be used for object-oriented modeling of *both* the domain and software for the domain. This approach is the basis of a new methodology called Reuse for Object-orientation, hereafter referred to as REO.

One way to "prove" this thesis, perhaps the only way, is by demonstrating its application in a systems development project. To that end, a prototype ICAPE system called "Icape-91" has been developed following REO methodology. Prototyping is a quick, a not-too-expensive, and yet a powerful way to validate research in design.† From the known literature, the Proto-ICAPE Project is the first of its kind, as is the prototypical ICAPE system Icape-91. The "experimental" subject of REO in this project is ASPEN,‡ a software system used mainly for chemical process flowsheet modeling and simulation.

1.3 Outline

The following is a brief outline of this dissertation. The next chapter discusses in detail the problems of integration in ICAPE and reviews research projects that are similar to the Proto-ICAPE Project. Chapter 3 discusses some concepts of and approaches to software reuse, and describes the REO methodology with examples. Chapter 4 describes the development of Icape-91, mainly the steps of modeling that follows the REO methodology, and briefly a design and

---

† Is there any other way to validate research in "design"--a practice, not a phenomenon?

‡ "ASPEN" stands for "Advanced System for Process Engineering." The ASPEN system was developed at M.I.T. during 1976-81 under the sponsorship of the United States Department of Energy and many industrial participants.

implementation. A more extensive model and design is in the Appendixes. The last chapter summarizes the main conclusions of this research and makes suggestions for future studies.

A few asides: It is assumed that the reader knows about the basic concepts of, and is familiar with, the developments of object-oriented programming and its applications in many fields. Object-oriented programming is no mystery by now. The field is too broad for an overview in this dissertation, and a large body of literature is widely available. The best introduction to the subject is in the textbooks on SmallTalk-80 by Goldberg and Robson [1983]. For an extensive and general introduction to the area, see the texts by the following authors: Brad Cox [1986]; Bertrand Meyer [1988]; Grady Booch [1991]; Rumbaugh, Blaha, Premerlani, Eddy and Lorenson [1991]. The seminal ideas on hierarchical program structuring as they relate to object-oriented modeling or programming are in the paper by Dahl and Hoare [1972].

In the Proto-ICAPE Project, it is assumed that object-oriented paradigm is a superior one. *The research described in this dissertation is about the means for reusing the existing stock of software by turning it into an object-oriented software, not a study of the many benefits of object-oriented programming*; the latter has been investigated and belabored upon by many.

The notation of Object Modeling Technique, hereafter referred to as OMT, given by Rumbaugh et al. [1991], is used extensively in this dissertation. However, it is used from a programmer's viewpoint, in a manner rather different than suggested by its developers; for example, if a relationship has no more than one constituent with "many" multiplicities, then it is modeled as an attribute rather than let is stand on its own. For a reader who is familiar with object-oriented programming, the use of the OMT notation in this dissertation is easy to follow.

The notion of "object" has been around in computer science long before the introduction of object-oriented programming through Simula. However, the idea of object in object-oriented programming is rather different; at the minimum, it involves encapsulation, dynamic binding, and inheritance. It is in this sense that the term "object" is used in this dissertation; and not in the sense of encapsulation alone.

2. Integrated Computer Aids for Process Engineering

There are a many computer aids for various tasks in process engineering, but the majority are complex. It is widely believed that impediments to the productivity of engineers can be overcome by providing "integrated" tools that are centered on engineering data management systems. The approach and techniques described in this dissertation, although the field of their application is process engineering, are also applicable to other engineering disciplines. Such broad applicability is important, because process engineering is often part of inter-disciplinary projects typically found in the process plant engineering and construction industry. As stated in Chapter 1, the study pertinent to the problem of integration of computer aids for engineering will be known as ICAE, and the study restricted to the domain of process engineering will be known as ICAPE.

This chapter first describes briefly the practice of CAPE. Second, it discusses in some detail the problems that create the need for integration. Finally, it reviews related research projects by others, and compares them with the Proto-ICAPE Project.

2.1 Computer Aided Process Engineering

The term *process engineering* is used in a rather broad sense to denote various activities such as the synthesis, design, analysis, modeling, simulation, and process development and the management of these activities. These activities are carried out under different functions such as process research and development, process engineering and design, process operations, project engineering, cost engineering, etc. Usually they involve a group of people. These are undertaken in

varied industries such as refineries, chemicals, pharmaceuticals, pulp and paper, polymers, semiconductors, specialty and fine chemicals, and metallurgical.

Consequently, the term *computer aided process engineering* (or CAPE) denotes process engineering activities that are undertaken through computer aids packaged as turn-key systems, software systems, software subsystems, libraries (of compiled program units), databases, information systems, and others. The activities include management or organizational tasks such as production and management of engineering project data and documents, and coordination and collaboration with various other networked role-players and subcontractors. In other words, the major activity of CAPE *as a practice* should be management of data and information† that are related to process engineering; that this is realized only to a very small extent is perhaps an accident. The information in question includes that which belongs to engineering as well as engineering or project management (for example, vessel diameter and the "freeze date" of project data respectively).

2.2 Integration Problems

The complexity of tools usually diverts end-users' attention from the main task of generating useful design data and information. Furthermore, the multiplicity of complex tools has forced many users to choose one tool as a de facto

---

† The terms *data* and *information* are used synonymously to be consistent with the terminology in the area of database technology. Additionally, a valid argument can be made that information is a kind of data. Information generally means meaningful data or facts, and "meaning" is interpretation of data. In computers, interpretation of data is provided by procedures expressed in computer languages, and these procedures in turn are data for other procedures. Thus, data with procedures that interpret the data together constitute information; some say information is at a level above data.

standard even though others may be more suitable for certain tasks. Additionally, one needs tools to effectively manage an enormous amount of complexly inter-dependent data that are generated. In the present state of affairs in the area of CAPE, the problems of integration fall into the following three categories:

1. At the application level, process engineering activities need to be integrated to improve the process of design.

2. At the data level, the variety of process engineering data one uses need to be integrated in one model for comprehensive and sound management.

3. At the software level, various software tools are neither integrated nor easy to integrate into a coherent whole.

These are inter-dependent problems. For example, some of the problems of software integration are due to a lack of data management (shared and centralized management). This section further elucidates these problems.

2.2.1 Process Engineering and Design

Lately, many researchers and industry leaders have expressed that improvements in productivity in engineering and design are possible by eliminating time lags between various activities and "doing it right the first time." One way such improvements might be achieved is by undertaking certain design evaluations earlier than currently practised, and thereby preventing situations where problems have to be solved by adding systems and increasing the overall complexity. The developments in the design of nuclear reactors to provide safety features is well-known. The new generation of reactor designs such as Process Inherent Ultimately Safe, unlike the conventional Pressurized Water Reactor design, are inherently safe. These new designs use "passive" safety features, such as natural convection of emergency coolant laced with moderating compounds, that do not

require activating plant elements to counter failures in operations or of equipments [Golay, 1990].

This interest in reorganizing the process of engineering or design by promoting design evaluation from downstream stages of the design life cycle, generally known as "concurrent engineering" or "simultaneous engineering," is growing in many engineering disciplines and is being adopted in the industry. In computer engineering, Gupta et al., [1991] have shown that one can perform reliability analysis of computer system design during early conceptual stages of the design life cycle rather than leave the task until the testing stage. In mechanical engineering, the study of product design for manufacturability is an active area of research. In process engineering, there is a growing interest in evaluating safety, operability, controllability, and other "abilities" of process designs during early stages in the design life cycle. In chemical or process engineering, the idea of integrated engineering—bringing in all engineering and manufacturing expertise to the task from day one—has been around since the 1960's under a different name, Process Systems Engineering [Gundersen, 1991].

The problem of integrating the analysis or design techniques of process engineering from different stages of process design life cycle is beyond the scope of the Proto-ICAPE Project and this dissertation. Nevertheless, the brief explanation given above is for a reason besides completion's sake: integration of various analysis and design techniques in process engineering will eventually create a demand for integration of respective software and data.

2.2.2 Data

The importance of comprehensive and sound data management in the operation of engineering and design organizations has been elaborated and

belabored by many in the literature on engineering and process engineering computing [Benayoune & Preece, 1987; Eastman, 1981; Graham & Giambelluca, 1987]. Data management is expected to provide improvements in two critical success factors: the productivity of engineers, and the quality of engineering data and "products" (design documents can be regarded as products of engineering). The cost-benefit analysis used to justify its adoption in the industry is difficult in the face of poorly quantifiable and usually non-quantifiable advantages, let alone their valuation. However, other researchers and managers have made cogent arguments based on various surveys, and a few are summarized below.

A study by Imperial Chemical Industries PLC of chemical process engineering and design estimated that clerical "data pushing" takes about 10% to 70% of an engineer's time [Benayoue & Preece, 1987]. In contracting companies, around 35% of an engineer's time is spent in information handling. A survey performed by an international oil company suggests that engineers are tied up by data retrieval and manipulation as shown in Table 2.1 [James, 1984]. Thus, data management indeed requires at least 50% of engineers' time. These studies also highlighted the extreme complexity of relationships that exist within multi-disciplinary environments in project engineering. This complexity exacerbates data management problems and promotes resignation of project engineers and managers [Graham & Giambelluca, 1987]. Clearly, gains can be achieved in the productivity of engineers by providing necessary tools for information management.

According to Engelke [1987], detailed analysis of engineering changes often will reveal that up to 50% of all engineering changes are *corrections of errors* rather than changes in requirements. Mismanagement of information will eventually translate into project delay time, and both keep growing as errors propagate throughout the organization. The impact of an engineering decision

tends to increase as a project continues. It is important, then, that all decisions are good ones and are made at the earliest appropriate time. Thus, it is necessary to develop techniques and environments that will allow engineers to have quick access to complete, accurate, and consistent information to improve the quality and reduce the cycle times of process design, engineering, and development activities.

Table 2.1 Time Utilization of a Typical Engineer

Activity	Percentage of an Engineer's Time
Data Retrieval	20 - 25
Analysis/Calculation	20 - 30
Data Manipulation/Issue	35 - 40
Planning/Administration	15 - 20

Database management systems, hereafter referred to as DBMS's, were invented to unify several data files of large size (over gigabytes) or "integrate" data between various users and end-users. Their function is to maintain the integrity, consistency, sharability, security and access controls, and availability of data. An introductory description of these functions can be found in the first few chapters of the database systems text by Date [1986] and also in the new text by Cattell [1991].

File systems may be viewed as the most elementary kind of DBMS's, but they lack most of the basic, quintessential functions for data management. The differences in the definitions of input and output data can be handled by automatic translators (see Figure 2.1). However, this solution leads to other problems. First, the effort required to develop such programs, even if based on neutral formats, is combinatorially expensive; it requires two translators for each pair of views that is implicit in the two programs. Second, a significant problem is the *loss* of information due to a lack of one-to-one mapping between different data definitions and interpretations. Third, the changes in data definitions that may be required from time to time would lead to a breakdown of both the translator programs and communication within the program as depicted in Figure 2.1. (The subprograms of a program are said to be tightly coupled if, for example, they share data formats. For more details on the "coupling" and "cohesion" of software design, see the text by Myers [1978].) The translator-based solution to the problem of sharing data is well understood and easily implemented for short-term needs, but in time many such encumbrances can only turn the system into a Rube Goldberg† contraption for both the system manager and developer. Blaha [1984] in his dissertation describes a sour experience of using a file system in lieu of a DBMS. Since files are based on fixed formats of data organization, Blaha developed customized programs to read and write data and used editor programs to automatically update data that were written and later retrieved by other programs. However, errors easily crept in due to modification of data in the files (a

† The idiom "a Rube Goldberg" means an incredibly complicated, impractical scheme or device. It is coined after the American cartoonist Reuben Lucius ("Rube") Goldberg [Americana, 1989].

modification easily done by an end-user through an editing program), and consequently the retrieval program "bombed out" (sic).

The file systems were thus replaced by DBMS's first in the MIS departments of various organizations. The programs that are integrated with a DBMS obtain input from, and save output to, databases that are instantiated from global shared logical models of data; through such a database the programs communicate and share information. The integration via DBMS's attracted attention of researchers in CAE and CAPE; presently, customized file systems are used instead of DBMS's.

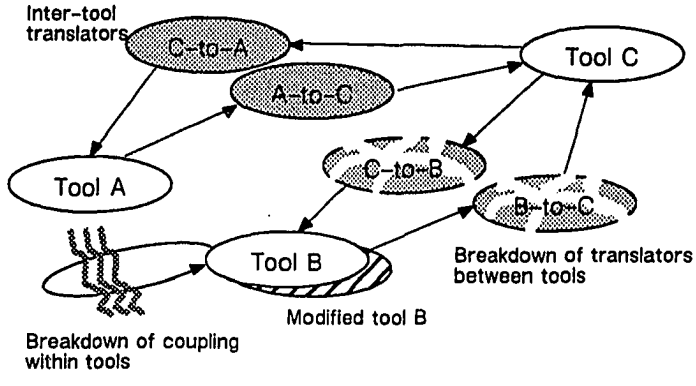


Figure 2.1 The Brittleness of Translator-based Data Integration between and within Tools

In short, a now commonly accepted principle in CAE and CAPE is that DBMS's should be the cornerstone of any organizations engaged in engineering or design. The major activity of CAPE as a practice, as discussed in Section 2.1, should be the management of process engineering data or information.

2.2.3 Software

The integration of software is another level of integration in ICAPE. More generally, one demands integration of various components of software including specifications, requirements, languages, and documentation.† Software systems are usually developed as packages dedicated to a specific task or function. However, the needs of a user or end-user usually cannot be met by a single software system: different software are used as needed for different functions or performance requirements. Thus, a need arises to "integrate" one software with another. The majority of CAD/CAE systems are closed packages: the user is "locked into" the system, although the user may be required to employ it as a tool only within a specific design methodology, design management practice, or office procedures.

Concurring views are expressed by others. Bushnell [1988] suggests that for VLSI CAD there are too many languages with too many commonalities beneath too many differences that are often far too trivial. Cifuentes [1987] has indicated that software for process modeling is too rigid and not easy to interface with other software, at least for programmers other than the original developer. According to Gadient [1987], the most fundamental problem in Integrated Computer Integrated Manufacturing is software integration, especially that of the existing stock. The best summary of this problem, true to this day, is stated by Terry Winograd [1979] as follows: "*The main activity of programming is not the origination of new*

† *The Prentice–Hall Standard Glossary of Computer Terminology* [Edmunds, 1985] states the following definition: "The program that makes a computer system function. Software consists of the operating system, all sorts of procedures, routines, specialized programs, including translators and utility programs. Software includes application programs as well. Software includes all related documentation, including manuals and instruction material."

independent programs, but in the integration, modification, and explanation of existing ones [italics in original, underlined mine]" (p. 392).

The problems of software integration are non-trivial and multifaceted. Some are attributable to a lack of data sharing or exchange between two different but related software. Others are attributable to differences in languages, run-time environments, user interfaces, standards incorporated, and requirements (for example, batch versus interactive or incremental systems). Unfortunately, the problems of integration are intermingled with the complexities of software, a lack of modularity, and difficulties in modification. To make matters worse, the majority of software are not developed through sound and formal practice: almost all are crafted.

The general approach to solving every problem associated with software integration is on a case-by-case basis and by employing various new and old tricks. One is then left with no new knowledge that can be applied to new problems, or the trick itself may cause a new set of problems at a later time—once again, leading one to a world of Rube Goldbergian contraptions.

In sum, the problems with integrating software are complex and interdependent. They are *not* solved by a simple "joining" of two black boxes (that is, merely by connecting their elements) and certainly not through data integration alone.

2.3 Solving Integration Problems

Of the three levels of integration in ICAPE discussed in the preceding section—applications, data, and software—the focus of this research is on the last two. This section explores different ways of solving the problems of data and software integration. First, this section examines the shortcomings of the existing

approaches. Next, it describes briefly a new approach that is first proposed and adopted in the Proto-ICAPE Project.

2.3.1 Existing Approaches

Engineering data has many peculiar characteristics unlike those of business or commercial data. Thus the database technology (relational database systems† superceded hierarchical and network database systems) that was developed for the business or commercial world was unadaptable for engineering [Blaha, 1984; Bray, 1987; Deltz, 1988; Fulton, 1987; Patakas, 1988]. Since the 1980's, extensive research has been undertaken in object-oriented and extended relational database technologies for engineering applications [Cattell, 1991; Joseph, Thatte, Thompson and Wells, 1991; Silberschatz, Stonebraker and Ullman, 1991]. As part of this research project, a review of post-relational approaches--extended relational and object-oriented--from ICAPE perspective is given by Mehta and Patakas [1988]. Although object-oriented database technology is researched and developed to a lesser extent than relational database technology, some object-oriented database management systems have demonstrated superior performance on certain benchmark problems [Cattell, 1991]. It is safe to assume that ICAE systems will be predicated on object-oriented database systems; at least, some ideas of object-oriented programming will take hold.

Of interest in this research is the first step in the design of an object-oriented database: object-oriented modeling. The analysis phase of all data modeling methodologies starts with documented and/or undocumented knowledge of the subject domain. Such an approach will be referred to as the "modeling from scratch" approach. For instance, Chen's entity-relationship modeling involves

† For a comprehensive text on relational database systems, see Date [1986].

identifying various physical or abstract entities and their relationships in the "real world." In object-oriented modeling, one of the objectives is to generate a classification scheme for objects. But there are potentially numerous principles of classification that may be used in modeling. Booch [1991, p.138] gives an interesting example based on research in conceptual clustering: for a mere "toy problem" of classifying ten trains, the modelers came up with ninety-three (93) different classifications! One can only imagine the difficulties one would face in modeling a domain as extensive as process engineering and allied fields, and the complexities of testing the models at that. On the other hand, for ICAE the currently available large stock of software raises doubts regarding the economics of modeling from scratch, given the difficulties in economic quantification and valuation in software engineering. The current situation is depicted in Figure 2.2. An economical alternative would be modeling based on the existing stock of software (see the graphical arrow comprised of tiny circles in the center of Figure 2.2). Such an alternative is developed in this research and discussed in subsequent sections.

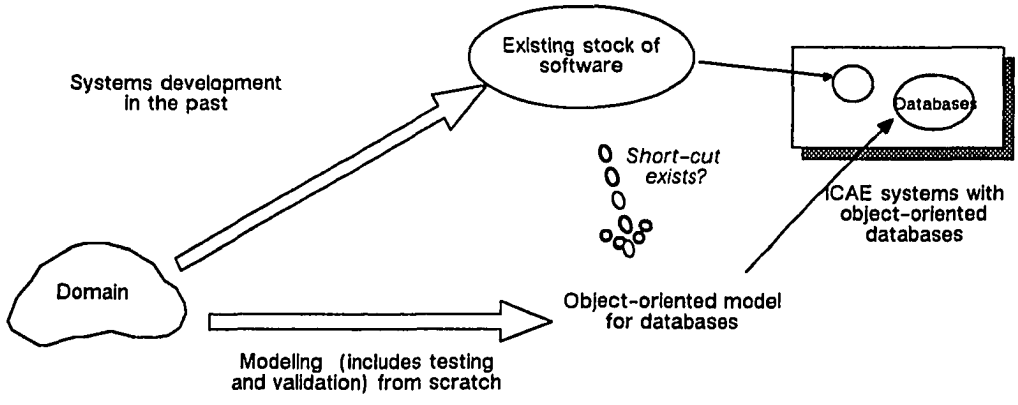


Figure 2.2 The Task of Object-oriented Modeling of Data for the Existing Stock of Software

Presently, researchers in CAE view the problem of software integration as one of tool integration and the focus is only on implementations of software. An important development during the last few years is that of the CAD Framework Initiative architecture in VLSI design [Harrison, Newton, Spickelmier and Barnes, 1990].† As shown in Figure 2.3, it consists of a tool integration environment--encompassing tool integration interfaces; services for user interfaces, versions, data representation, and data management; design and methodology management services--for the tool developers and CAD system integrators. Of particular interest in this tool integration environment is the foreign tool interface. With the help of this foreign tool interface, a tool views the input and output files in its internal formats while the data is being managed by the framework. In many cases, the foreign tool interface treats the entire input or output data as a single data record in a format that is native to the tool (also known as "stranger" data); otherwise, it translates data between the two representations in the framework and the tool. The limitation of this unit is that it facilitates integration only through the input and output of the tool, not any "deeper." For example, if the foreign tool is a batch system, then one is tied to its batch nature; this is a serious liability considering that the current generation of software are interactive.

An alternative approach has been proposed and developed by Yamashita [1986], which he calls "heterogeneous integration." The essential idea is based on an analogy to integration of heterogeneous hardware; the integration is viewed as a problem of converting data or "messages" (this is not related to the concept of messages in object-oriented programming) with the help of data converters. The

† *Framework* represents a collection of mechanisms at various levels of abstraction for software systems integrators. The role it plays in application development is analogous to that of an operating system.

idea of tool integration in VSM is to map messages between tools or program units with the help of "mapping objects" which function analogous to the alluded data converters.†

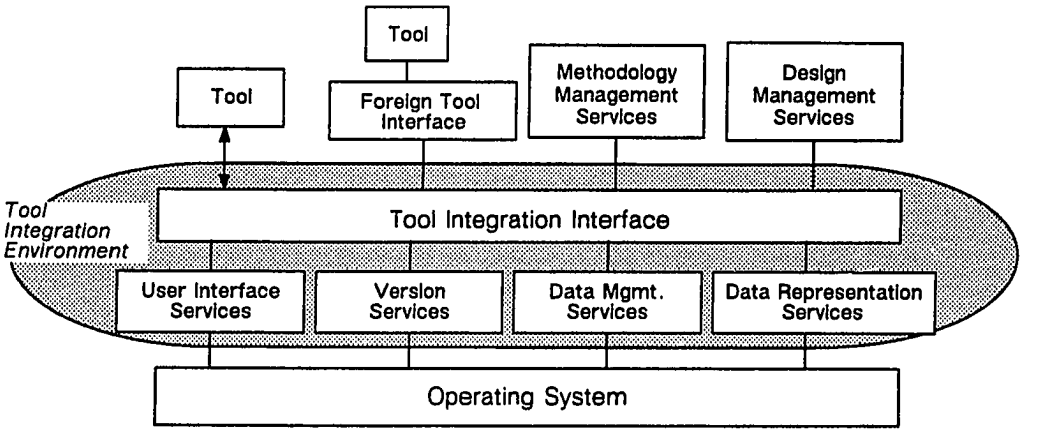


Figure 2.3 The CAD Framework Initiative Architecture for Tool Integration in Electronic CAD

Virtual Stack Machine, hereafter referred to as VSM, is a system that is based on this analogy and related ideas [Yamashita, 1987]. According to Yamashita [1986], in VSM one treats a subroutine or another kind of program unit as an object, and various data in the list of arguments as messages. This view is in stark contrast to the common knowledge that a program or routine in conventional imperative languages is analogous to a method in object-oriented languages

† This analogy between software integration and hardware integration is misleading, because it does *not* account for the true problems of software integration. In software integration, the major difficulties involve *complexities of data handling before and after* a particular program is executed, not communication during execution.

(a method is applied to the objects that correspond to the shared or parameter data). In fact, the view of subroutine in VSM as an object makes no departure from conventional programming, so one does not achieve the benefits of object-oriented programming; this view is valid only from a software development, *not* an application domain perspective. As an example, consider a system for matrix operations. The association should be between subroutines and methods, subroutine calls and messages, matrix data and matrix objects; the view in VSM, however, associates subroutines with objects and matrixes with messages to these objects (although, methods are objects at meta-level).

This difference of viewpoints is important if the goal is to develop objects for users who are domain experts. Moreover, merely mapping data and code in the foreign tool to objects and methods in VSM will *not* lead one to true object-orientation and concomitant benefits. Consider, for instance, the case wherein an existing graphics editor is integrated in VSM but the user is prevented from defining new classes of objects because certain parts, specifically the driver routines of the editor program, used in as-is condition will not accept new types of data--a legacy of old design decisions that are said to have considered all possible types of data. Is the resulting system object-oriented? It is only partly object-oriented because it fails to deliver on the expected benefits, least of all allowing the user to define new data type. It is object-oriented only in an illusory sense, because the data and code are "wrapped" into objects and methods. However, if the "hard-wired" driver routines are replaced or somehow modified dynamically, then the resulting system can be made truly object-oriented in the sense that it would accept new user-defined datatypes or subclasses.

Almost all researchers in systems integration strongly affirm, often dogmatically, the following black box principle: thou shall not "open" the program or

software. In the same vein, the principle underlying VSM is that the program or software should not be altered. This principle will be referred to as the non-modification principle. The adherence to, and stated import of, such principles by their proponents is dogmatic because many questions about their relevance and costs--such as under what conditions should they be followed, what are the tasks entailed, and others--are not even raised, let alone answered. The fact that a program unit, which is a part of the software being integrated, provides certain necessary functions in the old software does not imply that it should be integrated in as-is condition or integrated at all. If the program is rather complex, then its integration may turn into a bane for the system integrator. Even if the mechanisms for integration are simple, the total required effort of programming may be too large if the program is complex. Interestingly, the proponents of such principles are none other than the users of generally proprietary tools which the user wishes to integrate. The black box and non-modification principle need not be followed by the software manufacturer or vendor; the vendor may choose to modify its own software to ease integration.

In sum, the current approaches to object-oriented modeling of data and approaches to software integration have many limitations. As regards data modeling, any methodology that prescribes "modeling from scratch" for a large engineering domain is rather impractical and economically unattractive. As regards software integration, any approach based on strict adherence to either the black box or the non-modification principle has many disadvantages and will not provide most of the basic, quintessential, and significant benefits of the object-oriented paradigm.

2.3.2 The Proto-ICAPE Project Approach

A new approach, which is based on software reuse, to integration in ICAPE is proposed and also developed in this research. It takes one closer to both the object-oriented paradigm and the *primary goal of integration: to impose coherence or commonality on data, code, functions, interfaces, and other elements of the existing stock of software*.

The study of techniques for reusing *existing* software for new systems is the subject matter of *software reuse*,† a field in computer science that has had increasing interest since the 1980's. Further discussions of concepts and techniques are deferred until the next chapter. The two approaches discussed above, the black box approach and that of VSM, are reduced to special cases of reusing only the implementations of software.

A software package usually consists of components for different domains. Some components model the application domain (Winograd calls it the *subject domain* [1979]), some model the communications domain, some model the domain of man-machine interaction, and some model other domains. Any ICAE software involves domains from the following three areas: specific engineering discipline, general engineering, and software engineering. The primary concern in ICAPE should be the components for the domain of process engineering; those from other domains may be discarded in lieu of better and newer alternatives from other sources. For example, in the software to be integrated, one may discard all components for the domain of man-machine interaction if the user interface is built from old technologies. In short, the components that model different domains should be reused independently and differently.

† The term was first coined by M. D. McIlroy at a 1968 NATO conference.

The main thesis demonstrated in this research is the following: techniques for software reuse can be adopted for object-oriented modeling for *both* data and software integration (see Figure 2.4). The next chapter presents a systematic methodology based on this thesis. As a demonstration, a prototype ICAPE system called Icape-91 is developed. Icape-91 consists of software for the domain of process engineering from the existing stock. The discussions on its development follow that on the new methodology. Details of the prototype development are discussed in the chapter after next.

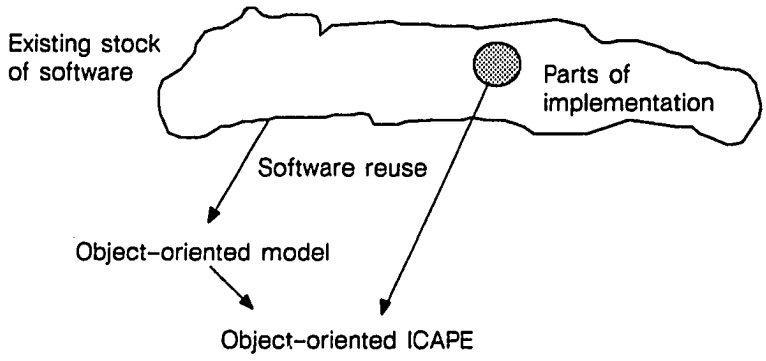


Figure 2.4 Software Reuse Approach of the Proto-ICAPE Project

2.4 Related Research

Many ICAE projects have been undertaken and are being conducted in various engineering disciplines, but only one is reported in chemical engineering literature. Only a handful, rather than all, of ICAE projects are reviewed in this section. The ones briefly reviewed and contrasted with the Proto-ICAPE Project are listed in Table 2.2.

Table 2.2 Major Research Projects in ICAE

Name	Domain	Place	Contrast with the Proto-ICAPE Project
IPAD	Aerospace Engineering and Manufacturing	Boeing Company, Seattle	Old technologies.
(by Randy Katz)	VLSI Design	University of California, Berkeley	A black box approach to integration.
Ulysses	VLSI Chip Design	Carnegie Mellon University, Pittsburgh	AI-based and a black box approach to integration.
DELI	Software Development	MCC, Austin	Different domain.
PROCEDE	Chemical Engineering	The University of Leeds	Considers the problem as that of specific software, not in its general terms. Result: a "package," not knowledge or concepts.

IPAD

The vision behind the IPAD project for aerospace engineering design and manufacturing carried out from 1971 to 1984 is one of the motivating factors behind the Proto-ICAPE Project. One of the objectives of IPAD is to share data in a global database between the conceptual design, final design, drafting, and manufacturing processes. The list of functions supported by IPAD has much in common with a typical desiderata of engineering information management in other disciplines. It includes multiple views of data, multiple levels of data descriptions, data definition modification, geometry data management, configuration management, metadata management and other functions. Being the first of its kind, its aim was in part to establish the functional requirements of an engineering data management system.

Despite its broad influence on the engineering and manufacturing industry, the IPAD project had some shortcomings. Being an initiative from users, it was

driven by needs rather than proven, state-of-the-art technology. For instance, it relied on distributed operating systems which were not available at that time and had to be custom developed. From today's standpoint, the software technologies that were used in IPAD are certainly old and overshadowed by many advances in the past decade. One can find no mention of object-oriented programming which became popular in the 1980's. Although the IPAD project advanced the implementation of relational DBMS technology for engineering by including repeating groups such as vectors and matrixes for data types, it made *no* contribution to the theoretical aspects of logical design (that is, the logical design of relational databases extended with vectors and matrixes). Even databases with multiple data models were developed along implementation, but not along theoretical lines; these efforts are now overshadowed by research in heterogeneous, federated, and other multidatabase technologies. On the whole, the IPAD project was largely development and little research.

By Randy Katz

Amongst all projects in engineering schools, one of the most comprehensive undertaking of design management systems is by Randy Katz at the University of California, Berkeley. Katz [1984] has written extensively on various information management needs of CAD systems based on his analysis of the design processes in diverse areas such as VLSI chip, piping systems, architecture, etc. Katz developed a prototypical design management system based on a semantic data model for design data, a nested-transaction model, and mechanisms to interface with CAD tools.

Katz describes various functions of a design information management system and problems that deserve attention in its development. His writings, however, provide scant description of techniques to reuse the large amount of

existing software. In contrast to Katz's research, the focus of the Proto-ICAPE Project is more on software integration and data modeling rather than design management systems.

Ulysses

Ulysses is a VLSI design environment that was developed at Carnegie Mellon University by Bushnell [1988]. Bushnell has developed techniques for automatic execution of specific CAD tools, in order to ease the management of design tasks that become difficult due to the multiplicity of tool-specific languages. The tool integration facility in Ulysses views a tool as a black box: the interface accepts output file(s) in the output language(s) and generates input file(s) in the input language(s) of the tool that is integrated.

The limitations of the black box approach to tools are discussed in Section 2.3.1. The approach is acceptable in VLSI CAD since the tools have formal languages for both input and output. This is rarely the case in other other engineering disciplines; a majority of process engineering software, including simulators, might require input in a specific language, but the outputs are merely formatted text files in no known formal language. Unlike Ulysses, the Proto-ICAPE Project is largely concerned with data modeling and tool integration; it is not concerned with design management or design methodologies since there are very few design methodologies in process engineering. Another difference between the two is that Ulysses uses production systems (popularly known as expert systems) to integrate tools, but Proto-ICAPE Project does not.

DELI

An interesting project in software engineering is in progress at MCC (Micro Electronics and Computer Technology Corporation), Austin, Texas, under the

Software Technology Program. This research proposes and develops what the author calls "post-facto integration" techniques to reuse large, heterogeneous systems with a minimum of reprogramming [Power, 1990]. The general idea of post-facto integration is to create a fixed abstract interface model for programmers and a mapping between it and the target system to be integrated.† Figure 2.5 illustrates an application of the post-facto integration to database systems. DELI presently consists of object models created by employing post-facto integration to window systems, database systems, and text file editors [Powers, 1990]. This research aims at providing automation tools and methods to support post-facto integration of software modules in different programming languages, run-time environments, and operating systems. Post-facto integration is proposed and pursued as a distinct area of research, which is highly commendable.

The goal of post-facto integration is the closest to that of the Proto-ICAPE Project. There are, however, no reports on the details of a systematic methodology for post-facto integration. The mere idea of an abstract interface model as depicted in Figure 2.5 provides little guidance in deriving the model; perhaps, model derivation is left to the programmer's art. The results therefore cannot be transferred to other domains and perhaps other systems. Additionally, there are no reports on the application of post-facto integration techniques to any traditional engineering domains.

PROCEDE

PROCEDE is a process engineering design environment (for a hardware environment consisting of desktop compuers) in development at The University of

† The notion of pre-facto and post-facto integration is confusing. As the term *pre-facto integration* is oxymoronic, so the qualifier *post-facto* is redundant; any problem of integration is after the fact.

Leeds, UK [Preece & Stephens, 1989]. The approach is to treat an application program as a black box, completely closed, and secondary to the activity of managing data in files or "databases" (the authors consider files as databases!). The authors give an example wherein an input file for a process simulator is generated from a flowsheet schematics program, after which an equipment design application is invoked, and then a PID is edited; the integrated programs are treated as black boxes. Their paper presents criticisms about the idea of employing data management systems based on poor performance of systems at that time. The conclusions of this paper lists various systems and subsystems that are included: two grades of printers, four operating systems, two process simulation programs, and two word processors.

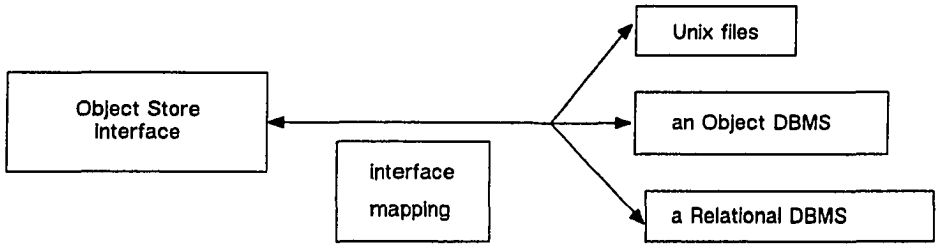


Figure 2.5 Post-facto Integration of Database Systems in the DELI Project

Although the scope is impressive, the result of this research is simply a "package deal" rather than concepts, principles, and methodologies. There are no new results that are easily transferred to other systems and applications. The criticisms of poor performance advanced by the authors on DBMS's are no longer valid, considering the advances in the last few years. In PROCEDE, data exchange techniques such as "clipboard," "dynamic data exchange," and others, are considered more useful than DBMS's for performance reasons. This argument

is weak in many ways. A DBMS is more than a means of exchanging data: it manages integrity, security, consistency, sharing, availability (in a distributed DBMS), and other functions which are of paramount importance when a database is used by many people and becomes the centerpiece of an organization. On the whole, the authors have not grasped the basic advantages, let alone the long-term benefits, of a data management system.

Others

In chemical engineering, others too have undertaken substantial application of object-oriented programming and deserve an examination. None, however, is aimed at integration, let alone software reuse, as in the Proto-ICAPE Project. Two such projects listed in Table 2.3 are briefly discussed below.

Table 2.3 Major Research Projects on Object-oriented Applications in Chemical Engineering

Name	Place	Contrast with the Proto-ICAPE Project
DESIGN-KIT	Massachusetts Institute of Technology, Cambridge	AI-based environment for process engineers
ASCEND	Carnegie Mellon University, Pittsburgh	Aims to provide equational modeling environment

DESIGN-KIT is an object-oriented environment that is designed to support modeling of declarative and procedural knowledge in process engineering [Stephanopolous, Johnstone, Kriticos, Lakshmanan, Mavrovouniotis, and Siletti, 1987]. The Proto-ICAPE Project, on the other hand, is concerned with data and procedures in the "conventional" sense; that is, no AI-based software are included. As regards object-oriented modeling, the authors seem to have followed ad hoc

and intuitive object modeling. For example, thermal characteristics of unit–operation are modeled as objects in their own right [Stephanopolous et al., 1987]. The subclasses include adiabaticity, isothermalilty, and such. However, an examination of almost any relevant software in process engineering shows no separate data structure(s) for thermal characteristics; in fact, the adiabaticity of any process is usually denoted simply by a value of real data type. (In certain systems such as SmallTalk–80, such real values are objects in their own right; but that is a different matter.) Unless one is developing a conceptual model, one need not create a class for thermal characteristics—it does stand on its own as a concept—since there is apparently *no* benefit in terms of programming effort. In general, the goals of ICAPE and the Proto–ICAPE Project are vastly different and have little in common with research in DESIGN–KIT.

ASCEND is another major application of object–oriented concepts to equational modeling in progress at the Engineering Design Research Center [Piela, Epperly, Westerberg & Westerberg, 1991] of Carnegie Mellon University, Pittsburgh. However, equational modeling and computation facilities in ASCEND are suitable only for the development of *new* process models, a goal that is rather different from that of the Proto–ICAPE Project.

2.5 Summary

Some of the problems with the present day CAPE software systems, the problems that created the need for integration in CAPE, can be classified as belonging to one of three levels: application, data, and software. In this chapter, the existing approaches to solve them are discussed, and the Proto–ICAPE Project is presented as a new alternative.

Some of the major research projects in ICAE are examined for a potential

solution or motivation for ICAPE. The previous attempts in ICAE take a black box approach to tool integration. Only the DELI project in progress at MCC attempts some software reuse by creating a common model of target systems to be integrated. The Proto-ICAPE Project is different from these projects in two important aspects. First, unlike all projects except the one at the University of Leeds, the domain of Proto-ICAPE Project is process engineering. Second, the software reuse approach to derive object-oriented models from existing software is its most important contradistinction.

3. Software Reuse Approach

The previous chapter advanced the thesis that software reuse can be adopted for object-oriented modeling for data and software integration in ICAPE. To this end, a systematic methodology called Reuse for Object-orientation (REO) is developed and is the main subject of this chapter.

This chapter consists of three sections. First, it discusses some concepts of software reuse and examines approaches to reuse from the standpoint of ICAPE requirements. Second, it describes in detail, along with examples, the REO methodology. Finally, it reviews related research by others. The next chapter discusses the development, which follows this methodology, of a prototypical ICAPE system.

3.1 Software Reuse Concepts and ICAPE

The field of *software*† *reuse* has been active since the early 1980's [Biggerstaff & Perlis, 1984], but more so now, as indicated by the recent formation of the Subcommittee on Reverse Engineering under the Technical Committee on Software Engineering of the IEEE Computer Society. By employing software reuse, one can increase the return of investment in the old software as well as reduce the cost of development of a new, equivalent system that uses different, perhaps new, concepts. There are two areas of study under software reuse: (1) reuse of *old* software in *new* and different software, and (2) software reusability. In the latter, reusability is the desired property of software (to be developed) and the goal is to find systems and languages that improve the reusability of software. (It can be argued

† The term *software* refers not only to the final product, but to the collection of all related documents--products in the form of data, programs, and descriptions--generated at various milestones throughout the product life cycle. (See the footnote on page 6.)

that reusability is not an inherent property of software, because clearly reusability depends on the techniques of reuse.) In the former, on the other hand, the primary goal is reuse of old software. Repeat: one is for developing reusable software, and the other to reuse developed software.

Software reuse of the former kind, using the old for the new, is of primary interest in ICAPE and ICAE. Henceforth, for a lack of a suitable alternative term, the term "software reuse" will be used in the sense of reusing old for new, not reusability. In the presence of a paradigm change, software reuse is a rather difficult problem because it cannot be solved through language–to–language translation. As an analogy, consider the problem of parallelization of software for parallel computers; new algorithms are still developed and considered more worthwhile over compilers for parallelization. In any case, given the expected benefits of an object–oriented paradigm, it is worth investing in developing techniques that can assist in software reuse.

An aside: One of the main impediments to progress in this area is from the legal standpoint of potential infringement on copyrights and patents. Nonetheless, reverse engineering one's own software is legal. Thus, the field of software reuse is of immense technological interest to various organizations that would like to undertake reuse of the software they own. As regards the Proto–ICAPE Project, the experimental subject is in the public domain. In the area of computer software, in general, there is inconsistency in many legal cases, and no definite legal test has been developed so far [Kinne & Kappes, 1992].

There are two types of reuse techniques, virgin and processed, depending on whether or not the software component being reused is subjected to any of the processes of software development. (The term software component refers to a product, which is in the form of a document, that is generated by one or more

software processes that are part of software development life cycle. A typical life cycle of software product consists of the following stages: requirements, specifications, design, implementation, testing and maintenance. Hereafter, a software component is often referred to simply as the *component*. Do not confuse this use of the word *component* with its use to mean a chemical component in subsequent chapters.) In *virgin reuse* techniques, a software component is used in as–is condition, in its virgin or as–is form, requiring only declarations of references and access paths. One example of a virgin reuse technique with which even amateur programmers are familiar is that of a subprogram call in which routines from a library (of compiled routines) are referenced. In this example, software components from the implementation and perhaps tested phase are reused without subjecting them to one or more software development processes. There are many techniques for virgin reuse of software components from the implementation stage. Virgin reuse techniques are of no interest in this research.

In *processed reuse* techniques, a software component is not used in its original form, but after subjecting it to one or more software development processes; in other words, the software component's use requires programming beyond mere declaration of references and access paths. There are two aspects of processed reuse techniques: the nature of processing and the kind of software component (the kind of software component refers to the stage of the software development life cycle). For example, the processing of components might consist of analysis to derive a model based on different concepts. The components of interest are modules and programs (that are implemented and tested), requirements, specifications, design, and even various manuals. These components are generated in the form of data files, programs, and manuals during various processes in the software development life cycle.

Software reuse should *not* be mandatory, since techniques may not exist or the subject may be too complex. In the existing literature, there are no definite guidelines by which one can decide when or how to reuse software. Thus, more research is needed.

Software reuse may be approached in more than one way. In the new paradigm, one may create software with equivalent, extended, or different functions from the old; the old software system will be referred to as the *subject software system*, or simply as the *subject* of reuse processes and methods. Some of the software reuse approaches are defined by Bachman [1990] as follows: (1) *redevelopment* means to re-create the system (component) requirements and develop the system (component) in terms of new concepts; (2) *reengineering* means to re-create the system (component) specifications in terms of new concepts from information in the existing system (component) implementation, also known as *reverse engineering*, and then design, implement and test the equivalent new system (component), also known as *forward engineering*. Between the two phases of reverse engineering and forward engineering, one may enhance the specifications to meet new requirements. Another set of definitions is given as follows [Chikofsky & Cross, 1990]: "(1) *reverse engineering* is the process of analyzing a subject system to identify its components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction; (2) *reengineering* is the examination and alteration of a subject system to reconstitute it in a new form and subsequently implement it in the new form—it generally includes some reverse engineering followed by some forward engineering or restructuring; (3) *restructuring* is the transformation from one form of representation to another at the same relative abstraction level, while preserving the subject system's external behavior—its functionality and semantics." This set of definitions is not as complete as that

given by Bachman; for instance, in the above definitions it is not clear what is meant by "reconstitute it in a new form." Hereafter, the terms will be used in the sense defined by Bachman.

Different approaches to software reuse are shown in Figure 3.1. In the redevelopment approach (S1 in Figure 3.1), an equivalent software system for the new paradigm is developed afresh from requirements that are obtained by direct or processed reuse from the old. This approach is no more economical than developing software from scratch, because the major costs of software development are incurred after the requirements stage. It is not a good choice for ICAPE because the amount of software that one has to cover is simply enormous.

In the reengineering approach (S2 in Figure 3.1), the extent of reuse is greater than in the redevelopment approach. However, forward engineering--consisting of all processes in software development after specifications: design, implementation, coding, testing--is necessary, and the work required and costs incurred are comparable to development that starts afresh from requirements (as in S1 in Figure 3.1). If software tools for reverse engineering are available or can be developed, then reengineering is more economical than redevelopment. This approach, for ICAPE, requires one to forward engineer an enormous amount of software; additionally, the task of testing systems made of millions of lines of code (the typical size of any useful ICAPE system) is rather costly.

In another approach, reengineering plus virgin reuse (S3 in Figure 3.1), components from the old implementation are reused in virgin or as-is condition in addition to reverse engineering to generate requirements and specifications. This approach saves substantial costs of implementation, and more importantly, testing of software. Some testing may be required for validating integration, but a lot of

domain-specific testing is eliminated. This approach fits well with the technological need of ICAPE, namely, reusing existing implementations.

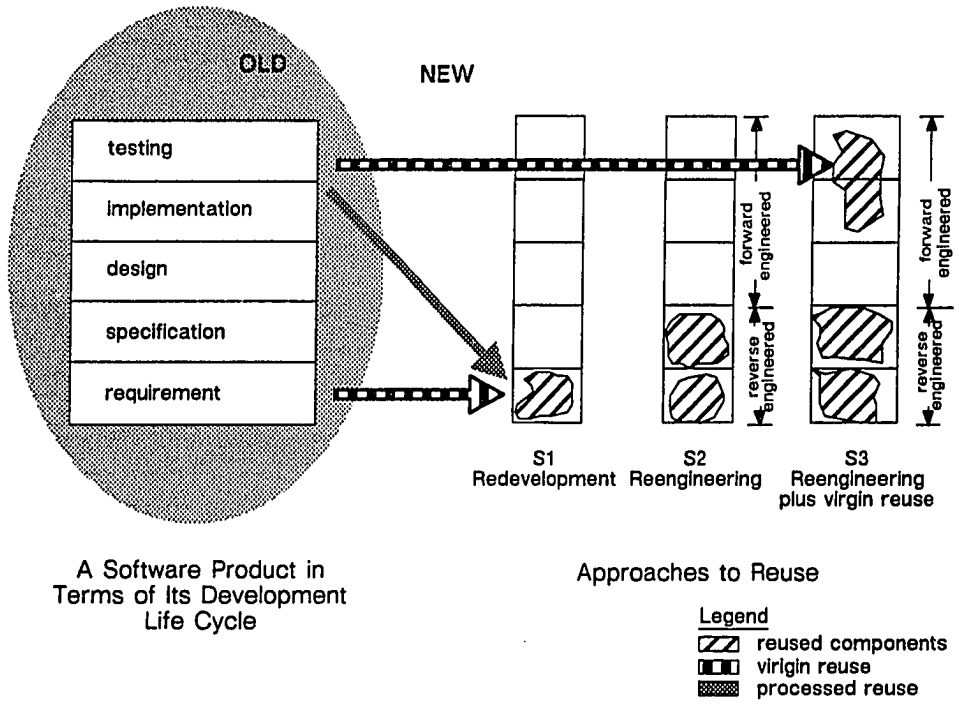


Figure 3.1 Some Approaches to Software Reuse

Summarily, there are three approaches to software reuse to migrate the existing software base to different concepts or a new paradigm in decreasing order of cost: reengineering, redevelopment, and reengineering plus virgin reuse. For object-oriented modeling for integration in CAPE, reengineering with virgin reuse is clearly the most economical approach.

3.2 REO Methodology

The derivation of object-oriented models for subject software systems (which are not object-oriented) can be aided by following a systematic methodology.† As shown in Figure 3.2, starting with the subject and knowledge about its application domain, one generates an object-oriented model that provides a basis for object-oriented system(s). The part or whole of the subject software system that is reused is said to be *covered* by REO. Often there are no formal requirements, specifications, or design for the software system; instead, one is left with informal descriptions in various manuals. The domain knowledge input is for validation or elaboration of semantics; however, its role is rather informal. REO is not a method of design that would lead one to a definite, particular solution; in other words, a lot depends on the decisions taken by the programmer or designer. As shown in Figure 3.2, REO consists of two distinct processes: reverse engineering and virgin reuse of the subject's components. For a particular component, both processes may be required. The components that should be covered include those that constitute the "universe of discourse" of software engineering such as requirements, specifications, program unit descriptions (of either the source or object level), and language descriptions.

An aside: Automation tools can be developed for some parts of REO methodology to improve the productivity of programmers and the quality of ICAE projects. This requires formalization of REO methodology that is beyond the scope of this research. Also note that the majority of potential subjects of REO, software for specific engineering disciplines, are not result of formal processes or of formal nature. Similarly, domain knowledge is not formalized for most areas; none is

† A systematic methodology means a system of concepts, principles, and techniques to solve problems.

reported for process engineering, and one is unlikely to be developed in the near future. Thus, total automation does not seem feasible at this time.

An object-oriented model is said to be fully defined if the public and private sections of each class (the unit of modularity in object-oriented programs) are fully defined. For example, if some of the class or instance methods are not defined or specified (for they may require objects that are unknown at the time), then such a model is not complete. The object-oriented model generated by reverse engineering has incompletely defined operations; but with virgin reuse, one can move closer to a complete model. The resulting model can be used in the development of database models, AI systems, ICAPE systems and other software.

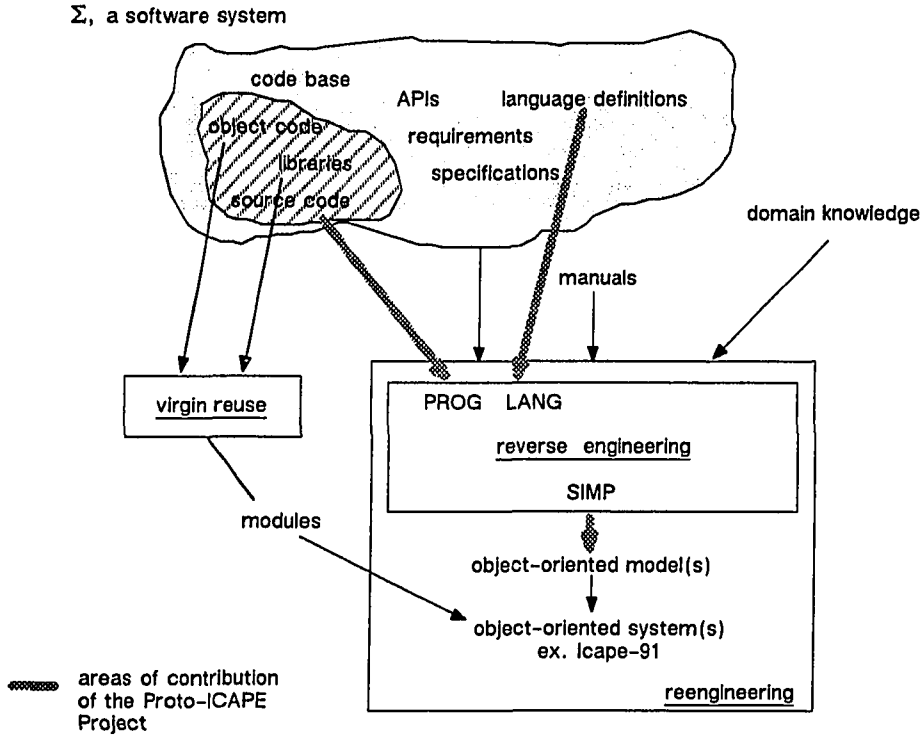


Figure 3.2 REO Methodology

The virgin reuse of components from the implementation stage, also known as code, is through modules. A *module* refers to a program unit that can be separately edited, compiled, debugged, and in effect replaced with another in the executable system; usually modules are statically, not dynamically, linked and loaded to create an executable program. The subject may already have a set of modules; otherwise, new modules are developed to interface with either the subject, or its subsystems, or both. The virgin reuse of other kinds of components, such as langauge definitions, requirements, and specifications, might be useful but is not required in the Proto-ICAPE Project.

The reverse engineering of software involves three steps: (1) selection, (2) derivation, and (3) refinement. The first step is the selection of components that can or should be reused; for example, the input and output language descriptions can be reused and so may be selected for reuse. The second step is the derivation of object-oriented models from the selected components. The third step is the simplification of the derived models. The following three sets of methods, each discussed in detail in separate sections later, are used for derivation and simplification steps:

1. LANG for reusing programming language descriptions. If it is required that the subject be unaltered and interfaced only with its input and output, then the LANG method is applied (presently, this set consists of only one method that is also named LANG) to derive object-oriented models for the input and output of the program. One can derive most of the structural, and some of behavioral parts of the object-oriented model.

2. PROG for reusing program unit descriptions. If reusing individual program units of the subject is permitted (there may be constraints imposed that prevent reuse of a program unit) or required, then the PROG methods are applied to

derive object-oriented models of the program units; presently, only program units implemented in FORTRAN are covered, because of the limited scope of this project.

3. SIMP for refinement by simplifications. The object-oriented models derived by application of the LANG and PROG methods have redundancies (in the structure and code) from the perspective of object-oriented programming. Thus, the SIMP methods are applied to simplify the derived model.

3.2.1 Model Derivation From Programming Language Descriptions

First, a short background and some terminology. The syntax of programming language is described in a notation called "context-free grammar"† or Backus Naur Formalism. The productions of such a grammar can be classified as terminal or non-terminal. The *terminal production* has only terminal symbols on the right side. The *non-terminal production* has at least one non-terminal symbol on the right side. A production consisting of only one non-terminal or terminal symbol on the right side will be referred to as *unary production*. Accordingly, a production with exactly one terminal symbol on the right side (for example, $X ::=$ Y) will be referred to as a *terminal unary production*. The derivation methods would be simpler for a single production rather than a group of productions, so it is recommended to write every group of productions as individual productions. For instance, the production, "$X ::= Y(A \mid B)$," should be written as three

† A grammar consists of a set of terminal symbols, a set of non-terminals, a set of productions, and a start symbol. A production consists of a non-terminal, a "->" or "::=" sign, and a sequence of terminal symbols and non-terminals. The part preceding the arrow sign is known as the left side of the production, and that which follows the arrow sign is known as the right side of the production. In the Extended Backus Naur Formalism notation, curly brackets are used to denote repetition of non-terminals, and a vertical bar ("|") to group productions with the same non-terminal on the left side.

productions, "$X ::= Y$," "$X ::= YA$," and "$X ::= YB$." The production, "$X ::= A \mid B$," should be rewritten as three productions, "$X ::= \epsilon$," (a production with an empty right side) "$X ::= A$," and "$X ::= B$."

The LANG method consists of developing an object-oriented model for the grammar (of the input or output language of the program) based on the association shown in Figure 3.3 between a production and a class or a class instantiation. A terminal unary production is associated with an instantiation of a class which in turn is associated with the non-terminal on the left side. A non-terminal production is associated with a class, a non-terminal on the right side with an attribute of that class, and a repetition of non-terminals with an array of attributes of that class. As regards naming classes and attributes, the programmer or designer may follow any naming scheme, but "domain-friendly" names are preferred. One simple rule is to adopt the syntactic variable name for the non-terminal on the left side of the production for naming the associated class; additionally, to distinguish different productions that have the same non-terminal, the variable name may be suffixed with a number. Another simple rule is to use a cue from the right side of the production; for example, if the right side consists of two non-terminals, the class name may be prefixed or suffixed with the word *binary*. The terminal symbols that are keywords, operators, constants, identifiers, literal strings, and punctuation symbols of the language are implicitly part of the class associated with the production; thus, no separate slots (in the class) are required for them. These terminal symbols in non-terminal productions are omitted from modeling consideration. Based on this association between the concepts of grammar productions and class, an object-oriented model can be derived. The definitions of operations or methods on objects can be developed from the semantics of the language, semantics which are usually given by informal descriptions and examples. The classes in the

resulting model are similar to pre-fabricated parts used to build a parse tree; thus one can include operations for generating and analyzing sentences to integrate the program through its input and output.

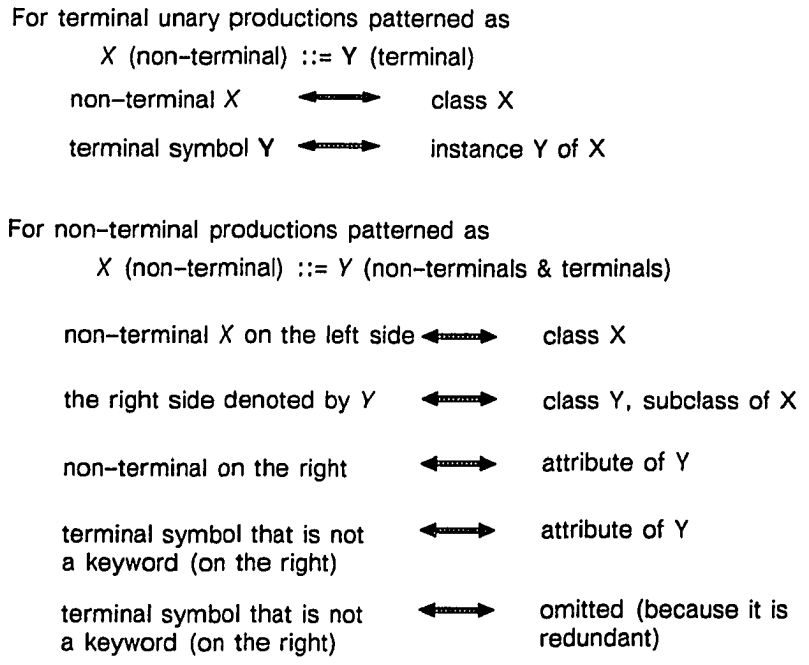


Figure 3.3 Association between Production (of Grammar) and Class

As an example application of the above method, consider a "toy" language called ARITH for arithmetic expressions. (This example is based on Example 4.2 in *Compilers, Principles, Techniques, and Tools* by Aho, Sethi and Ullman [1986]). Figure 3.4 shows the derivation of an object-oriented model from the production rules of a grammar for ARITH. The terminal unary productions with *op* on the left side and the arithmetic operators on the right side, denoted by terminal symbols, are associated with instantiations of class *operator*; for example, the terminal

unary production with + on the right side is associated with an instantiation of the class *operator*. The non-terminal productions are associated with classes related to *expression*; for example, the first production with *expr op expr* as the right side is associated with the class *binary*.

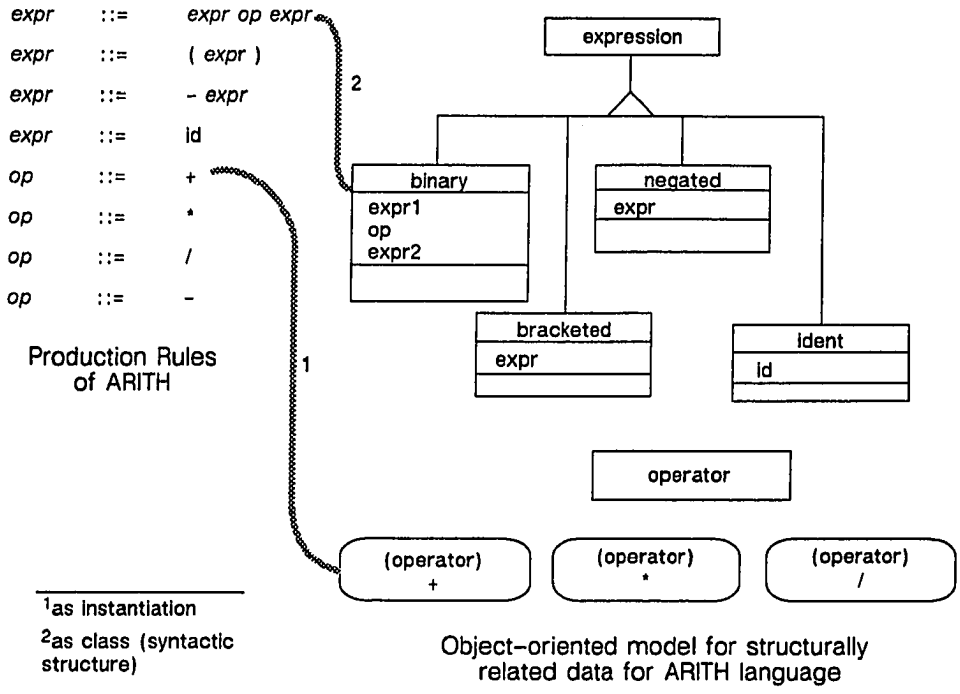


Figure 3.4 An Example of Association between Productions and Classes

The LANG method can be easily adapted for processing descriptions of form-based languages. A (textual) form consists of hierarchically structured named fields and values. The field names can be associated with non-terminals, and the values with terminal symbols.

The resulting object-oriented model is related to syntactic categories of the language and is useful for integrating the program through its input and output.

Such a model is of little interest to domain experts who are interested in objects and operations of arithmetic, not categories of syntactic structures. Nonetheless, the resulting model can be pruned to meet such domain-specific, non-linguistic requirements by simply omitting the unnecessary classes.

3.2.2 Model Derivation From Program Unit Descriptions

The processed reuse of a set of inter-dependent program units of a software system proceeds through three steps. First, one identifies the candidate program units for reuse. Second, one selects the candidate program units for reuse and the method of reuse. Third, one applies the chosen method of reuse to the selected program unit. The discussions below first cover a few simple, preliminary steps and then the specific reuse methods.

The first step, identifying the candidates, requires a program structure diagram for the subject. (The program structure diagram represents the call dependencies between program units of any software system.) The *candidate program unit* is the program unit that is reached during the execution of the program from a chosen starting node of a program structure diagram. Identifying candidate program units starting from a chosen node(s) is easily done by examining the references to other program units in the relevant object codes. The selection step requires a high-level function definition consisting of two parts, preferably stated in single words: the name of the action performed, and the names of data modified or referenced. An object-oriented model should *not* distinguish between input and output (this distinction stems from a functional view of the world), since particular inputs or outputs are merely messages to and from object(s) in certain state. The function definition may be proposed based on the informal description in domain-specific terms in the manuals or source code. Based on this function

definition, a program unit may be classified as relevant, bridge, or irrelevant. The *relevant program unit* is the program unit that modifies or references the objects in the model under development (the model under development is like a "running total"). The *bridge program unit* is the program unit needed to transfer the control of execution from one relevant program unit to another by virtue of its being on a path connecting the two. The *irrelevant program unit* is that which is neither a relevant nor a bridge program unit. The relevant and bridge program units are selected for reuse, and the irrelevant program units are discarded.

Ideally, formal documents are generated at various software development life cycle stages such as requirements, specifications, and design. Processed reuse techniques are developed for them. However, the practice of software development is rather different from this ideal; *one often finds software with no formal requirements, specifications or design*, as in this Proto-ICAPE Project. Thus, techniques of processed reuse are developed only for the documents found in practice. The first step is to associate a (selected) program unit with one or more of the existing classes from the object-oriented model, failing which one creates a new class and derives its structure and dynamics specifications. Next, one associates the program unit with methods, attributes, and message expressions. Realize, however, that these associations are secondary to those with class(es). In the final result, different parts or abstractions of the program unit are represented in the object-oriented model.

The processed reuse of program units can be undertaken in REO by applying one of the following methods:

1. CODE method, in which the code itself is directly reused; thus, it requires less effort for development. The overall cost of a project is reduced, but the code complexity may make it difficult to apply this method.

2. SORC method, in which the (object) code itself is not reused; instead, the segments of a source program are separately reverse engineered for the object–oriented model.

3. DOCU method, in which neither the source nor (object) code is reused. Instead, the natural language descriptions of the processing in the program unit are *rewritten* in terms of the objects from the model and the messages to these objects.

The selection of one of these three reuse methods is based either on technical or non–technical considerations or both. The procedure used to select a method is shown in Figure 3.5. The non–technical considerations, such as economics and management, are complex enough to warrant a separate study. One may also impose revocable constraints from project management considerations such as early prototyping; for example, reverse engineering of a source form, which takes more effort than direct reuse, may not be immediately required in which case the CODE method is applied. The selection may also be based on prior experience in reusing different program units. Some of the technical considerations are as follows:

1. A bridge program unit is reused by the DOCU method because its internal program and data structures are of no interest.

2. A relevant program unit may not be reusable in its compiled form through the CODE method due to various constraints imposed by the designer; for example, there may be constraints that prohibit including particular data.

3. A relevant program unit may be rather complex in its source form when applying the SORC method. (Complexity of code can be evaluated by some software metrics, or subjectively by a visual scan by the programmer.) In that case,

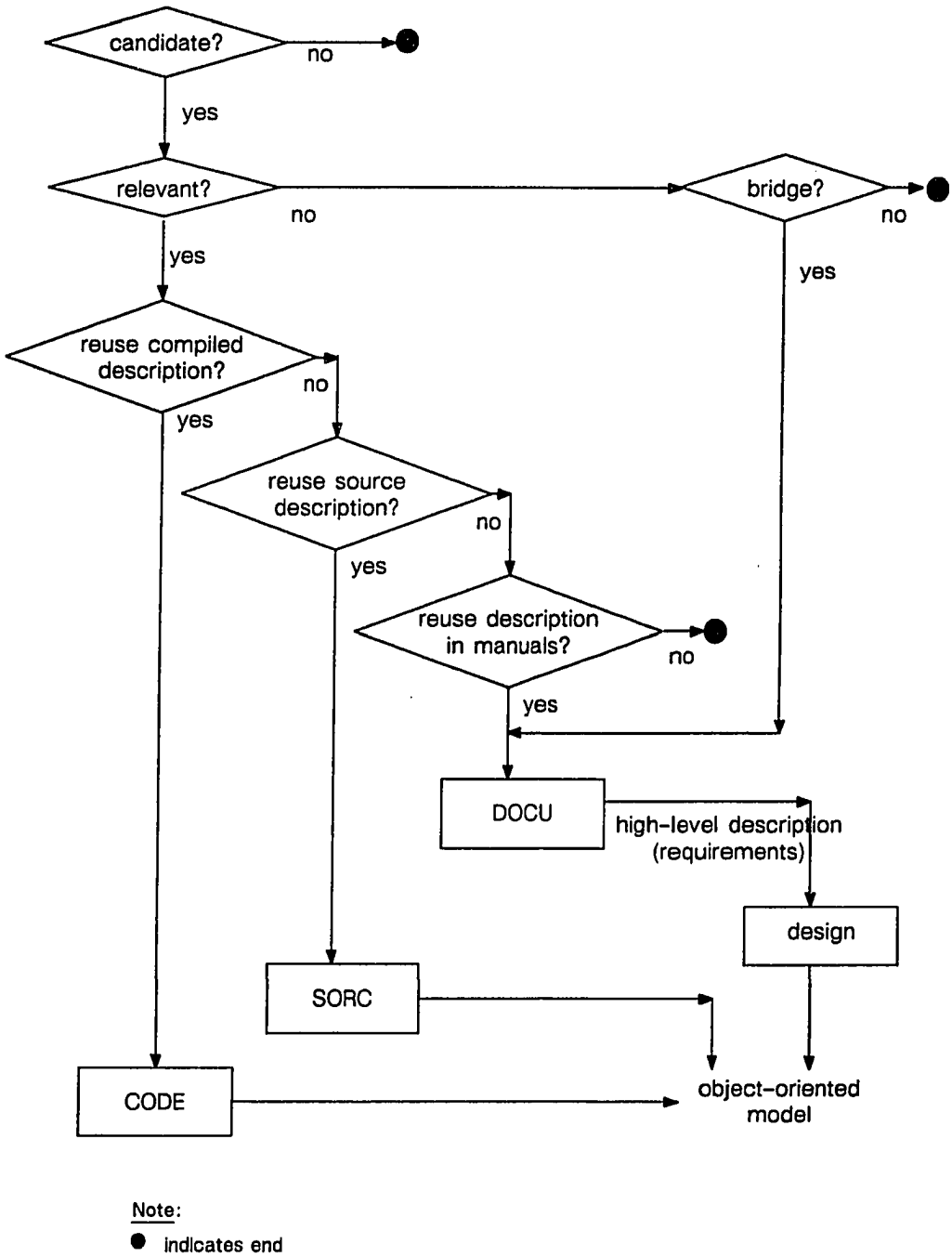


Figure 3.5 Procedure for Selecting a PROG Method

one can only work with the higher–level descriptions or specifications for which the DOCU method is applicable.

CODE

The CODE method of processed reuse of a program unit is based on the similarities between the concept of program in traditional imperative languages and the concept of method and class in object–oriented languages. A program is a means to update data, and a method (for a class or an instance of a class) is a means to update the state of object(s). A program is a construct, the only construct, to model a concept or task in the problem domain; and a class is an object–oriented construct for the same purpose.

The association between subroutine and class is shown in Figure 3.6. A subroutine is associated with a class and a method; a class may be associated with more than one subroutine because a class is a collection of "related" methods. The invocation of a subroutine is associated with the invocation of the corresponding method. The input, output, or shared variables are associable with either instance variables, or class variables, or global variables. A class is said to be *associable* with a subroutine if every object that represents the parameters and shared variables of the subroutine can be accessed from an instance of the class; or, in other words, if all the necessary data for the subroutine are reachable from the instance of the class. If none of the existing classes is associable, then new classes are created as follows: one class for the set of input variables, one for the set of output variables, and one for each shared variable.

The data specification statements (neglect the EQUIVALENCE statements† and such repeating specifications in other program units) are adapted for

† EQUIVALENCE statement is a data specification construct in FORTRAN.

the class such that each variable in the list of variables is associated with an attribute of the class. The organization of data, the data specified through different statements, may be related. Such relationships are not explicitly stated because of a lack of suitable constructs in FORTRAN or other programming languages. However, the relationship information is implicit in the source code since an algorithm follows structure; also informal descriptions are usually found in the manuals. For example, data in two different COMMON blocks in FORTRAN subroutines may be ordered by the same key values, but the relationship cannot be explicitly stated in FORTRAN; however, the relationship can be inferred from the pattern of access in one or more source programs. The data type declarations of variables are adapted in the object-oriented model; that is, they are rewritten in terms of classes that correspond to the data types.

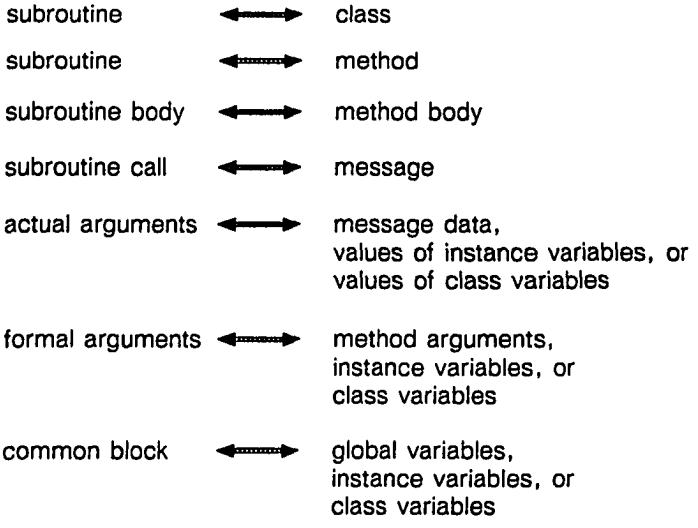


Figure 3.6 Association between Subroutine and Class

The dynamics of class include methods from four categories of events involving the program unit (the CODE method is used in conjunction with the virgin reuse method through a module interface):

1. the assignment of data or objects to attributes that are associated with shared variables in the program unit;

2. the assignment of data to attributes that are associated with the parameters of the program unit;

3. the linking of objects with the symbols in the code† in a form that is native to the program unit (for example, contiguously laid data for COMMON blocks); and

4. the calling of the module that interfaces with the code, and thereby updating an instance of the associated class.

In addition, one should specify constraints on the order of executing these methods. For example, the methods from the above categories 2 and 3 (preparation of data or objects, and linking the compiled form of the program) are executed before those from category 4 (calling the program); similarly, the execution of methods from categories 1 and 2 (preparation of data or objects) should precede the execution of method from category 3 (linking the program).

As an example, consider a subroutine shown in Figure 3.7 for computing the viscosity of a pure liquid at low temperatures using a modified Andrade correlation. A class named *andrade* and a method named *update* are associated with the subroutine MUL001. The class structure specification consists of attributes for input and output parameters, as well as the COMMON blocks. The aggregate of

† The method for linking data objects with symbols for COMMON block or shared variables depends on the linking facility in the target system. Note that static linking places many limits on interactive computing. Nowadays, dynamic linking and loading is commonly found in many experimental operating systems.

subroutine parameters T, IDX, NCP, etc., is modeled by the class *mixture* for input parameters and *mixture_property* for the output parameters. The definition of this class is easily derived from the type and size specifications of data that are readily available in the source code or program manuals. A module named *andrade_update* is created to interface with the subroutine code. The following methods are created to model events involving MUL001:

1. *attach*, to attach an instance of class *global* to the attribute *global* for the COMMON block GLOBAL;

2. similarly, *attach*, to attach an instance of class *ncomp* to the attribute *ncomp* for the COMMON block NCOMP;

3. *assignCOMMON*, to assign data from objects held at attributes *global* and *ncomp* to symbols for COMMON blocks GLOBAL and NCOMP, respectively;

4. *mixture*, to assign data to attributes associated with the input parameters to the subroutine; and

5. *update*, in which the module *andrade_update* that interfaces with MUL001 is called to update the instance of *andrade*.

In addition, constraints on the order of executing these methods are specified. Clearly, an invocation *update* must be preceded by at least one invocation of *assignCOMMON* and *mixture*. Note that the two events, call to the subroutine MUL001 and update of its (input, of course) parameters, are decoupled into two methods *update* and *mixture*. The execution of methods named *attach* must eventually be followed by that of *assignCOMMON* (since there are new data for the COMMON blocks).

SORC

The SORC method for processed reuse is based on the association between the constructs (for data and control structuring) of traditional and object-oriented

programming languages. For example, the case statement of traditional programming languages is used for type-dependent sections of a program. In object-oriented languages, such a program unit can be represented by an object of the type *collection*; this object collects the ones that represent different case blocks. The advantage in object-oriented language is that this collection can change during run-time.†

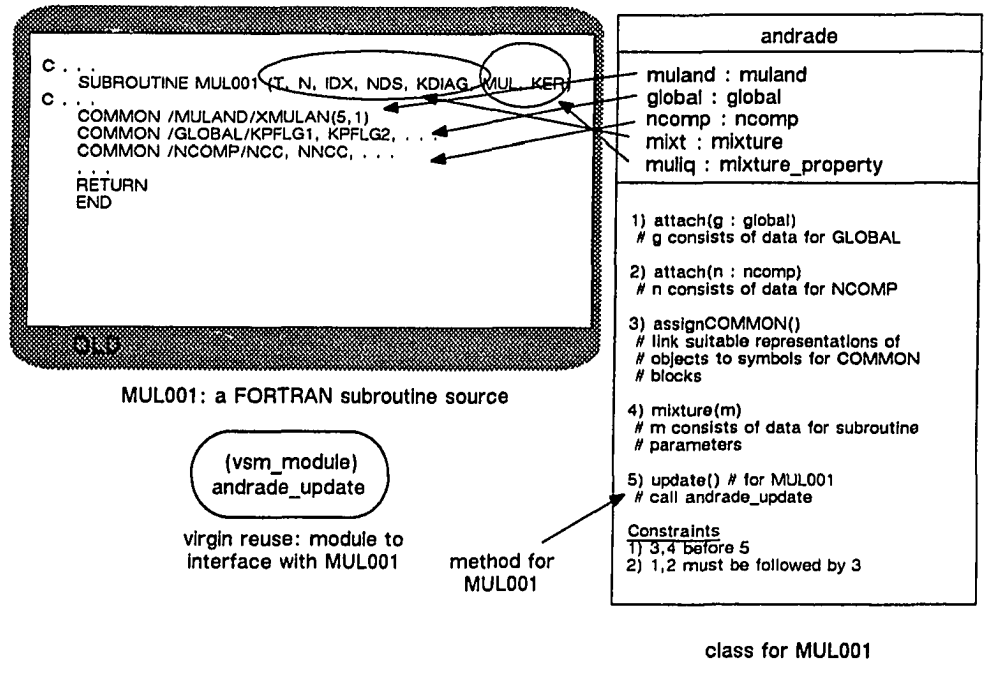


Figure 3.7 An Example Application of the CODE Method

† More precisely, in languages without dynamic binding, the control flow constructs such as IF-THEN-ELSE and GO TO statements are the only constructs to handle type-dependent code. Thus, one has to edit, compile, and relink the program unit (which is part of a statically linked program) whenever a new data type is defined. This is a classic example of the shortcoming that is overcome through object-oriented languages.

This association between constructs of traditional and object-oriented languages is shown in Figure 3.8; based on it one derives the structure of the associable class(es) from the source form of a subroutine (the distinctions between a main program, subroutine, and function subprogram are irrelevant). A case statement is associated with a collection of instances of classes for the case blocks, also known as "the arms of a case statement." The case labels (GOTO labels in FORTRAN) are associated with key values that index the collection. An assignment statement is associated with a method to update an object (or an assignment statement, if provided by the target object-oriented language, with the message expression on the right side) that represents the variables on the left side of the assignment. Similarly, a subroutine call is associated with an update message to an object that represents the updated data in the subroutine call.

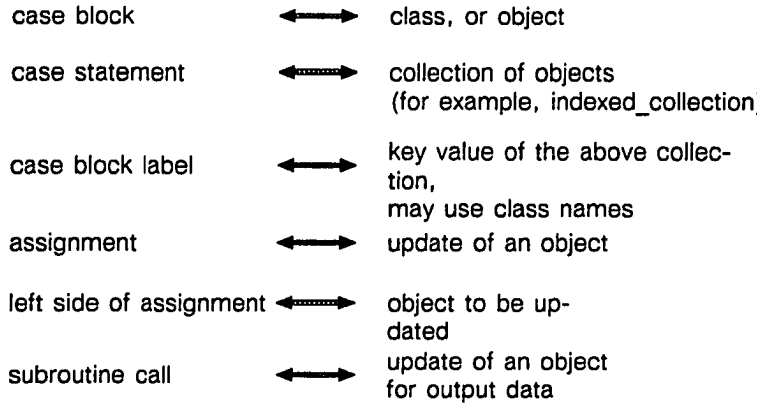


Figure 3.8 Association between the Constructs in Traditional and Object-oriented Programming Languages

Deriving classes from a program source first requires decomposing the source into segments that are readily associable. For example, a case statement

(includes case blocks) by itself can be considered as one segment, and each case block as another segment. There are *no* definite guidelines for partitioning a source program, but prior experience may help. During partitioning, a stage will eventually be reached when the segment cannot be partitioned any further in terms of the known associations. Such a segment will be referred to as "elementary," and it can only be simulated by manually rewriting it in the target object-oriented language.

As an example, consider a subroutine shown in Figure 3.9, in which the control of the program execution flows into different case blocks of the case statements. If a new case block is to be added to the subroutine SAMPLE, one is required to edit, test, compile, link and load it into a new executable program; if the executable program is large, then these steps are cumbersome, expensive, and sometimes simply unacceptable.† The same set of problems will not arise in its object-oriented equivalent. Each of the case blocks can be encapsulated into an *update* method of classes such as *classA* and *classB*. A new case block can be added any time by defining a new class in the system. The object-oriented equivalent of the subroutine, a set of message expressions, is immune to changes such as adding new classes for new case blocks, or modifying a class for some case block.

DOCU

If it is required that neither the object code nor the source be reused, then one may reuse a corresponding component from stages that precede the coding stage in the life cycle of the subject. Generally, for each program unit, some

† The seriousness of the problem is illustrated by Booch [1991] through a hypothetical situation in which the deployment of a military vessel is delayed by a day due to extensive compilation that is required after a slight modification of the software.

natural language descriptions of the function, data involved, and algorithm used are given in manuals. The DOCU method of processed reuse consists of *rewriting* the natural language description *in terms of* objects that represent the data, functions, and processes. Thus, one obtains a natural language description of the requirements or specifications of the model that needs to be developed for the program unit.

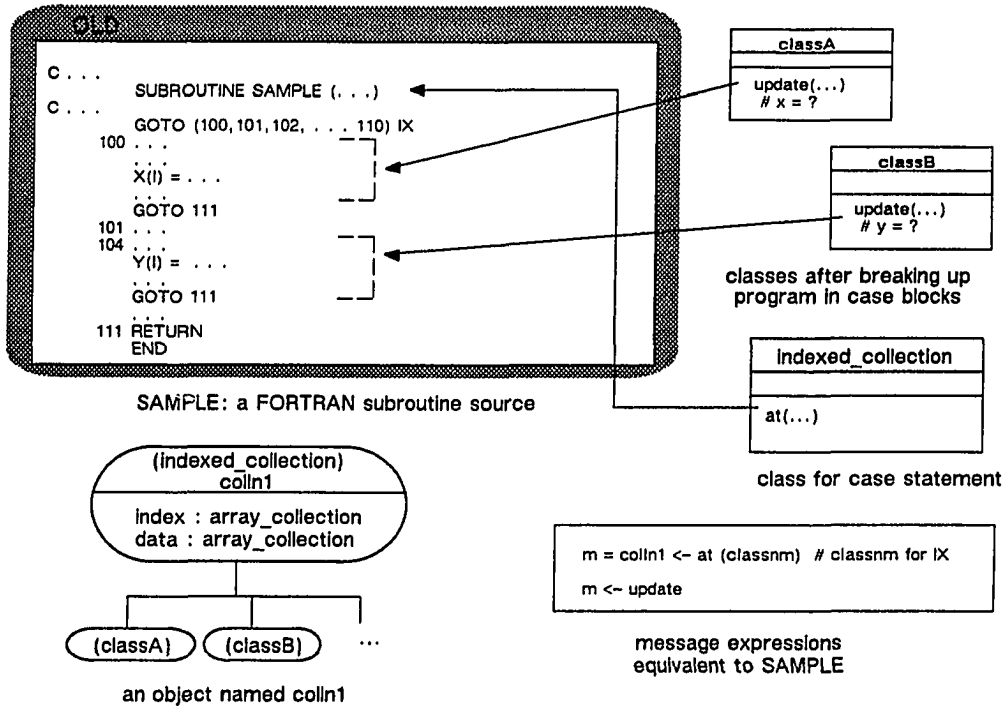


Figure 3.9 An Example Application of the SORC Method

3.2.3 Model Refinement by Simplification

An object-oriented model derived by processed reuse of a program unit usually has many redundancies, especially from the perspective of object-oriented

programming. These redundancies are derived from those built into the subject software system. The redundancies are handled separately because it is easier to do so; one first derives the model, then minimizes the redundancies (removing redundancies is analogous to normalization in the logical design of relational databases). The simplification methods used in the Proto-ICAPE Project are described below; their underlying rational can be seen by anybody familiar with object-oriented programming:

SIMP-1: An attribute which serves to identify uniquely an instance of each class is redundant in the object-oriented model and should be dropped. In object-oriented systems, an object or an instance of a class is assigned a unique identifier by the system itself; it is not the programmer's responsibility to do so.

SIMP-2: A class with only one attribute would only require methods for reading and writing a value or an object that is bound to the attribute. Such a class can be removed from the model, unless it is part of a class hierarchy which is retained for other reasons. The removal of any class requires modifying references to the class. All attributes that are of the removed class type are set to the same type as that of the "orphaned" attribute (see Figure 3.10).

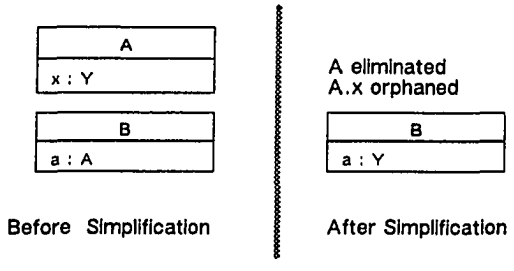


Figure 3.10 Application of the SIMP-2 Method to Eliminate Classes with Only One Attribute

SIMP–3: A class with no attributes is dropped. Its named instances in the model can be simulated by integral constants or enumerated data types in the target system.

SIMP–4: A class can be assigned a *structural signature*† consisting of a number and types of attributes. The order of attributes is as irrelevant in the class structure in object–oriented model as in the table structure of the relational data model. A class structure can be defined by a structural signature and a private name space; the names identify the attributes. The object–oriented model may consist of many classes with the same structural signature but different names for its attributes; such classes are said to be *structurally equivalent*. A class also has a behavioral part that consists of methods and the sequencing of their invocations. The classes that are structurally and behaviorally equivalent are said to be *equivalent*. All but one of the equivalent classes are dropped from the model (see Figure 3.11). Consequently, the instances of the eliminated classes are replaced with equivalent instances of the replacement class, and the attributes that hold instances of eliminated classes are set to hold instances of the replacement class.

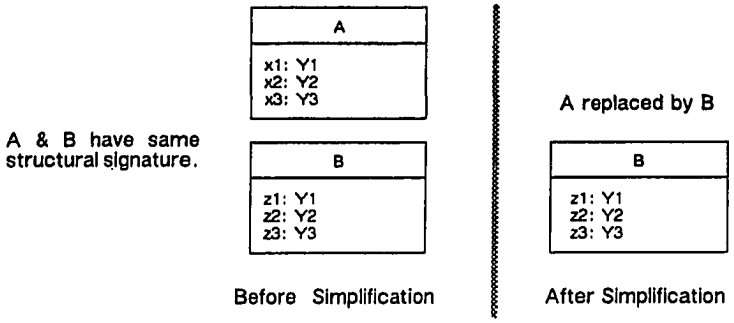


Figure 3.11 Application of the SIMP–4 Method to Eliminate Equivalent Classes

† In the literature on programming languages, the term *signature* generally refers to the number, order, and type of arguments of a function or procedure.

SIMP-5: The class that has only redundant information, which is already in other classes, is *extraneous*. Clearly, extraneous classes should be dropped from the model. Figure 3.12 shows a pattern, which was faced in this research, wherein a class consists of an instance of another class and some other data which is already in the contained object; the class that contains is extraneous. Why include such extraneous classes in the model? Clearly, they should be eliminated. There may be other patterns of redundancy, but this was the only one that was faced in this research.

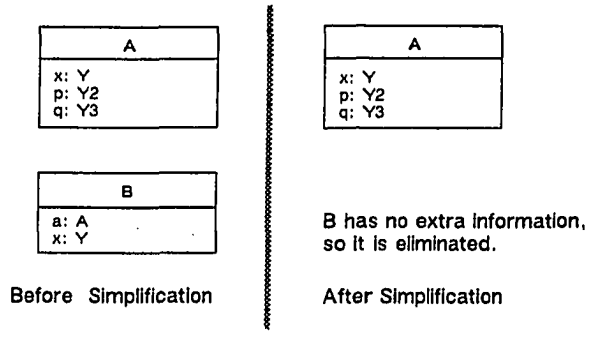


Figure 3.12 Application of the SIMP-5 Method to Eliminate Extraneous Class

3.3 Related Research by Others

Research activity in software reuse has been going on since the early 1980's, as mentioned in Section 3.1. The importance of this area is clear from the conclusion of an analysis that of "all" (sic) code produced in 1983, probably less than 15 percent was unique, novel, and specific to applications; the remaining 85 percent was common, generic, and could have been reused in applications other than that for which it was developed [Jones, 1984]. Of the two areas, software reusability

and reuse of existing software, the former is more researched than the latter [Anderson, Beck, and Buonanno 1988; Jones, 1984; Biggerstaff and Perlis, 1984; Durek and van Horne, 1988; Pyster and Barnes, 1988].

There are only a few attempts at software reuse reported in the literature. Notwithstanding the legal impediments to software reuse such as potential copyright encroachment, the commercial world has already shown much interest in software reuse during the past few years and has been actively developing software reuse technology [Bachman, 1988; Joyce, 1988]. The tools that are presently available are aimed at solving problems such as source-to-source translation, cross-referencing, quality evaluation by certain metrics; these tools are used in reverse engineering, but are of little import in software reuse (of old for the new) per se. Unfortunately, a lack of systematic exposition of knowledge on which these tools are based makes them seem like claims rather than true and tried solutions [Joyce, 1988]. Besides, none of the existing tools are designed specifically for migration to object-oriented systems.

Two related research efforts that are still underway need to be mentioned, but one cannot make a valid comparison between them and the Proto-ICAPE Project. One is conducted at the Micro Electronics and Computer Technology Corporation (MCC), Austin, Texas. The other is in progress at the University of Maryland, Baltimore, Maryland. The project at MCC on post-facto integration in conjunction with DELI is discussed in more detail in Section 2.4 under the heading DELI. Its primary focus is on reverse engineering. The idea of capturing a model of program and data abstractions in the existing software has been reportedly applied to two applications in systems software: window management, and persistent store management. Unlike the results of this dissertation, the reports are lacking in methodologies for software reuse that would be applicable to many

application domains. Additionally, the applications covered are rather small; a file system does not require much code.

Caldiera and Basili [1991] at the University of Maryland are investigating the problems of identification and "qualification" of reusable components. Their work on identification focuses on reusability that is of little relevance to ICAE and ICAPE. However, their work on qualification--specification, testing, encapsulation, and organization by classification--would be relevant to ICAPE and ICAE; but efforts are still in progress and results awaited.

Both these projects have yet to demonstrate the applicability of their techniques and tools on a scale beyond tens or hundreds of lines of code. The Proto-ICAPE Project has, in contrast, demonstrated its techniques on a larger scale; its subject, of which some parts are covered, consists of over a quarter (1/4) million lines of code, as described in the next chapter. Typical ICAPE or ICAE system would be of the order of millions or tens of million of lines of code; hence, scalability is an important requirement of any technique. In sum, the Proto-ICAPE Project has made substantial contributions in the field of software reuse with a systematic methodology and a demonstration of a reasonable scope of a non-trivial subject (as described in the next chapter).

3.4 Summary

Software reuse is a generalization of software integration in ICAPE. It can be approached in many ways, as examined in Section 3.2, but the most economical approach for ICAPE is reengineering with virgin reuse of components from the implementation of the subject software systems. A systematic methodology named REO embodying the chosen approach is presented. The REO methodology consists of three sets of methods: LANG to derive models from the programming

language descriptions; PROG to derive models from the program unit descriptions; and SIMP to remove some redundancies and simplify the derived object-oriented models. LANG consists of one method based on associations between the concepts of programming language grammar and object-orientation. PROG consists of three methods: CODE, in which the object code is directly reused; SORC, in which the source code is reverse engineered; and DOCU, in which descriptions in the manuals are rewritten in terms of the object-oriented model. SIMP presently consists of a few methods which are based on experience accumulated in the Proto-ICAPE Project.

Research in software reuse by others is reviewed and found lacking in both systematic methodologies and a primary focus on object-orientation. The ideas of software reuse are still new to the world of engineering computing.

4. Icape-91, A Prototypical ICAPE

The preceding chapter describes the REO methodology. This chapter describes its use in the development of a prototypical ICAPE system, Icape-91, that covers parts of ASPEN.† The development of the prototype can be considered as a case study in software reuse for ICAPE. This chapter consists of five sections. The first section gives a brief background on ASPEN. The second section gives a brief description of the ASPEN input language. The third section presents and justifies the scope of Icape-91. The fourth section describes the derivation of an object-oriented model for Icape-91. The final section discusses the design and implementation of Icape-91 in VSM. A detailed model and a VSM design is given in Appendixes A and B, respectively.

4.1 Background on ASPEN

ASPEN is a (steady state) software system well-known in the field of chemical engineering for chemical process modeling and simulation. One of its major user is to simulate a chemical process plant by a steady-state process flowsheet model. It is also used for various physical property computations (hereafter, "PP" stands for "physical property") such as generating tables and graphs of PP data, or estimating PP model parameters from experimental data.

The primary function of ASPEN is represented in the data flow diagram‡ in Figure 4.1. Given a chemical process description as input, ASPEN generates a data file called "Problem Data File," hereafter referred to as PDF, and a program for simulation. The PDF is a plex structured datafile; that is, a "linked-list" of

† For full name and its genesis, see footnote on page 5.

‡ In a data flow diagram an oval represents a process, a labeled arrow represents data values, and a pair of bars around a label represents a data store.

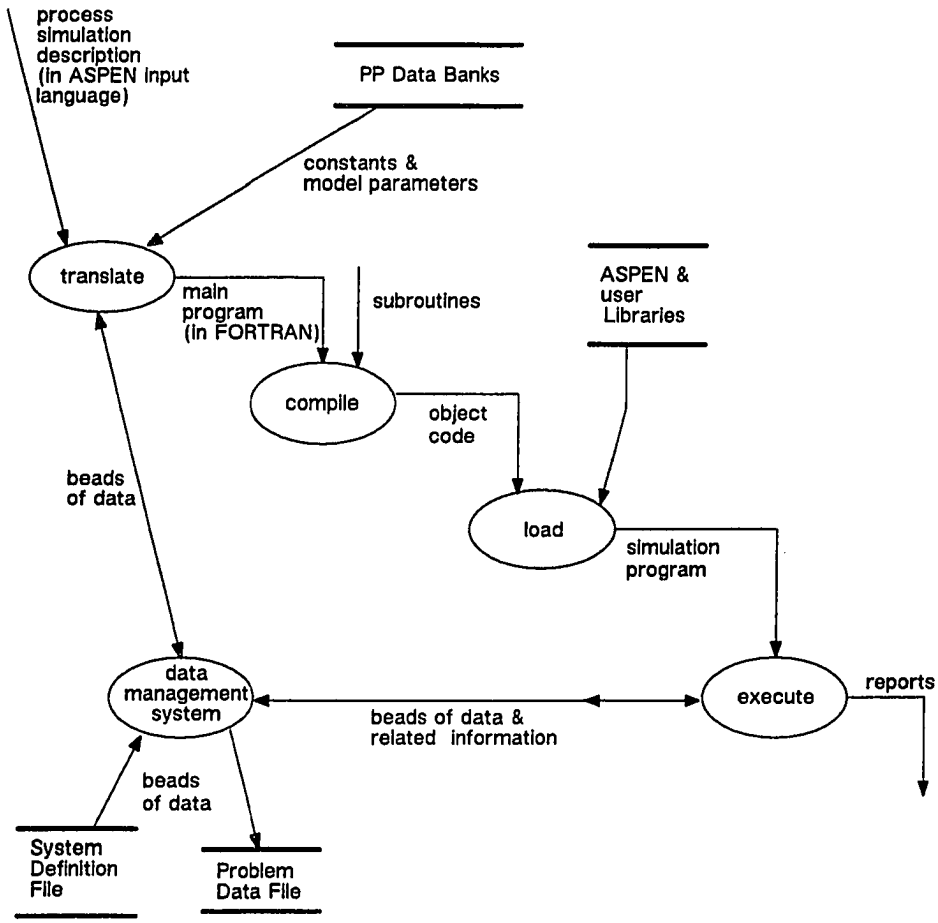


Figure 4.1 Data Flow Diagram for Simulation using ASPEN

"beads" of data including bead numbers (identifiers in a directory of beads) of the linked beads. This data structure is implemented as a large, statically allocated array in the main program in FORTRAN. The main program is compiled with user-supplied subroutines, and then the resulting object code is linked with codes from the system and user libraries into one load module, a simulation program that is executable. A simulation program in FORTRAN is generated for each input. The main reason behind this pre-processing approach is to include only the required subroutines and PP model parameter data, thereby reducing the size of the final program for loading. The data contained in PDF and other files for simulation history, error messages, etc., are updated during execution of the simulation program. Finally, reports of data in PDF are generated.

4.2 ASPEN Input Language

The ASPEN system recognizes input in a special language called "ASPEN input language." The ASPEN input language is similar to form-based languages. An input file in ASPEN input language is a hierarchically organized collection of keywords and data values. The hierarchical organization consists of three levels: primary, secondary and tertiary. Correspondingly, the ASPEN input language provides three constructs: paragraph, sentence, and value definition. A paragraph consists of a primary keyword and has many sentences. A sentence consists of a secondary keyword and has many value definitions. A value definition is given by a tertiary keyword, an equality sign and one or more data values (if value definitions are entered positionally, tertiary keywords are optional). The syntax analysis by the Input Translator is based on hierarchically organized tables of keywords and data value specifications in the System Definition File. The input is first converted into a Convenient Form Input format, an intermediate data structure that is closely

related to the specifications in the System Definition File. Some of the specifications for input file formatting, such as line continuation character and maximum number of characters per line, are handled by lexical analyzers. These are external to data of interest, and consequently these are not considered in this project.

As an example for familiarization with ASPEN input language, consider the sample input in Figure 4.2. It defines a flowsheet that consists of two blocks, a flow splitter named B1 and a mixer named B2. The output process streams S2 and S3 from the flow splitter are input to the mixer. (This is merely a hypothetical flowsheet used strictly for illustration.) The first paragraph begins with the primary keyword FLOWSHEET and the second begins with the secondary keyword BLOCK. The first paragraph has two sentences with BLOCK as the secondary keyword. The first of the two sentences starting with the keyword FRACTIONS has two value definitions, "STREAM-ID = S2" and "FRAC = 0.4," that together specify that the flow splitter B1 splits the input stream such that the output stream S2 is four-tenths of the input stream.

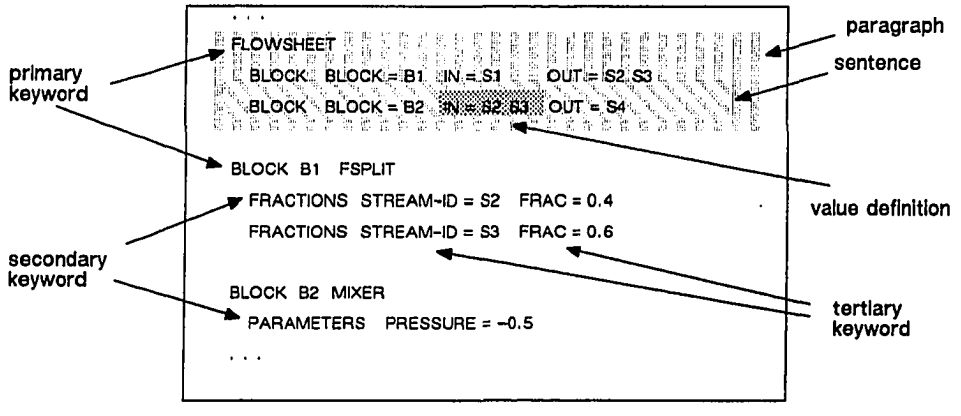


Figure 4.2 Section of a Sample Input File for ASPEN

4.3 Scope of Icape-91

The Proto-ICAPE Project covers only some parts, rather than the whole, of ASPEN for various reasons. First, the ASPEN system software is large with about a quarter (1/4) million lines of FORTRAN code [Motard, 1987]. There are over five thousand pages of manuals for the user and system administrator. Second, to cover all of ASPEN one would face extremely complex problems of testing and validation. The subjects of testing include both the "integrand" and the integration. The testing of integrand is not required for subjects that are well-developed and well-documented along with test suites. The testing of integration alone is a major cost factor in this project, mainly due to the project being the first of its kind; such costs, however, would diminish with accumulation of knowledge and experience. Presently, the testing of software and software components is usually done by a brute force approach. ASPEN was developed in the 1970's with thousands of man hours and millions of dollars from the National Science Foundation, the Department of Energy and over fifty industrial sponsors. At the time, methods of software engineering were not widely practiced; even today, it is the same to a large extent, although many advances are being made in the field of software engineering. One only has to guess the resources that one would need for testing alone if one were to integrate all parts of ASPEN. Third, it is sufficient to cover only parts of ASPEN for the aim is only to illustrate the applicability of REO techniques.

The PP subsystem of ASPEN is chosen for Icape-91 because almost all chemical engineers are familiar with various tasks in PP computation. In this dissertation, the task performed by the Table Generation System, hereafter referred to as TGS, is covered. TGS is used to generate tables of transport and thermodynamic property data for mixtures of components in many phases. The task of

parameter estimation of different PP models is handled by the Data Regression System.

4.4 Integration of ASPEN

Consider briefly the limitations of ASPEN in its current form first, after which the discussions on the derivation of object-oriented models are presented. First, the pre-processing approach is no longer required for modern environments that support dynamic linking and loading. (The problem is the non-modifiability of the structure, not only the data, of the plex data structure defined during pre-processing.) Second, the ASPEN system definition is not easy to modify. In order to add a new physical property model as a part of the system, one has to go through many steps including adding the statement label in the system definition file. Inclusion of the statement label is an example of very tight coupling: callers of the program unit, to which the code is added, control the "internal" execution of the called program unit. Third, even a simulation or model of a process cannot be modified in certain respects. Certain modifications that require a change in the plex data structure are impossible. For example, it is impossible to add a new chemical component after a simulation is created; it would require not only the extension of the data for all streams, but also the retrieval of various parameter data from the data bank and the updating of many parameter data of physical property routines. It is safe to say that these limitations are also largely true of the commercial versions of ASPEN that are based on the version in the public domain.

Thus ASPEN in its present form is not, or rather cannot be, highly interactive. Note that an interactive graphical user interface that interfaces with ASPEN at the input and output levels, a black box approach, does *not* make the system

interactive. For high interactivity, one would also have to cover the program units. This led to the possibility of applying the ideas and mechanisms of VSM to integrate ASPEN into Icape-91.

Unfortunately, VSM provides little help in integrating only parts rather than the whole (as discussed in Section 2.3.1) of ASPEN. In VSM, one would map data and code from the "application space" (the area of memory managed by the tool that is integrated) to the "VSM space" (as against the application space); that is, all tasks concerned with allocation and management of memory space are left to ASPEN. Thus, one has to include the complete subsystem for plex data management. (At this point, an obvious question that arises is why not use objects allocated in the VSM space, since VSM itself has almost all the functions one would need for the allocation and management of memory space?) All information required to organize data given in the input file and retrieved from other sources into the plex structure is coded or hard-wired in the pre-processing routines. Thus, one also has to include all routines employed in the pre-processing step, in addition to all routines that make a simulation. Note that in *no* way is the software or tool itself modified or improved. So, what is the net gain if integration inherits the encumbrances or the legacies of the old design, many of which are in conflict with the object-oriented paradigm? The only gain is that the data once mapped to objects can be shared with other objects and applications. Certainly, true object-orientation is lacking.

Thus, an alternative is to follow the REO methodology to derive object-oriented models and reuse implemented and tested code in Icape-91, as discussed in the next section.

4.5 REO for Icape-91

The derivation of object-oriented models following the REO methodology is described below by covering first the program input, then the output, and finally the program itself (any other order is admissible). The complete object-oriented model for the TGS subsystem will be called REO-TGS to indicate that it is derived by following the REO methodology and the subject is the TGS subsystem.

4.5.1 REO for Input

In this section, the derivation of the REO-TGS model for input to TGS in ASPEN input language is discussed. The first part (these parts are not marked) introduces an example of TGS input to familiarize the reader for the discussions that follow. The second part describes the syntax of fragments of the ASPEN input language. The third part presents the derivation of object-oriented models by applying the LANG method. The final part simplifies the derived model by applying the SIMP methods.

A sample input for TGS is shown in Figure 4.3. For a pair of chemical components (defined by the keyword COMPONENTS), a new set of PP tables is required (defined by the keyword PPTABLES) that would contain various properties (defined by the keyword DEP-VAR) of a mixture of the given components in a fixed composition (defined by the keyword SYSTEM) for a range of values (defined by the keyword RANGE) of temperature and pressure (defined by the keyword INDEP-VAR) based on a PP model of thermodynamic equations and correlations (defined by the keyword PROPERTIES). In other words, for given components, the sample input demands computation of fugacities, both in a pure and mixed state, and the enthalpies and volume of the mixture in the vapor phase. The two tables required (defined by the keyword TABLE) are for component

fugacities and various mixture properties. The PP models of thermodynamic equations and correlations (referred to by the keyword OPSETID) are discussed separately later.

```
COMPONENTS  COMP = METHANE  NAME = CH4
COMPONENTS  COMP = ETHANE    NAME = C2H6
PPTABLES  TBL34   PROPS
     DESCRIPTION   NAME = 'a sample'
     PROPERTIES   OPSETID = SYSOP0
     SYSTEM  NO = 1  COMP = METHANE  FLOW = 0.4 / COMP = ETHANE  FLOW = 0.6
     INDEP-VAR   NO = 1  VARNAME = TEMP
     RANGE        NO = 1   LIST  = 400.0  430.0  460.0  490.0  520.0
     INDEP-VAR   NO = 2  VARNAME =  PRES
     RANGE         NO = 2  START-VAL = 1.0e+5  FINAL-VAL = 2.0e+6  INCREMENT = 1.0e+5
     DEP-VAR      NO = 1  VARNAME = PHIVMX  PHIV
     DEP-VAR      NO = 2  VARNAME = HVMX  DHVMX  VVMX
     TABLE   NO = 1  HEADING = 'Mixed and Pure Component Fugacities'  &
      SYSTEM = 1   INDEP-VAR = 1  RANGE = 1   INDEP-VAR = 2  RANGE = 2  &
      DEP-VAR = 1
     TABLE  NO = 2   HEADING = 'Mixture Enthalpy, Enthalpy Departure & Volume'  &
      SYSTEM = 1   INDEP-VAR = 1  RANGE = 1   INDEP-VAR =  2  RANGE  =  2  &
      DEP-VAR = 2
```

Figure 4.3 A Sample Input for TGS

The parts of ASPEN input language relevant to TGS are specified in the EBNF notation and the regular definitions shown in Figure 4.4. The terminal symbol with only one lexeme pattern, such as the definition d19 in Figure 4.4, "**COMP** ::= COMP," is irrelevant, for it need not be associated with any class, instance or method. One may redefine the terminal symbol **prop** (see d17), as shown in Figure 4.4, in terms of the symbol for component (the terminal symbol **cm-prop**) and mixture (the terminal symbol **mx-prop**). For the sake of simplicity, this extension (see Figure 4.5) is excluded from this project. The full syntax for ASPEN input is informally described in Chapter 11 of the ASPEN User Manual, Volume 1 [Graham, 1982a].

d1) *tgs* ::= { *comps* } *pptbls*
d2) *comps* ::= **COMPONENTS COMP = cid NAME = name**
d3) *pptbls* ::= **PPTABLES tabid tabtyp** *snts*
d4) *snts* ::= [*desc*] [*mdl*] [*optn**] { *sys* } { *ivar* } { *dvars* } [*state**] { *rng* } { *tbl* }
d5) *desc* ::= **DESCRIPTION NAME = text**
d6) *mdl* ::= **PROPERTIES OPSETID = opsetid [SSCLISTID = scclistid]**
d7) *sys* ::= **SYSTEM NO = num { COMP = cid FLOW = num }**
d8) *ivar* ::= **INDEP-VAR NO = num VARNAME = ivarname**
d9) *dvars* ::= **DEP-VAR NO = num VARNAME =** {*dvarname*} [**PHASES = phasek**]
d10) *rng* ::= **RANGE NO = num** (*list-sp* | *range-sp*)
d11) *tbl* ::= **TABLE NO = num HEADING = text SYSTEM = num**
{ **INDEP-VAR = num RANGE = num** } **DEP-VAR = num**
d12) *dvarname* ::= **ivarname** | **prop**
d13) *list-sp* ::= **LIST = {num}**
d14) *range-sp* ::= **START-VAL = num FINAL-VAL = num INCREMENT = num**

* specified for other uses
example: sentence for *state* is needed if combination of independent variables *ivar* and range specification *rng* is not sufficient to completely specify the state of the system

PRODUCTIONS

d15) **tabtyp** ::= PROPS | FLASHCURVE | PTENVELOPE
d16) **ivarname** ::= TEMP | PRES | VFRAC | MOLEFRAC | MOLE
d17) **prop** ::= PHI | PHIMX | DHMX | †..... | SIG | SIGMX
d18) **phasek** ::= V | L | S | VL | VS | LS | VLS
d19) **COMP** ::= COMP

† more on pages 469-71 of ASPEN User Manual, Volume 1 [Graham, 1982a]

DEFINITIONS for some terminal symbols

Figure 4.4 Syntax Specification of Parts of the ASPEN Input Language for TGS

dA) **prop** ::= **cm-prop** | **mx-prop**
dB) **cm-prop** ::= PHIMX | **propfn**
dC) **mx-prop** ::= **propfn** MX
dD) **propfn** ::= PHI | H | S | G | V | DH | DS | DG | HXS | GXS | ... | MU | SIG | K | D

NOTE:
* cm-prop (see definition dB) is for pure and mixed components
* dD can be further redefined for property departures

Figure 4.5 Specification of Some Terminal Symbols in the ASPEN Input Language

Once the language syntax is specified, the LANG method is applied to derive an object-oriented data model, as shown in Figure 4.6, that is associated with the language specification. The model would have all the data necessary to generate input for the program. The production d3 (in Figure 4.4, the production d4 is split from d3 for convenience) is associated with the class named *pp_tables* (in Figure 4.6), d5 with the class named *description*, d6 with the class named *pp_model*, d11 (d11 is a group of productions) with the class named *range_def* and subclasses *list* and *range*, and d12 with the class named *table*. The definition d17 is associated with the class *property_keyword* and instantiations for each of the choices on the right side.

This model is further simplified by applying the SIMP methods (described in Section 3.2.3), as shown in Figure 4.7. If one applies the SIMP-1 method, the class *component* loses the attribute *id*, the class *pp_tables* loses the attribute *id*, the classes *system*, *independent_var*, *dependent_vars*, and *range_def* lose the attribute *number*, and so on. If one applies the SIMP-2 method, the class *description* with the only attribute named *name* of the type *text* is dropped from the model, and the attribute *desc* in class *pp_tables* is set to the type *text*. Similarly, the class *independent_var* with the only remaining attribute named *varname* of the type *ivarname* is dropped from the model, and the attribute *ivar* in class *pp_tables* is set to the type *ivarname*. Note that the class *list* should be retained because it is part of the hierarchy that is retained; the class *range_def* is retained because the class *range* is retained. If one applies the SIMP-3 method, the classes *property_keyword*, *phasek*, *ivarname*, *dvarname* (*dvarname* has only two subclasses, *property_keyword* and *ivarname*, both of which are dropped) and *tabtype* are dropped, and replaced by integral constants. Thus the attribute *tabtype* in class *pp_tables*, and the attributes *varname* and *phases* in class *dependent_vars*, are set to the type *integer*.

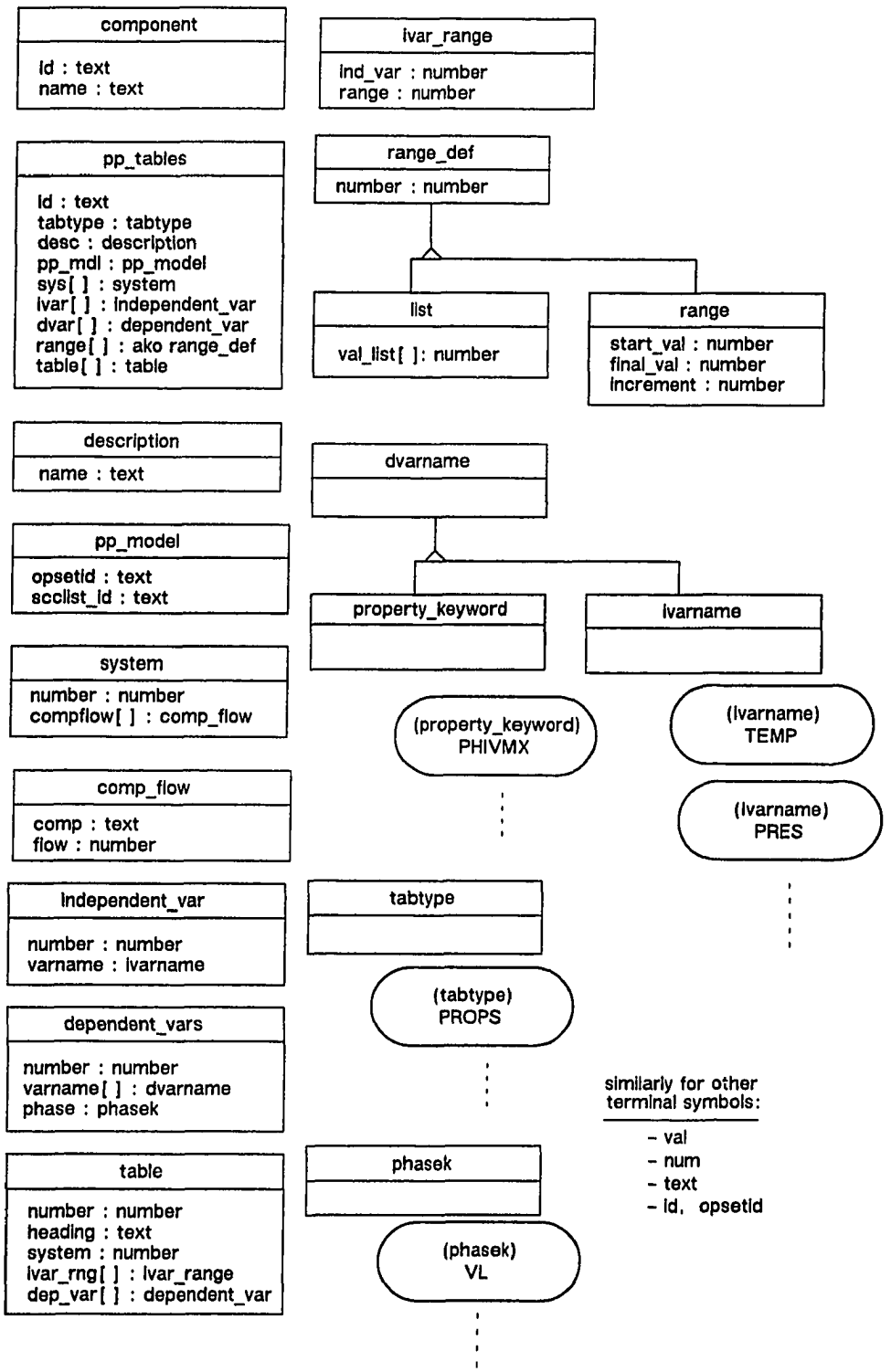


Figure 4.6 Application of the LANG Method to the Specifications in Figure 4.4 and 4.5

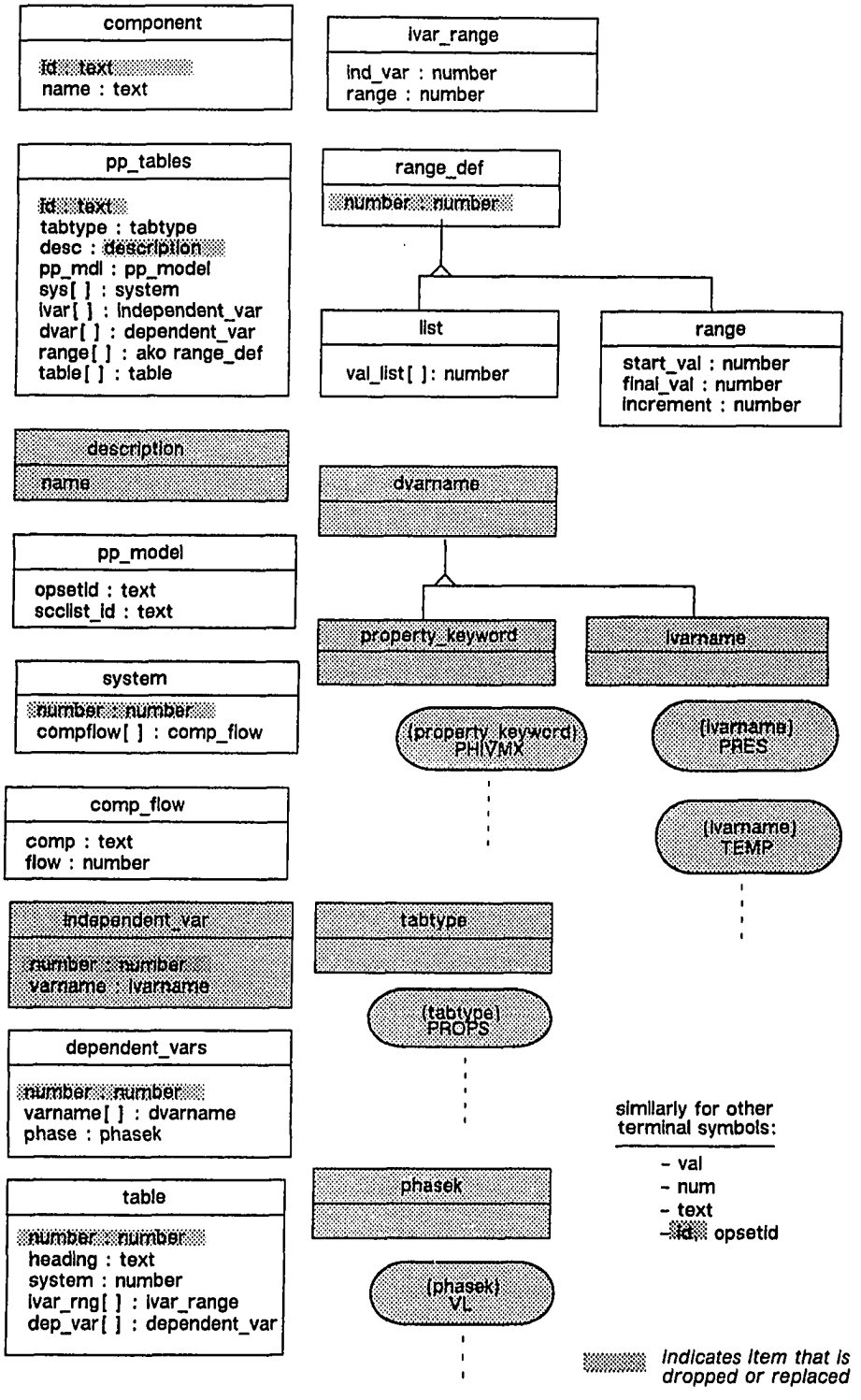


Figure 4.7 Application of the SIMP Methods to the Model Derived as Shown in Figure 4.6.

Next, consider the language fragments for defining PP models. But first some background is necessary. The ASPEN system provides a construct called "option set" for combining various equations and correlations for PP computation. (Why such a construct is named "option set" is not explained in ASPEN documentation.) Consider first a simple example illustrated in Figure 4.8 (it shows a way to calculate the fugacity coefficient of a component in the liquid phase) for a short introduction to the terminology of ASPEN; the terminology may confuse many chemical engineers who are not familiar it. In ASPEN, the equations that are thermodynamic derivations, such as the vapor liquid equilibrium equations shown in Figure 4.8 for PHILMX and PHIL properties, are referred to as "methods." The PP correlations and fitting equations, such as the Antoine equation for vapor pressure, the Andrade equation for viscosity, and the Redlich Kwong equation of state, are referred to as "models." Since ASPEN's terminology conflicts with that of object-oriented programming, the terms "AP† methods" and "AP models" are used instead of "methods" and "models," respectively.

In ASPEN, PP's are classified into three distinct categories: major properties, subordinate properties and intermediate properties. (In a strict sense, some of the commonly occurring terms, such as the Poynting correction, in thermodynamic calculations are not properties of physical substances. In ASPEN, however, these are expediently called physical properties.) The major properties are those that are required for simulating unit-operations; these include fugacity coefficient, enthalpy, entropy, free energy, molar volume, viscosity, diffusion coefficient, surface tension, and thermal conductivity. Other properties are required solely to compute major properties. These include the intermediate properties such as the

† Hereafter, the symbol "AP" stands for "ASPEN Physical Property Subsystem."

vapor pressure of liquid that can be computed only through AP models. The remaining PP's, the subordinate properties, are computed through AP methods; these include departure functions, excess functions, and pressure corrections for various thermodynamic properties. A combination of AP methods, AP models, and various approximations (defaults, specified through codes) that fully define a model for computing a physical property is called a *route*. A collection of routes is called an *option set*.

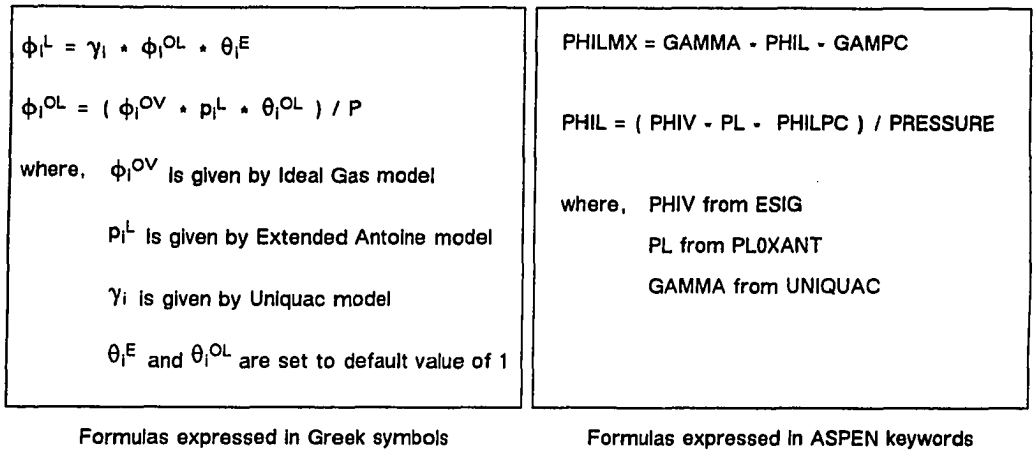


Figure 4.8 A Sample Route to Calculate Fugacity Coefficients

The ASPEN system provides many built–in option sets, routes, AP methods and AP models. The user can define new option sets, routes, AP methods and AP models either from scratch or as modifications of the existing ones. Figure 4.9 shows a sample input to create an option set and routes for the above example. The option set (defined by the keyword PROP–OPTIONS) consists of two major properties. An asterisk sign is used to denote null value. The two equations in Figure 4.8 are specified as the two routes for major properties in Figure 4.9 (defined by the keyword MP–ROUTE). A route is uniquely identified by a property

keyword and a "method code" (for example, the number 2 in MP-ROUTE sentences), and is completely defined when methods are specified for the input properties. The routes for input properties are specified through the keyword MPROP for the major properties, SPROP for the subordinate properties, and MODEL for the intermediate properties; all are specified in a particular order given in Appendix PP4.5 of the ASPEN User Manual, Volume 1 [Graham, 1982a]. For example, the input properties for PHIL using method code 2 (see Figure 4.9) are specified in the following order: PL, PHIV, and PHILPC. This requirement stems from hard-wired sequence control of program statements for calculating PHIL in a certain program unit. The handling of derivatives with respect to temperature and integrals over pressure complicates the program execution, but the ASPEN input language is unaffected.

$$\phi_i^L = \gamma_i * \phi_i^{OL} * \theta_i^E$$

$$\phi_i^{OL} = (\phi_i^{OV} * p_i^L * \theta_i^{OL}) / P$$

where, ϕ_i^{OV} is given by Ideal Gas model

p_i^L is given by Extended Antoine model

γ_i is given by Uniquac model

A Sample Option Set

```
. . .
PROP-OPTIONS  OPSET55 *  PHILMX PHILMX55 / PHIL PHIL55
MP-ROUTE  PHILMX55  PHILMX  2   *  ; for PHILMX
     MODEL  UNIQUAC                 ; for GAMMA
     MPROP  PHIL  PHIL55            ; for PHIL
                                    ; for GAMPC use default value
MP-ROUTE  PHIL55      PHIL     2   *  ; for PHIL
     MODEL  PL0XANT                 ; for PL
     MODEL  ESIG                    ; for PHIV
                                    ; for PHILPC use default value
. . .
```

Note: *"*" is for missing information.*
The names OPSET55, PHILMX55, PHIL55 are arbitrarily given.

Sample Option Set in ASPEN input language

Figure 4.9 A Sample Input for Defining an Option Set

The syntax of the language fragment for creating an option set is shown in Figure 4.10. The definitions d5, d6, and d7 describe the syntax for specifying an AP model for an intermediate or major property, a route for a major property, and a route for a subordinate property, respectively. The definitions d3 and d4 describe the syntax for specifying a route for major and subordinate properties, respectively. The definitions d1 and d2 together describe the syntax for specifying an option set. A new option set or route can also be defined as a "modification" of another option set or route, known as the "base." In Figure 4.10, an asterisk sign is used to indicate a null value for the base option set or base route. (Literally speaking, modification should mean only in-place changes, not a derivation of a new version.) This matter is excluded from the scope of Icape-91 because VSM—the implementation platform for Icape-91—does not support versions of objects. As discussed in the preceding paragraph, the full specifications for a route—the syntax of which are described by definitions d5, d6, and d7—must be stated in a fixed order given in Appendix PP4.5 of the ASPEN User Manual, Volume 1 [Graham, 1982a].

The LANG method is applied to the language specification presented above resulting in an object-oriented model shown in Figure 4.11. Classes are created for each production: *option_set* for d1; *major_prop_spec* for d2; *mprop_route* for d3; *sprop_route* for d4; and *mdl_spec*, *mprop_spec*, and *sprop_spec* for d5, d6 and d7 respectively. Classes and instantiations are created for each regular definition: *prop_kwd* and *sprop_kwd* for d10 and d11, respectively; instances for every choice of tokens on the right side of d10 and d11. As before, any regular definition with only one lexeme pattern, such as d12, need not be considered.

```
d1 ) opset      ::=   PROP-OPTIONS opsetid *  { mprp }
d2 ) mprp       ::=   MP-KWD = mpkwd  MP-ROUTE = id
d3 ) mprt       ::=   MP-ROUTE ROUTE-ID = id KEYWORD = mpkwd  METHOD = num *
                      [ { mdl } ] [ { mprp } ] [ { sprp } ]
d4 ) sprt       ::=   SP-ROUTE ROUTE-ID = id KEYWORD = spkwd  METHOD = num *
                      [ { mdl } ] [ { mprp } ] [ { sprp } ]
d5 ) mdl        ::=   MODEL  MODEL = text
d6 ) mprp       ::=   MPROP  MP-KWD = mpkwd  MP-ROUTE = id
d7 ) sprp       ::=   SPROP  SP-KWD = spkwd  SP-ROUTE = id
```

PRODUCTIONS

```
d10) mpkwd      ::=   PHIV | PHIVMX | PHIL | PHILMX | †..... | SIGL | SIGLMX
d11) spkwd      ::=   DHV | DHVMX | DHL | DHLMX | †..... | GAMPC | HNRYPC
d12) MPROP      ::=   MPROP
```

† more on page 469-71 of ASPEN User Manual, Volume 1 [Graham, 1982a]

DEFINITIONS for some terminal symbols

Figure 4.10 Syntax Specification of Parts of the ASPEN Input Language for Option Set

This model shown in Figure 4.11 is simplified by applying the SIMP methods (refer to Section 3.2.3) as shown in Figure 4.12. If one applies the SIMP–1 method, the class *option_set* loses the attribute *optionset_id* and *mprop_route*, and the class *sprop_route* loses the attribute *id*. If one applies the SIMP–2 method, the class named *mdl_spec* is dropped and the attribute named *mdl_spec* of classes named *mprop_route* and *sprop_route* is thus set to the type *text*. If one applies the SIMP–4 method, of the two equivalent classes *major_prop_spec* and *mprop_spec*, the former is dropped. Similarly, the classes *mprop_kwd* and *sprop_kwd* are replaced with *prop_kwd*, and its set of instances is a union of a set of instances of the replaced classes. Furthermore, the two attributes *mprop_kwd* and *sprop_kwd* in the classes *mprop_route* and *sprop_route* are set to be of the type *prop_kwd*. Similarly, the type of two attributes *mprop_kwd* and *sprop_kwd* in the classes

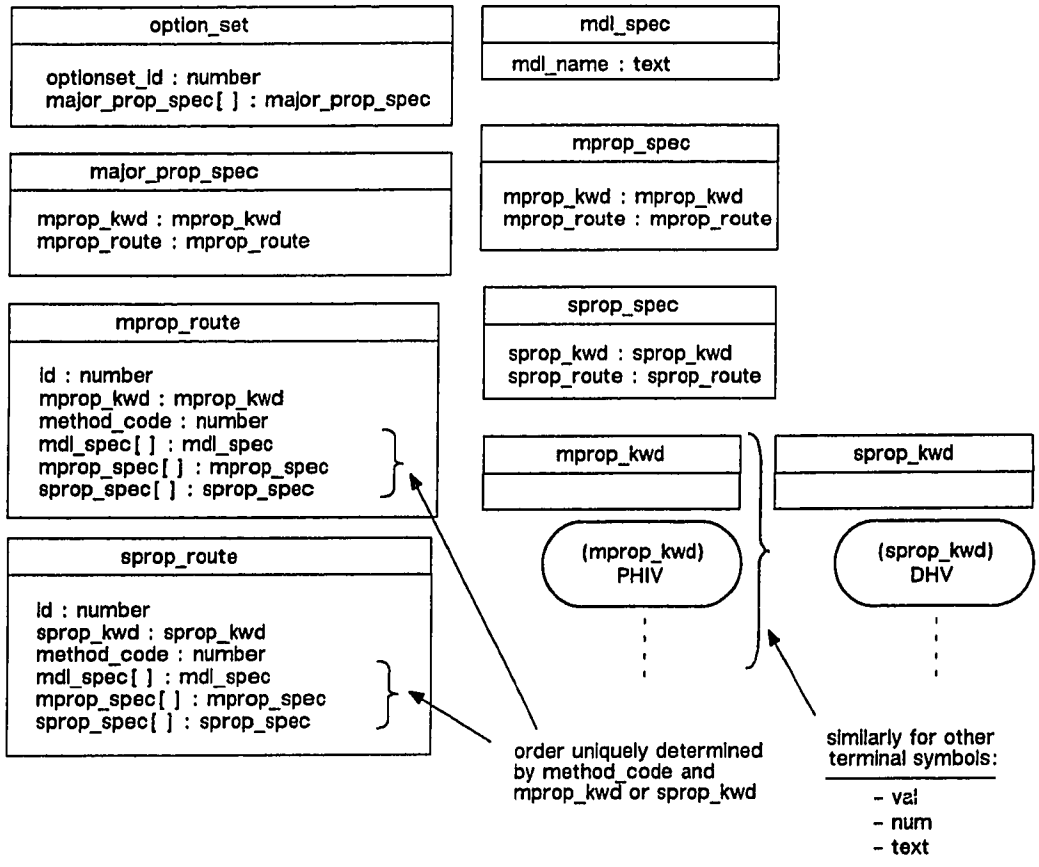


Figure 4.11 Application of the LANG Method to the Specifications in Figure 4.10

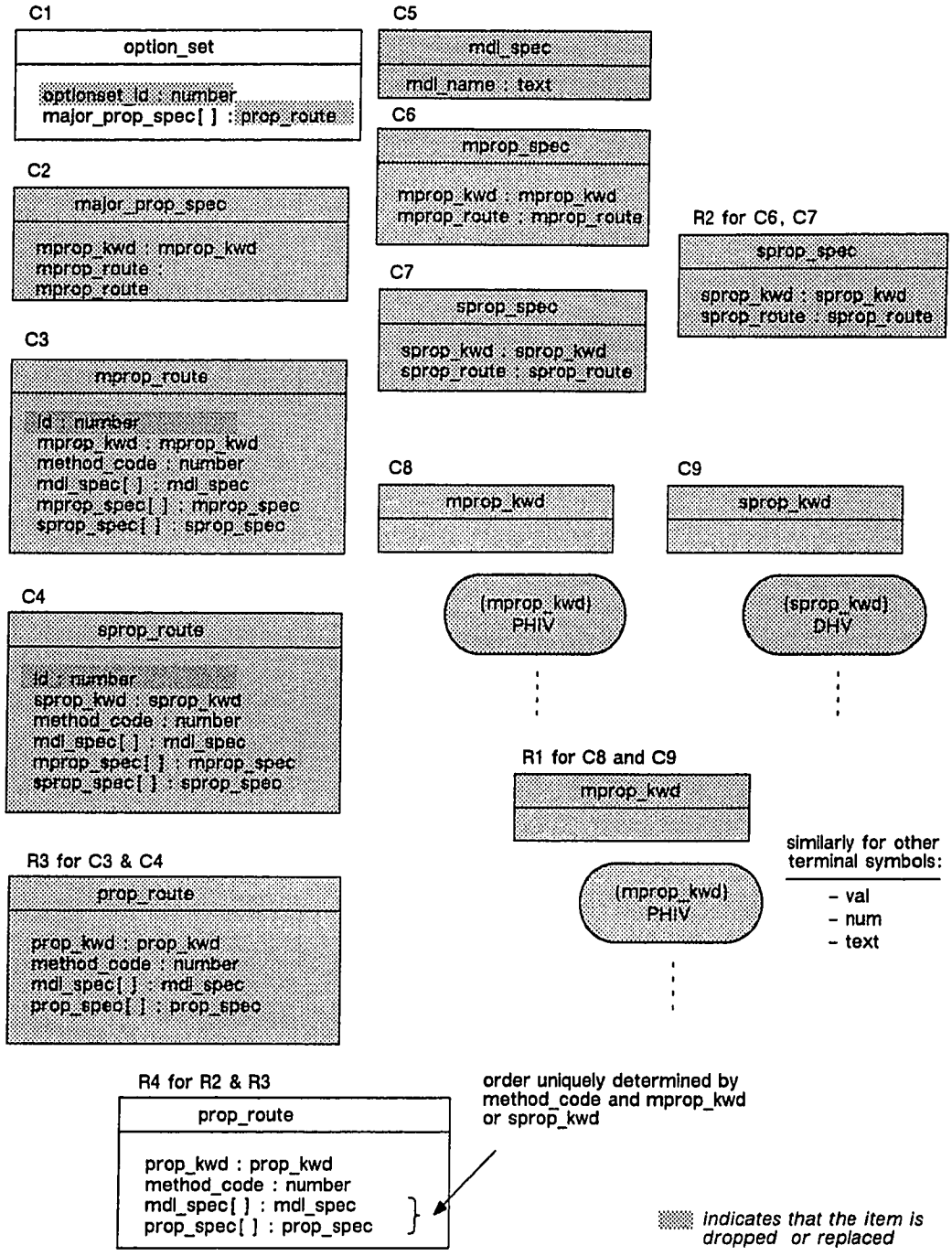


Figure 4.12 Application of the SIMP Methods to the Model Derived as Shown in Figure 4.11

mprop_spec and *sprop_spec* is set to *prop_kwd*. Again, according to the SIMP-4 method, the two classes *mprop_spec* and *sprop_spec* are replaced with the class *prop_spec*. The classes *mprop_route* and *sprop_route* are replaced with *prop_route*; and the two attributes *mprop_spec* and *sprop_spec* in the resulting new candidate class *prop_route* are merged into the attribute *prop_spec*, and the type of attribute *prop_route* in the new candidate class *prop_spec* is set to *prop_route*. If one applies the SIMP-3 method, the class *prop_kwd* is dropped and its instances replaced by integral constants; and the type of attribute *prop_kwd* in the new classes *prop_route* and *prop_spec* are set to the type *integer*. Since the class *prop_spec* is extraneous, according to the SIMP-5 method it is merged with *prop_route*. The resulting model after application of the SIMP methods is clearer, smaller, simpler and has fewer classes. This model consists of only two classes, *option_set*† and *prop_route* (C1 and R4 in Figure 4.12), and can be used to redefine the ASPEN input language for conciseness and simplicity. Note that the three PP categories, major, intermediate, and subordinate, are no longer present; furthermore, they are not found in the subject of thermodynamics itself. This classification of properties is specific only to the ASPEN system and stems from implementation considerations.

The complete REO-TGS model that combines the model in Figure 4.6 and 4.12 is presented in Figure 4.13. The SIMP methods can be further applied to eliminate redundant--duplicate, equivalent, and extraneous--classes, if any. The model so far includes only the structural aspects of data in TGS input, but it

† In terms of graph theory, an option set is a directed acyclic graph with nodes consisting of AP methods and AP models. A route is a tree in such a graph, a collection of connected nodes with a root node for computing a particular major property.

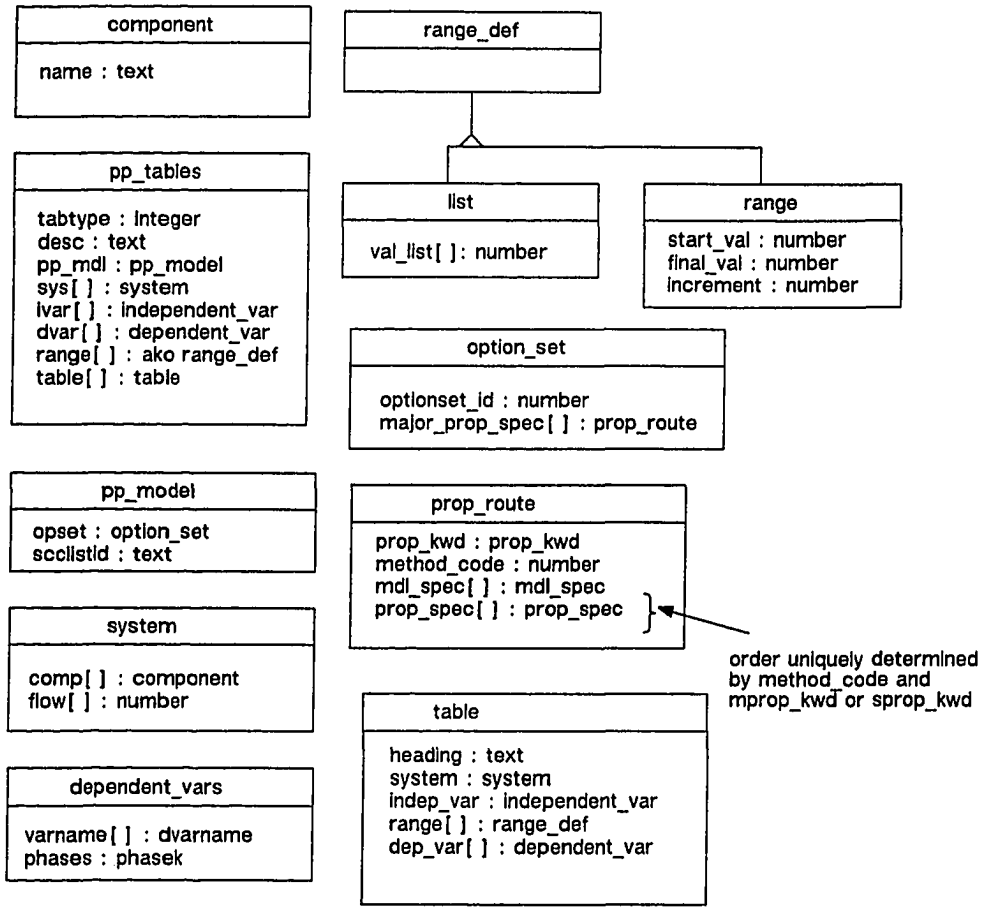


Figure 4.13 REO-TGS, An Object Oriented Model of Input Data for TGS

should be extended with behavioral aspects, at the least, with methods to update the objects.

4.5.2 REO for Output

The above modeling process can be repeated for the TGS program output language, if any. The output from TGS consists of tabularly formatted PP data for the specified set of dependent variables for the given mixture of components, option set, and range of values for independent variables. These outputs are in custom formats, and there is no language discipline. For Icape–91, the TGS output is not considered for the following reasons. First, it is rather tedious to create a language definition by analyzing the code or many samples of many kinds of outputs. Second, if both the input and the program are covered, then logically it is not necessary to cover the output. Third, the goal of the Proto–ICAPE Project is to integrate programs at a level deeper than the black–box at input and output level.

4.5.3 REO for Program

The program units that are built into or may be generated by ASPEN are subjected to the PROG methods of reuse in the manner discussed in Section 3.2.2. Processed reuse of a selected program unit involves deriving and associating one or more classes.

The program structure diagram for TGS is "rooted" in a program unit called ASPEN that is generated by the input translator. Actually, the input translator generates a program named MAIN and a subprogram named ASPEN; MAIN consists of only one executable statement, a call to ASPEN. The handling of in–line FORTRAN statements will not be discussed to keep matters simple. A

typical ASPEN subroutine is given in Section 3.4 of the ASPEN System Administration Manual, Volume 1 [Graham, 1982b].

The candidate program units for the TGS program are listed in Table 4.1 along with the reason for their selection, whether they are relevant or bridge program units. The list does not expand the program units that have no selected dependents, direct or indirect. For example, none of the dependents of EXMON are selected, hence they are not included in the table. ASPEN calls INTSIM, PPLOAD, LDSTW, EXMON, SEQMON, RPTMON and other subroutines (the name is given as "MNF" suffixed with an integer) that are generated for each process unit in the process flowsheet. The processing of input is done by TGS1 and TGSI, the latter called by RPTMON (see Chapter 10 of the System Administration Manual, Volume 2 [Graham, 1982c]). TGSI and TGS1 are relevant program units because both affect objects in REO-TGS as stated in their function definition. RPTMON is a bridge program unit between two relevant program units, MAIN and TGSI. WRITTBL processes the specification of independent variables and their ranges in the input for TGS that are associated with objects in REO-TGS, thus it is a relevant program unit. THERMO, TXPORT, EOSMON, and others are relevant program units by similar arguments. ERROR and LERRPT are not qualified as relevant or bridge program units because their function is solely to generate outputs (create logs of error reports).

The next step involves selecting a method of reuse, DOCU, SORC, or CODE, for each of the selected program units, as shown in Table 4.2. The DOCU method is chosen for all bridge program units. Other methods are selected, as discussed in Section 3.2.2, for the relevant program units. The code of TGSI or TGS1 is too complex for direct reuse because it requires many data structures and program units that are not selected (in principle, any arbitrary constraint may be

Table 4.1 Selection of Candidate Program Units

Callers	Program Units	Function Definition	Select? Reasons if affirmative.
MAIN	ASPEN	Simulate : A process model in ASPEN input language.	Yes. Relevant (starting node).
ASPEN	INTSIM	Initialization : Files, COMMONs.	
	PPLOAD	Load : PP data.	
	LDSTW	Load : Stream work COMMON.	
	EXMON	Decide : Simulation, report writing, or both.	
	SEQMON	Decide : Block for simulation.	
	RPTMON	Output (Report) : Unit Operations, Physical Properties, Streams, Cost, Economics, Data Regression, Table Generation.	Yes. Bridge: ASPEN, TGSI.
	MNFn	Simulate : A block.	
RPTMON	REPORT	Output (Report) : Layout and Table of Contents of a report.	
	FLWRPT	Output : Flowsheet section.	
	PRPRPT	Output : Physical Properties.	
	UOSRPT	Output : Unit Operations.	
	STRRPT	Output : Report for Streams.	
	CSTRPT	Output : Report for Cost.	
	ECORPT	Output : Economic Evaluation.	
	DRSI	Output : Data Regression.	
	TGSI	Output : Table Generation. Interface to TGS routine TGS1. Process : PPTABLES occurence.	Yes. Bridge: RPTMON, TGS1. Relevant to REO-TGS.

Table 4.1, continued

Callers	Program Units	Function Definition	Select? Reasons if affirmative.
TGSI	TGS1	Process : TABLE specs.	Yes. Relevant to REO-TGS.
TGS1	WRTTBL	Calculate : Independent variables with the given range specifications.	Yes. Relevant to REO-TGS.
WRTTBL	CMON	Print : Computed (through subroutines) values of dependent variables.	Yes. Bridge: WRTTBL, VTHRM.
	XFLASH	Calculate : VLE data.	
	PTENVI	Calculate : PT envelope.	
CMON	VTHRM	Calculate : Certain vapor phase properties	Yes. Bridge: CMON, THERMO.
	LTHRM	Calculate : Certain liquid phase properties.	– do –
	...	...	...
	SRFTEN	Calculate : Surface tension properties.	– do –
VTHRM, LTHRM, ... SRFTEN	THERMO, TXPORT	Calculate: Resolved route bead (set of GOTO labels) for an option set.	Yes. Relevant to REO-TGS.
THERMO, TXPORT	EOSMON	Calculate: GOTO label for equation of state AP model from the equation of state bead.	Yes. Relevant to REO-TGS.
	IGMON	Calculate : Use ideal-gas equation of state AP model.	Yes. Relevant to REO-TGS.
	CALMON	Calculate : Major properties using information in resolved route bead for the option set.	Yes. Relevant to REO-TGS.

Table 4.1, continued

Callers	Program Units	Function Definition	Select? Reasons if affirmative.
EOSMON	ES01, ES02, ES03, ...	Calculate : Use a specific equation of state AP model.	Yes. Relevant to REO-TGS.
CALMON	DFTMON	Calculate : Default values of major properties.	Yes. Relevant to REO-TGS.
	CLMON1, CLMON2, CLMON3	Calculate : Physical property through AP methods.	Yes. Relevant to REO-TGS.
CLMON1, CLMON2, CLMON3	MODMON	Calculate : GOTO label for AP models, temperature derivatives, pressure integrals of various properties.	Yes. Relevant to REO-TGS.
MODMON	MDMON1, MDMON2, MDMON3	Calculate : Physical property through AP models.	Yes. Relevant to REO-TGS.
MDMON1	PL002	Calculate : Use Cavett vapor pressure AP model.	Yes. Relevant to REO-TGS.
	PL001	Calculate : Use Extended Antoine vapor pressure AP model.	-do-
	...	...	...
MDMON2	...	...	...
MDMON3	...	...	...
PL001, PL002, ... ES01, ... IDLGAS	ERROR	Record : Errors Stop : MAIN if the limit on errors is reached.	
	LERRPT	Record : Errors Stop : MAIN if the limit on errors is reached.	

imposed for integration); hence, the CODE method is inapplicable. Furthermore, it is not necessary to capture the program structure, the internals, of both TGSI and TGS1; thus, the SORC method is not applicable. The only remaining method of reuse, the DOCU method, is applicable and selected for TGSI and TGS1. In contrast, the program structure of the WRTTBL unit is relevant, and its source form is not too complex for reverse-engineering and the object code is not directly reusable; thus the SORC method is chosen for its reuse. The code of PL001 to IDLGS program units can be directly reused; for these the CODE method is selected. A reuse method is chosen for other program units in the same manner.

Table 4.2 Program Units and the Selected Method of Reuse

Program Unit	Type†	Method of Program Reuse‡	Program Unit	Type†	Method of Program Reuse‡
ASPEN	R	DOCU	DFTMON	R	DOCU
RPTMON	B	DOCU	CLMON1	R	SORC
TGSI	R	DOCU	CLMON2	R	SORC
TGS1	R	DOCU	CLMON3	R	SORC
WRTTBL	R	SORC	MODMON	R	SORC
CMON	B	DOCU	MDMON1	R	SORC
VTHRM	B	DOCU	MDMON2	R	SORC
LTHRM	B	DOCU	MDMON3	R	SORC
...	..	...	PL001	R	CODE
SRFTEN	B	DOCU	PL002	R	CODE
THERMO	R	DOCU	...	..	...
EOSMON	R	SORC	ES01	R	CODE
IGMON	R	SORC	...	..	...
CALMON	R	SORC	IDLGAS	R	CODE

† "B" for bridge program unit, "R" for relevant program unit.
‡ for details see Section 3.2.2

The next step is to associate one or more classes with each program unit, as shown in Table 4.3. If none of the classes from the current set in the object-oriented model is associable, then a new class is created and its structure and

Table 4.3 Selected Program Units and Their Associated Classes

Program Unit	Type†	Classes
ASPEN	R	-
RPTMON	B	-
TGSI	R	pp_tables‡
TGS1	R	table‡
WRTTBL	R	table‡
CMON	B	-
VTHRM	B	-
LTHRM	B	-
...		
SRFTEN	B	-
THERMO	R	option_set‡
EOSMON	R	an indexed_collection
IGMON	R	ideal_gas
CALMON	R	option_set‡
DFTMON	R	an indexed_collection
CLMON1	R	an indexed collection, AP_method_1_1, ... AP_method_1_p
CLMON2	R	an indexed_collection, AP_method_2_1, ... AP_method_2_q
CLMON3	R	an indexed_collection, AP_method_3_1, ... AP_method_3_r
MODMON	R	APmodel
MDMON1	R	an indexed_collection
MDMON2	R	an indexed_collection
MDMON3	R	an indexed_collection

Program Unit	Type†	Classes
PL001	R	extended_antoine
PL002	R	cavett_vapor_pressure
PS001	R	solid_anotoine
VL004	R	rackett
VL201	R	cavett
KL002	R	sato_reidel
KL201	R	vredeveld
MUL001	R	andrade
MUL201	R	log_average_mixing
MUV001	R	chapman_enskog
MUV201	R	brokaw
MUV202	R	dean_stiel
GM01	R	scatchard_hildebrand
GM03	R	wilson
GM04	R	vanlaar
GM05	R	renon
PHL001	R	grayson_streed
SIG002	R	hakim_steinberg_stiel
SIG201	R	power_law_mixing
DL101	R	wilke_chang
DV001	R	chapman_enskog_wilke_lee
DV002	R	dawson_khoury_kobayashi
DV101	R	blanc
ES01	R	redlich_kwong
ES00	R	ideal_gas
...		
IDLGAS	R	ideal_gas_heat_capacity

† "B" for bridge program unit, "R" for relevant program unit

‡ from REO-TGS, others are new classes

dynamics are defined. A list of associable, new and old, classes for the selected program units is shown in Table 4.3. The following paragraphs discuss the derivation from a couple of selected program units illustrating the application of each method of reuse in Icape-91 (derivation from *all* selected program units is rather complex and would be wearisome to read.)

CODE: PL001 to IDLGAS

The following discussion is mainly about PL002 instead of all directly reused program units. The modeling of shared data is discussed first, and then the program unit itself. The COMMON blocks for some of these program units are listed in Table 4.4; the ones that are specific to the program unit, that is, those not found in others, are highlighted. The first entry in the table for the class *extended_antoine* associated with the subroutine PL001 lists the attributes *global*, *ncomp*, and *coeff* (to be defined) for the COMMON blocks GLOBAL, NCOMP, and PLXANT, respectively. The data declaration for the highlighted COMMON block PLXANT is found only in PL001. Similarly, the entry for the class *cavett_vapor_pressure* associated with the subroutine PL002 lists the five attributes *global*, *ncomp*, *tc*, *pc*, and *coeff* for the COMMON blocks GLOBAL, NCOMP, TC, PC, and PLCAVT, respectively. The highlighted COMMON block PLCAVT is found only in PL002.

The classes for the COMMON blocks are defined based on descriptions in programs and manuals, most of which are informal. The specifications for some of the COMMON blocks from those listed in Table 4.4 are given in Figure 4.14. The COMMON blocks COMP, TC, PC, ZC, PLCAVT, FRMULA, and LJPAR (and MW, VC, TB, VB, OMEGA, and STKPAR not in Figure 4.14) hold arrays of values for various universal constants for chemical compounds. The data in these COMMON blocks are ordered identically; that is, TC(4) in TC block, PC(4) in

Table 4.4 Classes Associated with Program Units for AP Models

Class	Program Unit	Attributes/COMMON blocks
extended_antoine	PL001	global/GLOBAL, ncomp/NCOMP, coeff/PLXANT
cavett_vapor_pressure	PL002	global/GLOBAL, ncomp/NCOMP, tc/TC, pc/PC, coeff/PLCAVT
solid_antoine	PS001	global/GLOBAL, ncomp/NCOMP, coeff/PSANT
rackett	VL004	global/GLOBAL, ncomp/NCOMP, ppglob/PPGLOB, tc/TC, pc/PC, coeff/RKTZRA

indicates attributes or COMMON blocks that are not shared with other classes or program units

PC block, and C(3,4) in PLCAVT block store data for the same compound given in COMP(4) in COMP block. This relationship can also be inferred by examining the iteration structures in the source code, if the documentation is lacking. Thus, the order of data in COMP can serve as an index to data in TC, PC or others holding data about chemical compounds. As shown in Figure 4.15, the new classes *global*, *rglob*, *ncomp*, and *ppglob* are defined based on the specifications d1 to d4 in Figure 4.14 for the COMMON blocks GLOBAL, RGLOB, NCOMP, and PPGLOB, respectively. Similarly, the classes *comp*, *tc*, *pc*, *zc*, *formula*, *LJparam*, and *coeff* are defined for the COMMON blocks COMP, TC, PC, ZC, FRMULA, LJPAR, and PLCAVT, as specified in d6 to d12 (the classes for MW, VC, TB, VB, OMEGA, and STKPAR are not shown). Corresponding to the ordering relationship between data in these COMMON blocks, a relationship between the attributes of associated classes is defined (see the dashed line connecting the boxes for

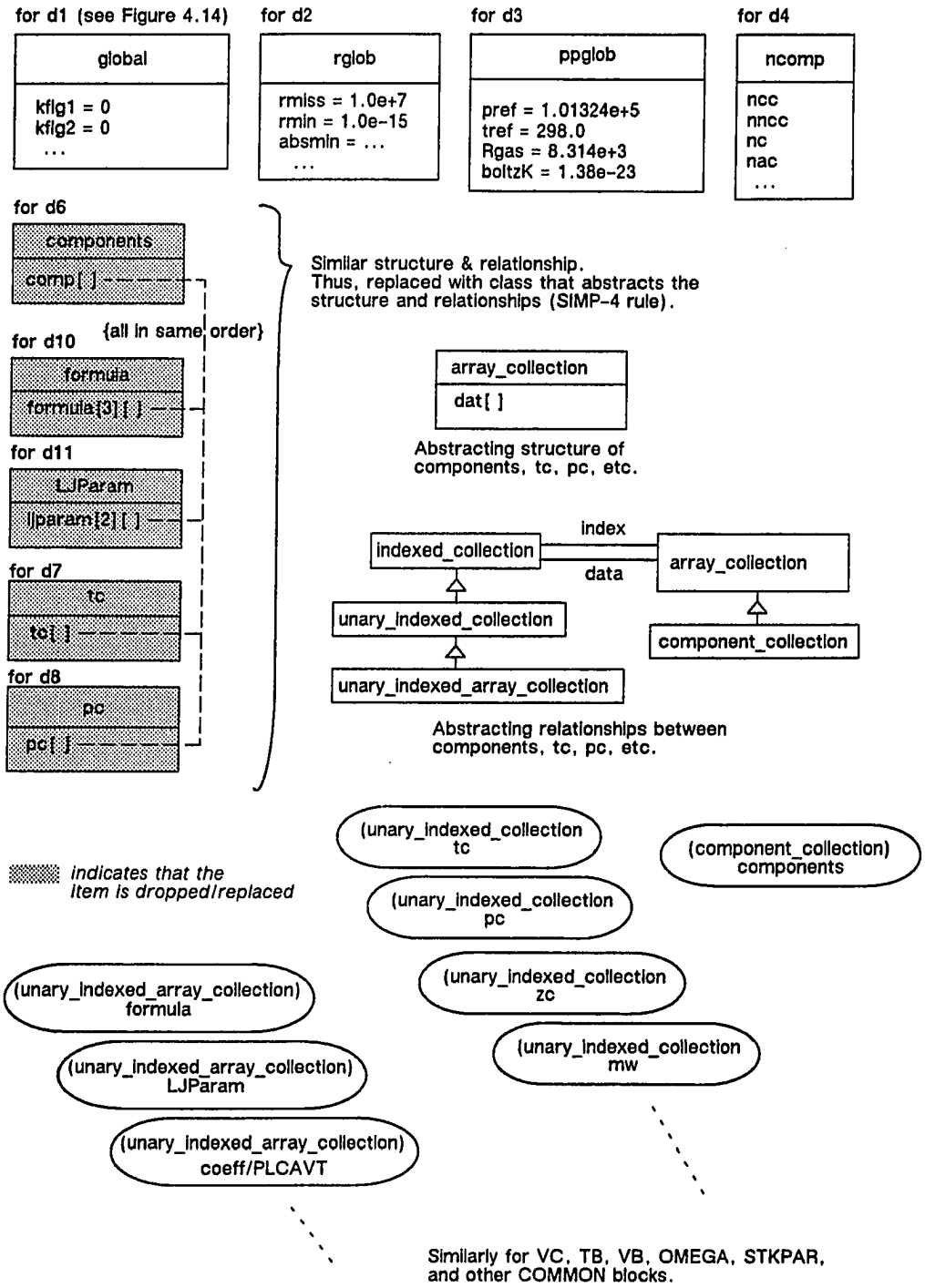


Figure 4.14 Some Structural Parts of the REO-TGS Model for the COMMON Blocks

classes *components*, *formula*, etc., in Figure 4.15). The model is further transformed by applying the SIMP–4 method. The classes *components*, *formula*, *LJParam*, *tc*, and *pc* for component data in Figure 4.15 are replaced by classes of higher abstraction: *array_collection*, *indexed_collection*, *unary_indexed_collection*, and *unary_indexed_array_collection*. The class *array_collection* abstracts the structural feature of each class associated with the COMMON blocks that are defined in d6 to d12; the structural feature of these classes is that each consists of an array variable. The class *components* may be replaced with *array_collection*. Instead, in order to restrict the data values to a specific range of values, it is replaced with *component_collection*, which is a kind of *array_collection*. The ordering relationship between the attributes of two classes, *component_collection* and others, is abstracted in the class *indexed_collection*. This relationship is further constrained to a functional dependency between a key value and a data value, scalar or vector, in the subclasses *unary_indexed_collection* and *unary_indexed_array_collection*. This relationship is constrained to be a functional dependency between a pair of key values and a data value in the class *binary_indexed_collection*; such a class is for the COMMON blocks that store binary (for a pair of components) parameters. The same procedure is repeated with other COMMON blocks in other program units under the CODE method.

As regards the dynamics of these classes, none of the selected program units affect the data in the COMMON blocks TC, PC, etc. (these data are only read by PL001, as one can readily infer by a visual scan of the program). New methods for these classes are required to manipulate these objects and maintain the relationship between *index* and *data* attributes. Consequently, four methods are defined: *indexedby* to bind an indexing object, *atput* to insert data for a particular key value, *atreplace* to update data for a particular key value, and *at* to retrieve

data for a particular key value. The specifications of these methods, the number and type of arguments, are different for different subclasses of *indexed_collection*. A method *set* with two arguments, the attribute name and a value to be assigned to the attribute, is created for the classes *global*, *rglob*, *ppglob*, and *ncomp*. Any update, of course, should also satisfy any "intra-class" constraints between the attributes.

d1) COMMON /GLOBAL/ KPFLG1, KPFGL2, KPFLG3, LABORT, NH, ...IRNCLS
d2) COMMON /RGLOB/ RMISS, RMIN, ABSMIN, SCLMIN, XMIN, HSCALE, ...TNOW
d3) COMMON /PPGLOB/ PREF, TREF, RGAS, KBOLT
d4) COMMON /NCOMP / NCC, NNCC, NC, NAC, NACC, NVCP, NVNCP, NVACC, NVANCC
d5) COMMON /PPWORK/ WORK(1)
d6) COMMON /COMP/ COMP(1)
d7) COMMON /TC/ TC(1)
d8) COMMON /PC/ PC(1)
d9) COMMON /ZC/ ZC(1)
d10) COMMON /FRMULA/ FRMULA(3,1)
d11) COMMON /LJPAR/LJPAR(2,1)
d12) COMMON /PLCAVT/ C(4,1)

Note: A dimension of an array in COMMON block is stated to be 1 in d5 to d11, since the actual size is determined in some other program unit.

Relationships:

The data structures defined by d6 (COMP) to d11 are related. The order of data defined by d6 (COMP) is the same as that defined by d7 to d11 along the last dimension (of size 1). Thus, the order of data in COMP can serve as an index to data in TC, PC, ZC, FRMULA, LJPAR.

COMMON Specifications

Figure 4.15 Data Specifications Given in the TGS Program Units

Recall that certain attributes associated with the COMMON blocks are shared with other classes (see Table 4.4), and these can be aggregated. A list of common attributes in various program units is prepared, as shown in Figure 4.16. The objects from this list that are related are aggregated into new classes; for example, the object *comp* that is an instance of class *component_collection* is part of objects such as *tc* and *pc* (*comp* serves as an index of *tc*, *pc*, etc., as shown in the class *unary_indexed_collection* in Figure 4.15). The class *universal_constants* is

created (see Figure 4.16) for this aggregation of objects that share an instance of *component_collection* as their indexing object. The rest are aggregated into another class called *aspen* (because the COMMON blocks GLOBAL, RGLOB, NCOMP, and PPGLOB are specific to the ASPEN simulator). This reorganization is reflected in all other classes for AP models; the common attributes are replaced with (fewer) attributes that hold instances of the new aggregate classes *aspen* and *universal_constants*.

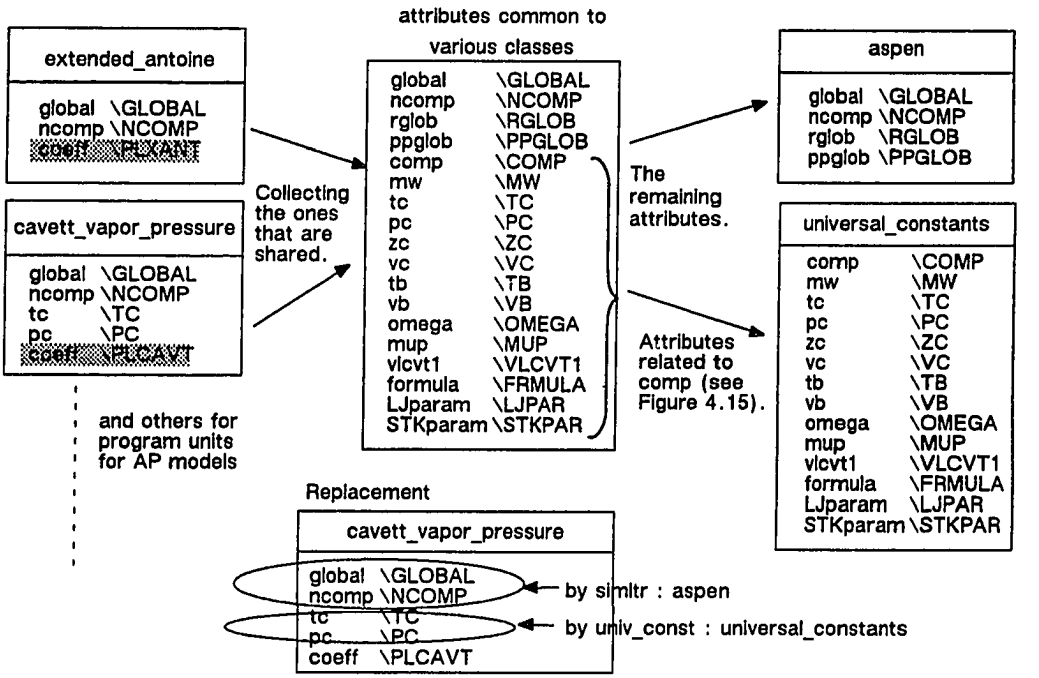


Figure 4.16 Aggregation of Classes' Attributes for the Shared COMMON Blocks in the Program Units

Turning attention back to the program itself, consider the subroutine PL002 and its associated class *cavett_vapor_pressure* (see Figure 4.17). The primary function of PL002 is the calculation of pure component vapor pressure in the liquid

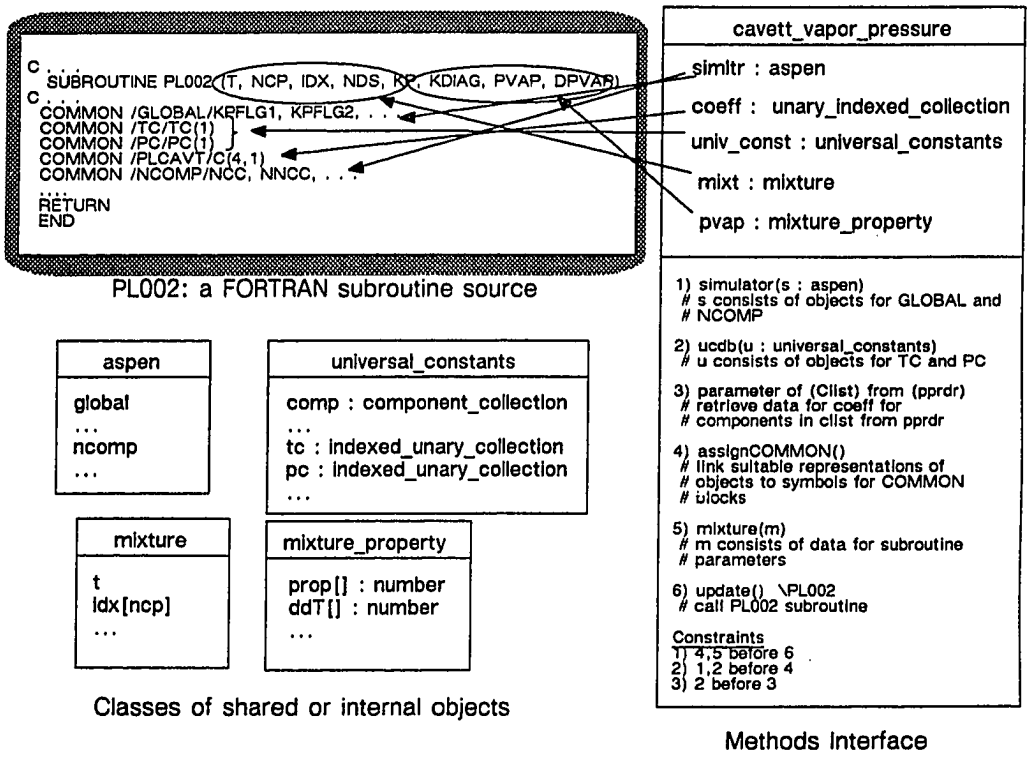


Figure 4.17 Application of the CODE Method to the Program Unit PL002

phase using the Cavett equation. The aggregate of subroutine parameters T, IDX, NCP, etc., of PL002 is modeled by the classes *mixture* and *mixture_property*, the former for input parameters and the latter for output parameters. Definition of this class is easily derived from data specifications, type and size, that are readily available in the source code or program documentation. As shown in Figure 4.17, six methods are created to model events concerning the PL002 program:

1. *simulator*, to assign an instance of class *aspen* that consists of objects for the COMMON blocks GLOBAL and NCOMP;

2. *ucdb*, to assign an instance of class *universal_constants* that consists of objects for the COMMON blocks TC and PC;

3. *parametersoffrom*, to provide an object to retrieve data for the attribute

object *coeff* from a simple databank file or PP database

4. *assignCOMMON*, to assign data--from objects held with the attributes *global*, *ncomp*, *tc*, *pc* and *coeff*--to the symbols in the directly reused code for the COMMON blocks GLOBAL, NCOMP, TC, PC, and PLCAVT, respectively;

5. *mixture*, to assign data to attributes associated with the subroutine parameters; and

6. *update*, in which the subroutine PL002 is called to update objects.

In addition, constraints on the order of executing these methods are specified in the method interface. Other program units, PL001 to IDLGAS in Table 4.2, are similarly processed. The classes *mixture_property* and *mixture* are used to represent the subroutine parameters in each case.

SORC: CLMON1, DFTMON

The compiled form of these program units is not directly reused; it is more advantageous to give an equivalent object-oriented program. First CLMON1, then DFTMON is covered. The subroutine CLMON1 consists of only one case statement (in its executable part). The segment of its source program for each case block consists of some local computations in addition to calls to other subprograms; thus, one requires new classes for the case blocks. As illustrated for CLMON1 in Figure 4.18, each case block of the case statement is associated with a new class that is named after the thermodynamic derivation it represents. The case statement itself is associated with an instance *APmethod_colln_1* of the class *indexed_collection*, and it consists of a collection of instances of classes for different case blocks. The subroutines CLMON2 and CLMON3 are similar to CLMON1, so are reused similarly. To simplify further, the objects *APmethod_colln_1*, *APmethod_colln_2*, and *APmethod_colln_3* for CLMON1, CLMON2 and CLMON3 are replaced by a union object *APmethod_colln*.

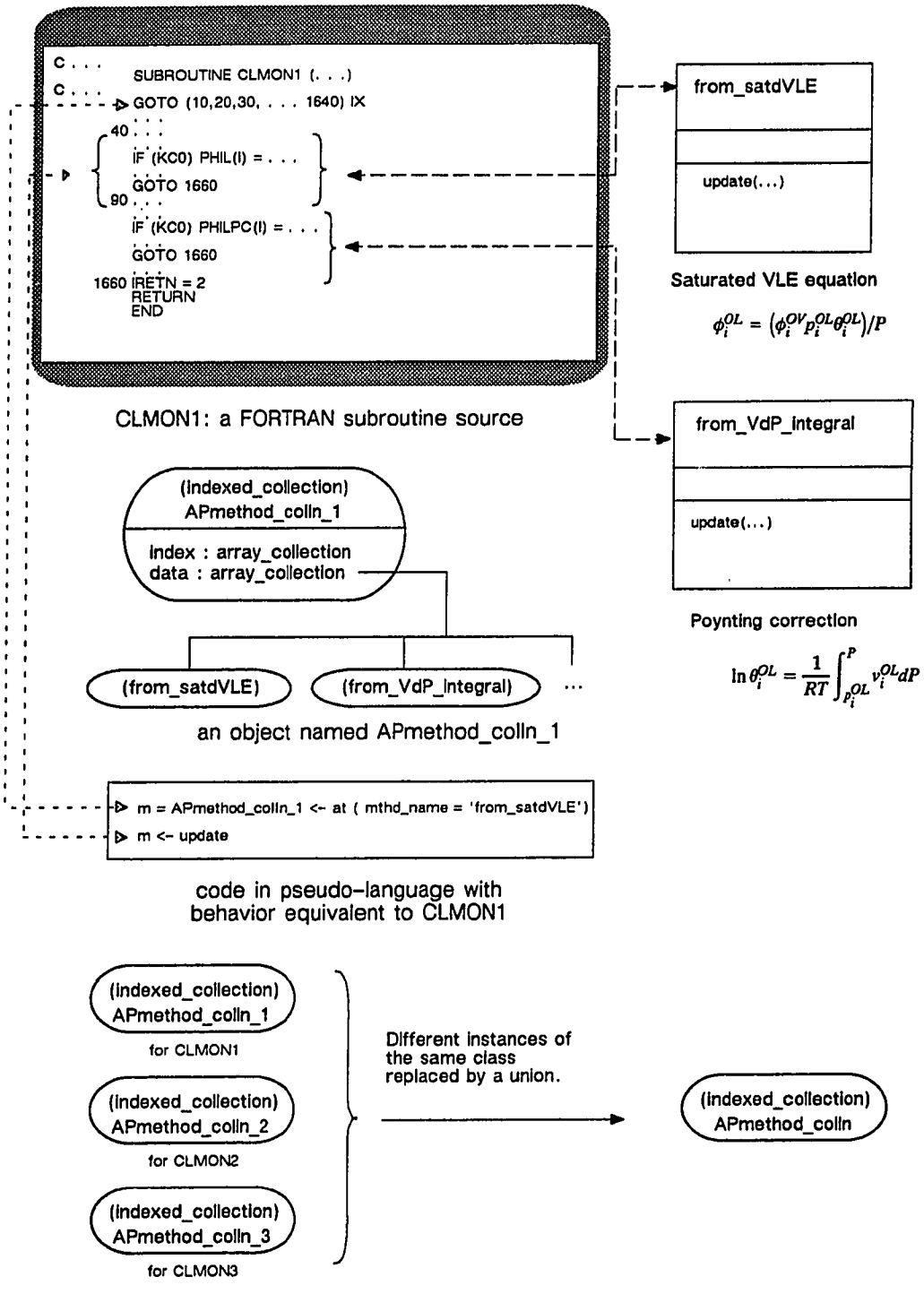
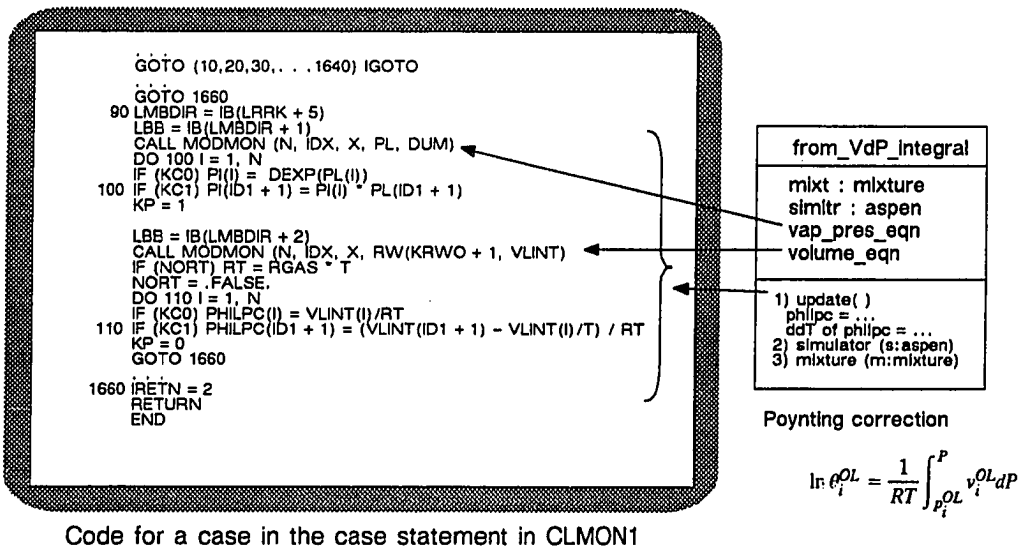


Figure 4.18 Application of the SORC Method to the Program Units CLMON1, CLMON2 and CLMON3

The definition of the classes associated with different case blocks is complicated. Presently, there is no known procedure to translate source code in traditional imperative language into an object–oriented modeling or programming language. Although, some of the following associations can be considered. It is clear that every subroutine call may be associated with an attribute referencing an object associated with the subroutine parameters that are updated. As an example, consider the case block for computing the Poynting correction of liquid phase fugacity (see Figure 4.19) and its associated class *from_VdP_integral*. The source segment consists of two subroutine calls to calculate data named PL and VLINT. Thus, two attributes are created to reference objects representing these data. The two attributes could have been proposed based on the equation for the Poynting correction; the equation can be viewed as consisting of two terms, the vapor pressure of liquid and the integral of molar volume with respect to pressure. (Note, however, that this involves knowledge which is not implicit in the source program, and thus requires human intervention.) In addition, the case block contains references to data in the COMMON blocks PPGLOB, RGLOB, and GLOBAL that already have representations in the REO–TGS model. Thus, attributes should be added to the class *from_VdP_integral* to hold instances of *ppglob*, *rglob*, and *global*. Instead, the attribute *simltr* is added to hold an instance of *aspen* that aggregates *ppglob*, *rglob*, and *global*. Similarly, the attribute *mixt* is added to hold an instance of *mixture* since the source segment contains references to the data already represented in REO–TGS by *mixture*. As regards operations on this class, methods are created to attach instances of *aspen* and *mixture*. The source segment of this case block is also represented by the method *update*, as specified in Figure 4.19, the eventual effect of which is equivalent to updating PHILPC. In this fashion, each case block in a case statement of CLMON1, CLMON2 and CLMON3 is modeled.

A complete list of such classes, along with a typical thermodynamic equation, is shown in Table 4.5. Each class in this table represents an equation, of thermodynamics or transport, parameterized over the physical property computed. Parameterization, of course, reduces the number of classes one would require; for example, the class *from_mixing* can be easily used in creating a suitable instance object for the calculation of enthalpy of liquid phase, h^L, besides free energy of liquid phase, g^L, as shown in Table 4.5.



Code for a case in the case statement in CLMON1

Figure 4.19 Application of the SORC Method to Segments of the Program Unit CLMON1

The subroutine DFTMON is structured similar to CLMON1 discussed above; thus, it is modeled in the same manner as DFTMON by creating the class *prop_data_colln*. Furthermore, as shown in Figure 4.20, the equivalent classes associated with different case blocks such as *phivmx*, *phiv*, and *philmx* are replaced by the structurally equivalent class *prop_data*. The attribute *index* of the class *prop_data_colln* collects property names that correspond to the statement labels for

Table 4.5 Classes Dervied from the Program Units CLMON1, CLMON2, and CLMON3

No.	Class†	A Typical Thermodynamic Equation	Attributes for other AP models and AP methods
1	from_VLE	$\phi_i^L = \gamma_i \phi_i^{OL} \theta_i^E$	gamma_eqn phil_eqn gamma_pc_eqn
2	from_VdP_integral	$\ln \theta_i^{OL} = \frac{1}{RT}\int_{p_i^{OL}}^{P} v_i^{OL} dP$	vol_eqn vap_pres_eqn
3	from_acitivity	$h^{EL} = -RT^2 \sum_i x_i \frac{\partial \ln \gamma_i}{\partial T}$	activity_coeff_eqn
4	from_fugacity	$\Delta h^V = -RT^2 \sum_i x_i \frac{\partial \ln \phi_i^V}{\partial T}$	fugacity_coeff_eqn
5	from_idealgas_departure	$s_i^{OV} = s_i^{OIG} + \Delta s_i^{OV}$	idealgas_eqn departure_eqn
6	from_maxwell	$s_i^{OV} = \frac{1}{T}\left(h_i^{OV} - g_i^{OV}\right)$	h_eqn g_eqn
7	from_mixing	$g^L = \sum_i x_i g_i^{OL} + RT \sum_i x_i \ln x_i + g^{EL}$	component_eqn excess_eqn
8	from_satdVLE	$\phi_i^{OL} = \frac{\phi_i^{OV} p_i^{OL} \theta_i^{OL}}{P}$	phiv_eqn vap_pres_eqn phi_pc_eqn
9	from_vapor	$\Delta h_i^{OL} = \Delta h_i^{OV}\left(T, p_i^{OL}\right) - \Delta h_i^{OVAP}(T) + \Delta_h^{OL}(T,P)$	dep_vapor_eqn vaporization_eqn press_corr_eqn

† The names of these classes begin with "from_" to indicate that they represent thermodynamic derivations.

different case blocks. The case blocks consist of mutually exclusive segments (there is no control dependency between them), each consisting of an assignment statement to update the value of a temperature derivative of the corresponding PP; the first is for the physical property itself, the second is for its first-order temperature derivative, and the third is for its second-order temperature derivative. Thus, the attribute *prop* for *prop_data* can be decomposed into three distinct attributes: *prop* for the physical property, *ddT* for its first-order temperature derivative, and *d2dT2* for its second-order derivative.

As regards dynamics, the source program statements for each case consists of assigning default zero values to different temperature derivatives. This is modeled by the methods *default*, *default ddT*, *default d2dT2* or *defaultall* to set the attributes *prop*, *ddT*, or *d2dT2* to default values. The complete source program is then associated with two message expressions: (1) one containing the message *at* to choose an object from the collection of *prop_data* instances; and (2) the messages *default*, *default ddT*, *default d2dT2* or *defaultall* for the update of the chosen object.

DOCU: CALMON, THERMO

In this case, the first step is to identify one or more classes consisting of attributes associated with the data that are either referenced or updated, and the second step is to specify methods that model the effects, as described in the manual or in a high-level requirements specification language, of the program unit. First CALMON is covered, then THERMO.

The description in the manual states that the function of CALMON is to compute various PP's for a given option set. Since CALMON references data in a particular option set, it is associated with the new method *update2* for the class *option_set*.

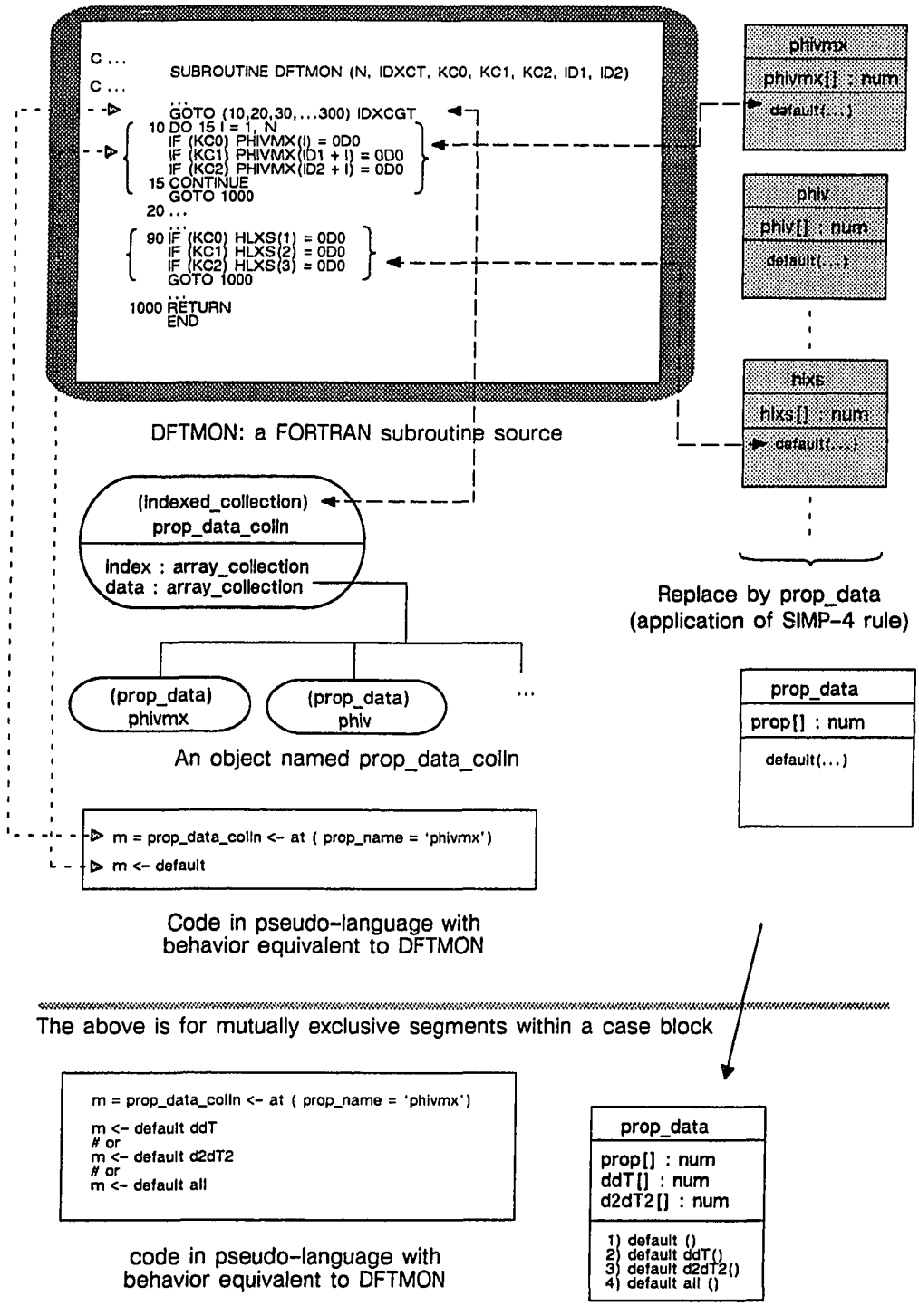


Figure 4.20 Application of the SORC Method to the Program Unit DFTMON

The procedure of this program unit as described in the manual consists of: retrieving the statement labels of the case blocks--of the case statements of the subprograms CLMON1, CLMON2, CLMON3 and DFTMON for the given PP's to be computed and the option set.

These statement labels are stored in the entries, one for each PP, in the resolved route bead. (A resolved route bead consists of control information that is computed from an option set. There is one such bead for every combination of the three data items: the physical property, codes for the temperature derivatives and integrals over pressure of the physical property, and the option set.) In terms of the object-oriented model developed so far, these statement labels are associated with names of the classes which in turn are associated with the case blocks of the case statement in the subroutines CLMON1, CLMON2, CLMON3 and DFTMON. Thus, the procedure for the method *update2* consists of: retrieving the names of the specified class and its instance in the object *prop_route_colln*, for the given PP's to be computed and the option set.

The name of the class for computing a physical property (under a particular option set) is specified in the class *prop_route*; accessing this information violates the principle of information hiding. An alternative procedure for the method *update2* is to leave the responsibility of retrieving the instance of the required class, from the *prop_route_colln* object, with the class *prop_route* (see Figure 4.21). Furthermore, to hold the objects retrieved from *prop_route_colln*, *prop_route* is assigned the new attribute *mdl* and the new methods *modelsource* and *update*. Thus, the procedure for *update2* in *option_set* consists of: retrieving an instance of *prop_route* and in turn requesting an *update* based on information that is internal to the retrieved *prop_route* instance, for the given PP's to be computed and the

option set. The method *update* can also be recursively structured, since the class *prop_route* itself is recursively structured.

In searching for instances of *prop_route* in *option_set*, only a limited number can be found in the attribute *major_prop_route*. However, recall that instances of *prop_route* form a directed acyclic graph (see the model in Figure 4.13 on page 78). Thus, one should examine all instances of *prop_route* that are reachable from *major_prop_route* in *option_set*. Recursive search is avoided by adding the new attribute *prop_route_colln* which represents all the reachable instances of *prop_route* from an instance of class *option_set*. This new attribute holds a collection from which an instance of *prop_route* is retrieved through an index of PP's.

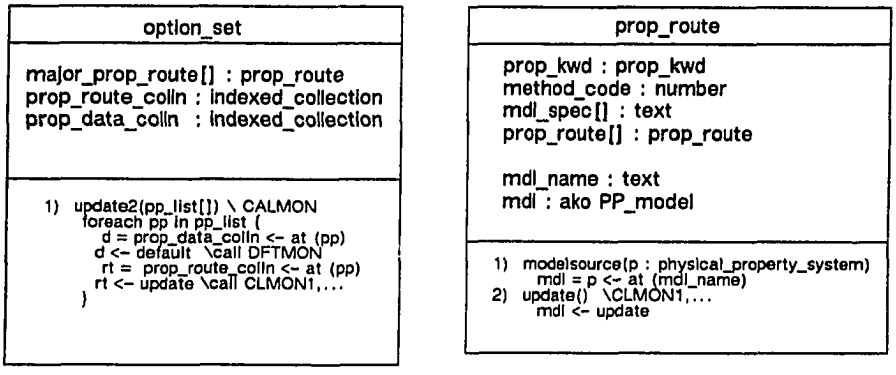


Figure 4.21 The Result of Applying the DOCU Method to the Program Unit CALMON

The subroutine THERMO, according to the manual, references data for option sets. This is also verified by a visual scan of the source code. Thus, THERMO is associated with the method *update3* of the class *option_set*.

The main function of THERMO, according to the manual, consists of route resolution and overall calculation control. The route resolution, for a given

option set and codes of various temperature derivatives to be computed for PP's, results in construction of resolved route beads. The overall calculation control simply passes the execution control to the subroutines IGMON, EOSMON and CALMON that carry out the computation according to the information in the resolved route beads. The called subroutines, already covered in REO–TGS, do not require any resolved route bead. In fact, route resolution is irrelevant to logical data modeling because it is designed to provide control information only to improve the performance; the idea underlying route resolution is to collect multiple expected calls to the same subroutine into a single call. Thus, the route resolution step is discarded, and the method *update3* consists only of message expressions that correspond to the subroutine calls IGMON, EOSMON and CALMON (see Figure 4.22).

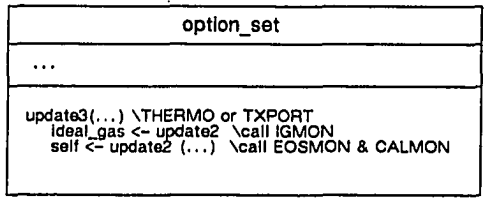


Figure 4.22 The Result of Applying the DOCU Method to the Program Unit THERMO

4.6 Design and Implementation in VSM

Icape–91 is designed and implemented, based on the REO–TGS model, in VSM. In this section, some aspects of this design are discussed, but first a short introduction to VSM is in order. (More details on a VSM design are given in Appendix B.)

VSM provides two main kinds of objects [Yamashita, 1987]: a vsm module, as it is called, to interface with a foreign module; and a Generic Data Structure (GDS) for object-oriented programming. In terms of REO, the *vsm module* is a construct to interface with the foreign module for virgin reuse or direct reuse. FORTRAN subroutines are included by interface procedures (for which C is the recommended language in version 1.1 of VSM), and interface procedures are catalogued as vsm modules in lieu of the FORTRAN subroutine itself. Some pre-processing of foreign modules is required for the dynamic linker-loader of VSM. A vsm module is developed in three steps: first, the vsm module is implemented; second, the vsm module and related object codes are catalogued or described in a catalog file; and finally, the catalog file is processed to create an overlay file that can be dynamically loaded into the system. For a highly flexible and dynamic system, VSM provides a construct that allows one to assign dynamically, during run-time, new data values to various symbols in the foreign module. With dynamic symbols, one can overcome the constraint of static allocation of COMMON blocks in FORTRAN. The use of dynamic symbols involves replacing the ordinary, static symbols (symbols that are resolved by static linking will be referred to as static symbols) declared as "external" (that is, defined elsewhere) with "dynamic symbols," thereby permitting the dynamic linker-loader to take appropriate steps. (The symbols that are not declared as external may be replaced by simple editing tricks with new external symbols.) The programming involves two steps: first, the declaration of dynamic symbols; and second, the redirection of the original static symbols to these dynamic symbols.

The concept of *GDS* is similar to that of class in object-oriented languages: Analogous to class, GDS has instances, instance variables, class variables, instance methods, and class methods. (Many other capabilities of GDS are not considered

in the Proto–ICAPE Project.) While naming various objects, "domain–friendly" and non–cryptic names are preferred over those that are adopted from the system that is integrated. For example, a preferred name for the GDS for the class *component* is *chemical_component*; the previous name is too general compared to the new one. Similarly, the preferred names for GDS's that represent the classes *global* and *rglob* are *integer_global_aspen* and *real_global_aspen*, respectively.

Vsm modules are designed for the directly reused program units to which the CODE method is applied, and GDS's are designed for all classes in the REO–TGS model. In this section, one example including the design of both vsm modules and GDS's is described. The example chosen is the subroutine PL001 for calculating vapor pressures of components in a liquid phase through the Extended Antoine model (see Figure 4.23). The vsm module for interfacing with PL001 is called XANT. It is developed in three steps: first, an implementation of the vsm module XANT is developed in C; second, this and related object codes are cataloged in the XANT.cat file; and third, the catalog file is processed into an overlay file XANT.ov for dynamic loading. For a highly flexible system, it is desirable to be able to assign new data values to the (statically sized) COMMON blocks GLOBAL, NCOMP, and PLXANT in the PL001 program unit. Thus, these symbols are replaced by dynamic symbols in two steps (see Figure 4.23): first, declaration of the dynamic symbols *XANT_global*, *XANT_ncomp*, and *XANT_coeff*; and second, redirection of the original, static symbols to the dynamic symbols.

A GDS design for the class *extended_antoine* and its superclass *PP_model* is shown in Figure 4.24. The GDS *PP_model* definition contains slots in the instance block for each of the attributes *simltr*, *ucdb*, and *mixt*. For generality's sake, the type of instance variables is set to the type *ptr* (this holds a reference to an object in the object table, and is not the same as the address pointer of C programming

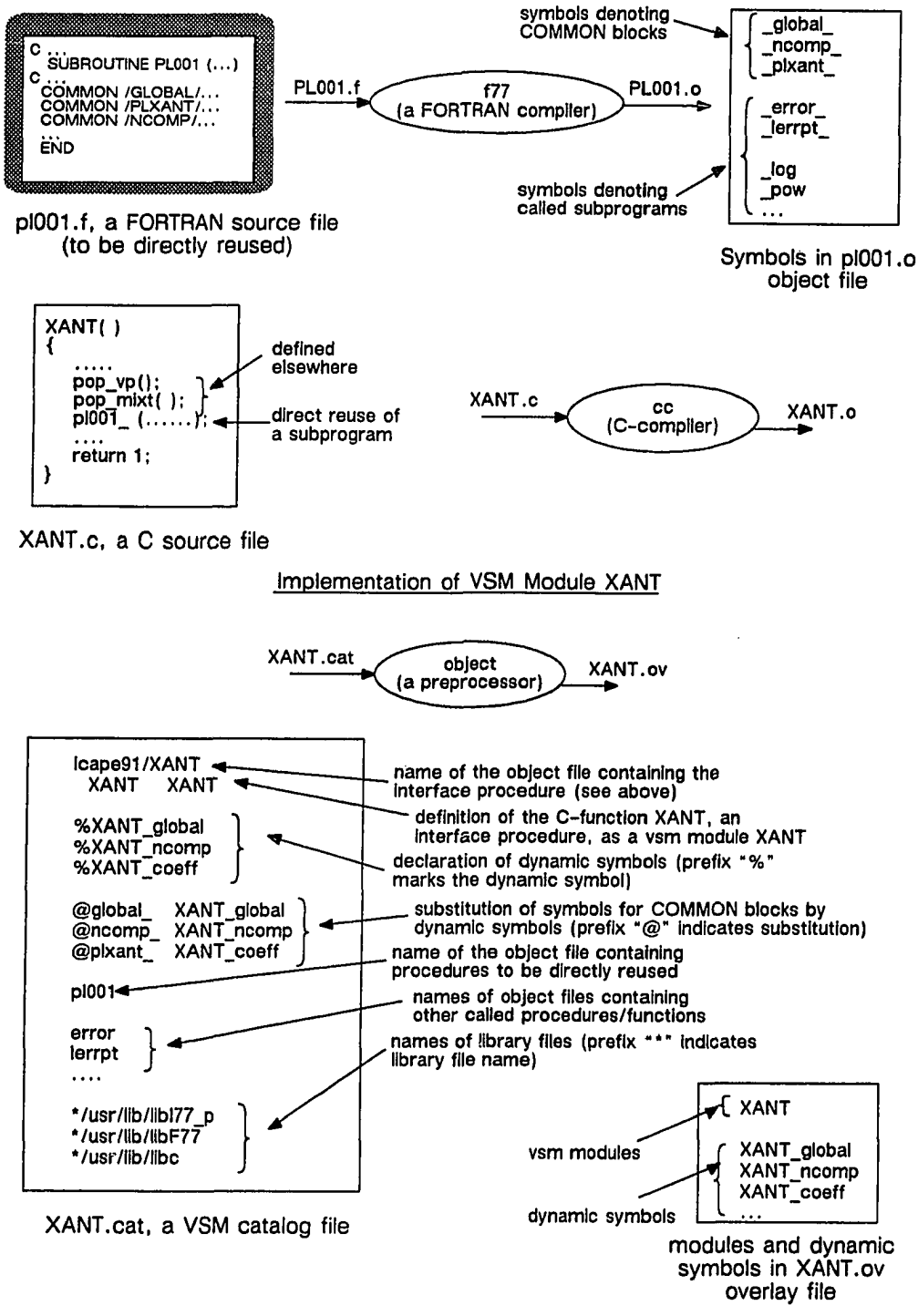
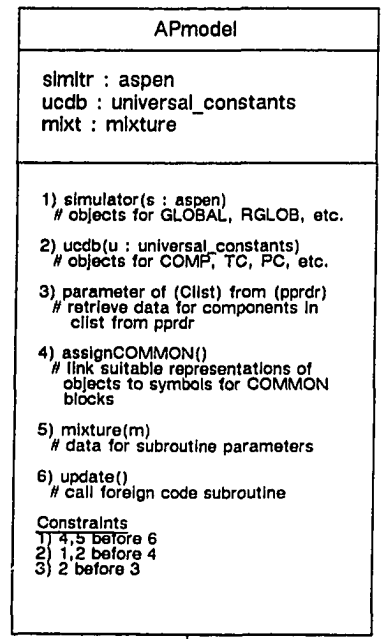


Figure 4.23 A VSM Module for lcape-91



Parts of REO-TGS model

```
GDS physical_property_model
INSTANCE {
  ptr      simitr
  ptr      ucdb
  ptr      mixt
}
METHOD {
 simulator (s : aspen)
 {
   simitr = s <- address
 }
 pp database (d : universal_constants)
 {
    ucdb = d <- address
 }
 parameters of (comps) from (pprdr)
 {
    # do nothing
 }
 assignCOMMON ( )
 {
    # do nothing
 }
 mixture (m : mixture)
 {
    mixt = m <- address
 }
 update ( )
 {
    # do nothing, subclass responsibility
 }
}
END
```

```
GDS extended_antoine IS physical_property_model
INSTANCE {
  ptr     coeff
  ptr     pvap
}
METHOD {
 pp database (d : universal_constants)
 {
    self <- super  pp database (d)
    coeff <- indexed by (d <- components)
 }
 parameters of (comps) from (pprdr)
 {
    x = pprdr <- get ('extended antoine') ('coeff') of (comps)
     foreach e in x {
       coeff <- at (e <- component) put (e <- coeff)
     }
 }
 assignCOMMON ( )
 {
     simitr.iglobal<- assignCOMMON ('XANT_iglobal')
     simitr.ncomp<- assignCOMMON ('XANT_ncomp')
     coeff          <- assignCOMMON ('XANT_coeff')
 }
 update ( )
 {
   pvap <- default all
   XANT mixt pvap  # calling PL001 subroutine
 }
}
END
```

GDS Designs for Classes in REO-TGS Model

Figure 4.24 Design of a GDS for Icape-91

language). The interface specifications of the related methods *simulator*, *ppdatabase*, and *mixture* adopt the required restrictions on the type of objects attached. In this "root" superclass, some methods such as *update* and *assignCOMMON* have no effect on its instance variables.

The GDS *extended_antoine* for the class *extended_antoine* inherits the structure and methods from the GDS *PP_model*. It has instance variables for each of the attributes *coeff* and *pvap* of the type *ptr*. The methods *pp database*, *assignCOMMON*, and *update* that affect objects held at the slots *coeff* and *pvap* are redefined or "refined" to supplement those inherited from the superclass. The method *pp database* attaches to the object held at *coeff*, an instance of *component_collection* from an instance of *universal_constants*. The method *assignCOMMON* attaches to the dynamic symbols a contiguous memory representation of the data in the objects associated with the COMMON blocks. The method *update* first updates the data objects that are passed to the vsm module *XANT* with default values, and then calls the vsm module *XANT* with the given *mixt* object to update the *pvap* object; the second step is similar to calling a subroutine in FORTRAN.

4.7 Summary

In this chapter, an object-oriented model called REO-TGS is derived first from the input language descriptions and then from program descriptions of the TGS subsystem of ASPEN. The design and implementation of a prototypical ICAPE system, Icape-91, based on REO-TGS are briefly discussed. With the "concept to demo" method of research, the Proto-ICAPE Project has demonstrated an entirely new approach, based on software reuse, to object-oriented modeling for integration in ICAPE.

5. Conclusions

ICAPE, ICAE, object-oriented programming and software reuse are promising new fields of research. This final chapter presents the conclusions and contributions of this research, and makes suggestions for future work.

5.1 Conclusions

The integration in ICAPE involves integration of data and software. The integration of process engineering data is rather important. To this end, the first essential step is the development of a model for the management of process engineering data. The complex characteristics and requirements of engineering data can be modeled and met respectively with the help of object-oriented programming. However, the development of object-oriented models for large domains such as process engineering and allied areas is non-trivial; that it should be developed is easier said than done. One approach identified and developed in this research is to derive the model from the existing software rather than from scratch; the new model can be improvised subsequently.

The integration of software has certainly caught, perhaps in different disguises, the attention of many. Any approach that takes a black box view of tools to be integrated has many disadvantages, as argued in Section 2.3.2 and discussed in Section 4.4. The major disadvantage is that one is constrained by all the limitations inherent in the design and implementation of tools to be integrated. Any approach that takes a glass box view of tools to be integrated also has many disadvantages such as inheriting a legacy of design decisions from the old. Mere mechanisms for virgin reuse of software components are not enough. (This research project was partly motivated by mechanisms proposed and implemented by

Yamashita [1987].) Some, often substantial, modifications of the tool may be required to achieve some of the simplest benefits of object-oriented programming.

As argued in this dissertation, the field of software reuse provides a more general, hence more powerful, framework for solving the above problems. Researchers in computer science take a linguistic approach to systems to solve any problem. For instance, computer scientists have developed data models, formal languages, and systems engineering to achieve data integration. There is no notion of "data integration" in computer science, so one should view it only as a desideratum, and not as a construct or basis for building scientific and technological knowledge. (The view in computer science seems to be that data integration is achieved by managing data in a database through a DBMS. It is the task of DBMS to coordinate the use of databases by many users.) Similarly, software reuse is a linguistic view of the problems of software integration, and hence is a framework more general than software integration. There is no notion of "software integration" in computer science, so it should be considered only as a desideratum of the field of ICAPE and ICAE. Another reason to adopt a software reuse perspective is that in this field knowledge is being continuously added that can be readily brought to bear on problems in ICAE.

A methodology such as REO has many benefits discussed and illustrated in the preceding chapters. An outstanding benefit is simplification: in the previous chapter, the final REO-TGS model did not carry many concepts--such as the categorization of the physical properties into major, intermediate, and subordinate properties--from ASPEN. REO can also readily accommodate black box approach to integration.

There are limitations of this research so far. One of these limitations is that there is no explicit mention of conditions under which the code reused in as-is

condition may fail in its task. Hence, the model of reused code does not include "operational" failures. In fact, this project excluded from its scope any consideration of error logs to which all program units write.

5.2 Contributions

The Proto-ICAPE Project has made the following contributions to the field of process engineering, ICAE, and software reuse:

1. This research presents a new analysis of the pressing problem of integration in ICAPE (see Section 2.1). The problem of integration involves three levels: application, data and software. The integration at the level of data is solved by a DBMS. The integration of software itself involves different domains of software engineering that are intermixed in particular software. This intermixing is often hard-coded in any software for a lack of strict discipline of modularity in constructs employed or on the part of the designer. It is important in software integration to identify these layers and handle each independently, perhaps discarding some.

2. Most importantly, this research provides an alternative to the two rather dogmatically followed principles, the black box principle (thou shalt not open the program), and the non-modification principle of the glass box approaches (thou shalt not alter the program) that are implicit in all previous attempts at ICAE in different disciplines. The limitations of approaches based on these principles are identified with respect to the goal of developing ICAPE systems that are object-oriented.

3. This research also develops a software reuse approach that fits nicely with the goals of ICAPE. It identifies two categories of software reuse techniques: virgin and processed. Techniques for virgin reuse are often necessary, but are *not*

sufficient to realize some "refinement" of the tool. This research develops new techniques of processed reuse of software components from and to different stages of the software development life cycle.

4. This research develops a systematic software reuse methodology called REO. If one follows the REO methodology, one can simultaneously achieve both the derivation of object-oriented models and the integration in ICAPE. The REO methodology is applicable to subjects of large scale, as demonstrated in this research. Presently, it covers two major software components of interest: program descriptions, and languages for program input and output.

5. This research develops a prototypical ICAPE system, Icape-91, that covers most parts of a subsystem of ASPEN--a decade-old batch system consisting of a quarter (1/4) million lines of FORTRAN code. Icape-91 includes an object-oriented model, relational data models and graphical user interfaces; the result of all this is an object-oriented system that is highly interactive for certain process engineering tasks that were facilitated by ASPEN.

5.3 Suggestions for Future Work

1. The success in developing Icape-91 should motivate coverage of other subsystems of ASPEN; at the present stage, the prototype covers only a subsystem of ASPEN.

2. A productive development environment, including various CASE tools to practise REO, would be extremely helpful. It is also important to make use of or develop extensive class libraries to improve a programmer's productivity.

3. The field of man-machine interface technologies is rather important for ICAPE and ICAE; in a typical project, this area consumes well over fifty percent

(50%) of development resources. This area should be examined in detail, especially as it relates to object-oriented programming.

4. ICAE projects can gain a lot from the field of software reuse as shown through this research. Researchers in organizations such as MCC, Austin, Texas, who are working in this area could provide valuable experience and tools. Collaborative projects with them should be undertaken by ICAE researchers.

5. One of the products of this research, object-oriented models, can be used in research and development projects in the application of object-oriented database technology, which is growing rapidly, to process engineering.

6. Most importantly, the success with the software reuse approach and the REO methodology in developing the Icape-91 system suggests that a large-scale ICAPE system, which would resemble the IPAD project in the aerospace industry, be undertaken. Such a project should cover various design programs, different process simulators and their families (such as steady state, dynamic), plant design systems, mathematical systems, etc. In fact, a need already exists for what is known as a multi-purpose simulator, a kind of concurrent process engineering tool, to support conceptual process design, detailed process design, control system design, operator training and even plant operations and maintenance.

Appendix A. Object-oriented Model for Icape-91

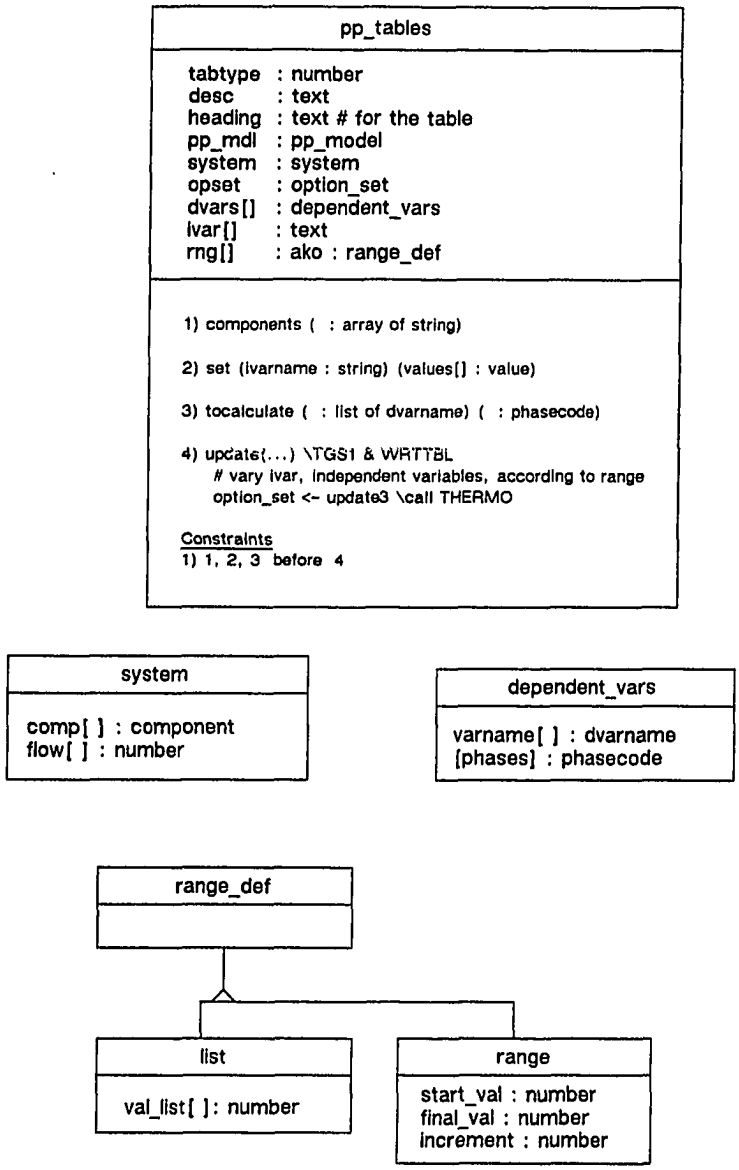
A "REOgenous" (that which is derived with the help of the REO methodology) object-oriented model, REO-TGS, is presented in the following pages. The model discussed in Chapter 4 is presented first in Figure A.1 to A.4. Next, the inter-class diagrams that show inheritance and part-of relationships between classes are presented.

The OMT notation is used in these diagrams. Each page shows only part of the whole diagram if the diagram is too large to fit on one page. The OMT notation is supplemented with the following new constructs:

1. The class that is shown in a dash box is not part of the model, but is shown to make the diagram "friendlier" for domain experts.

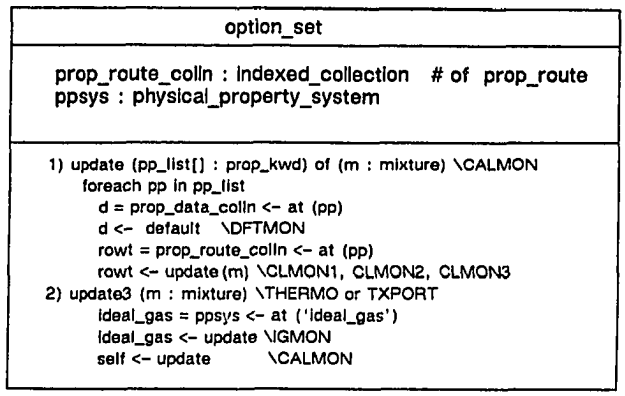
2. The sources of derivation of classes, methods, and messages are annotated with a backslash.

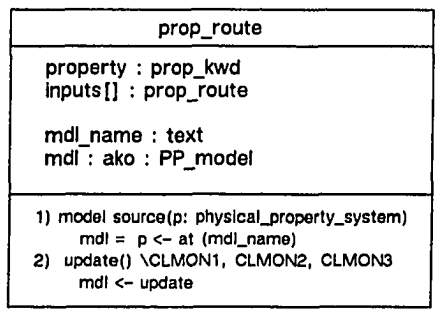
3. As a constituent of objects and relationships, one of the subclasses of a class is represented in a box that is named after the class but prefixed with "ako :" to denote "a kind of" constraint. There may be additional restrictions on the constituents, such as permitting only specific rather than all subclasses; such restrictions are not shown in these diagrams.



Note: The Figures A.1 to A.4 summarize Chapter 4.
This is the highest level of the REO-TGS diagram.

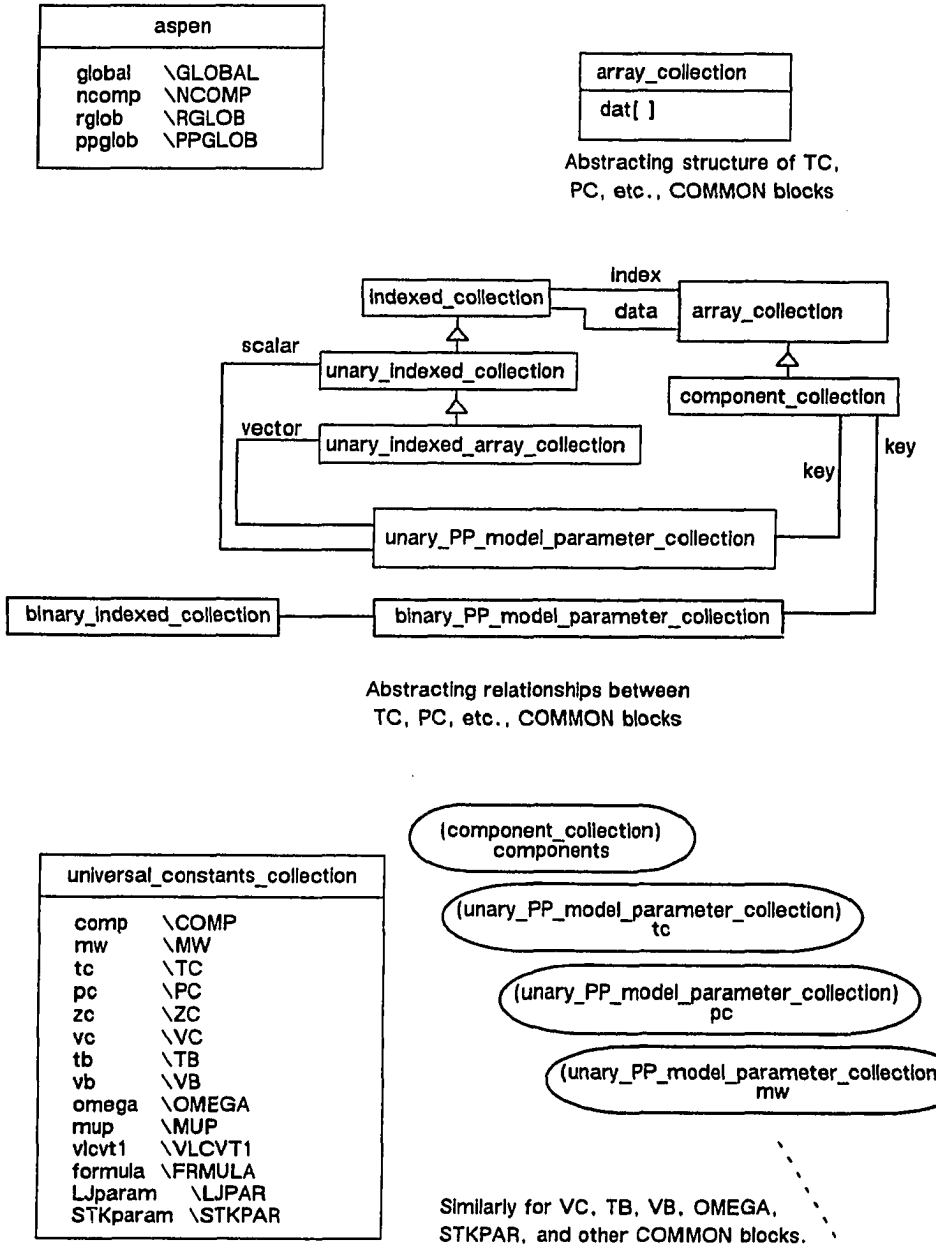
Figure A.1 Parts of REO-TGS for PP Tables





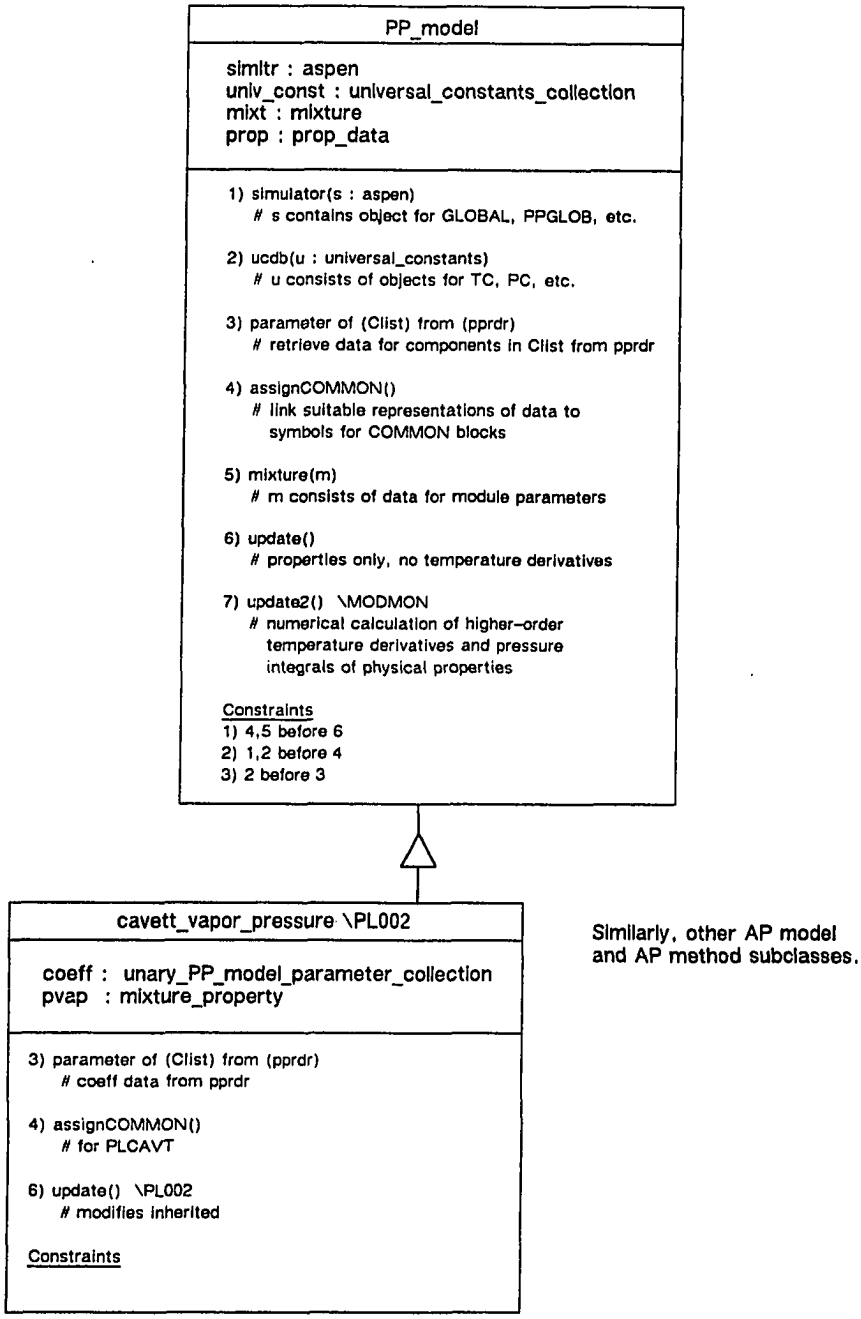
Note: The Figures A.1 to A.4 summarize Chapter 4.

Figure A.2 Parts of REO-TGS for Option Set



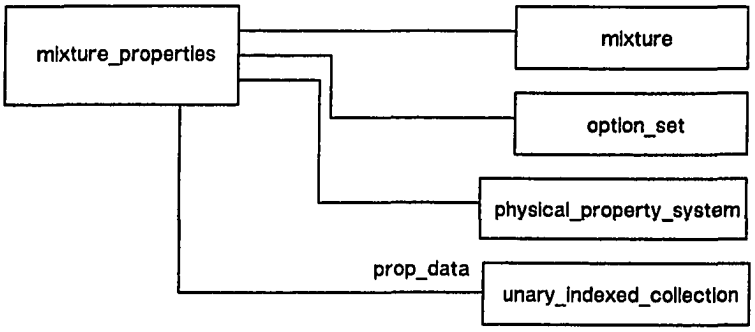
Note: The Figures A.1 to A.4 summarize Chapter 4.

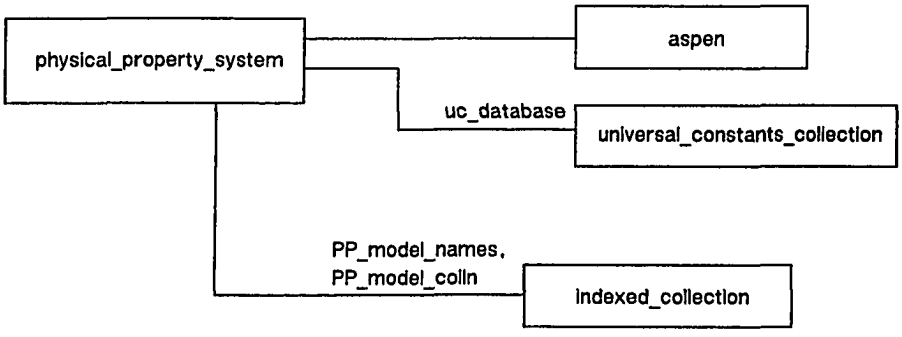
Figure A.3 Parts of REO-TGS for the COMMON Blocks



Note: The Figures A.1 to A.4 summarize Chapter 4

Figure A.4 Parts of REO-TGS for the PP Model Routines





Note: This is the highest level of REO-TGS Model.

Figure A.5 Inter-class Relationship Diagram for PP Tables

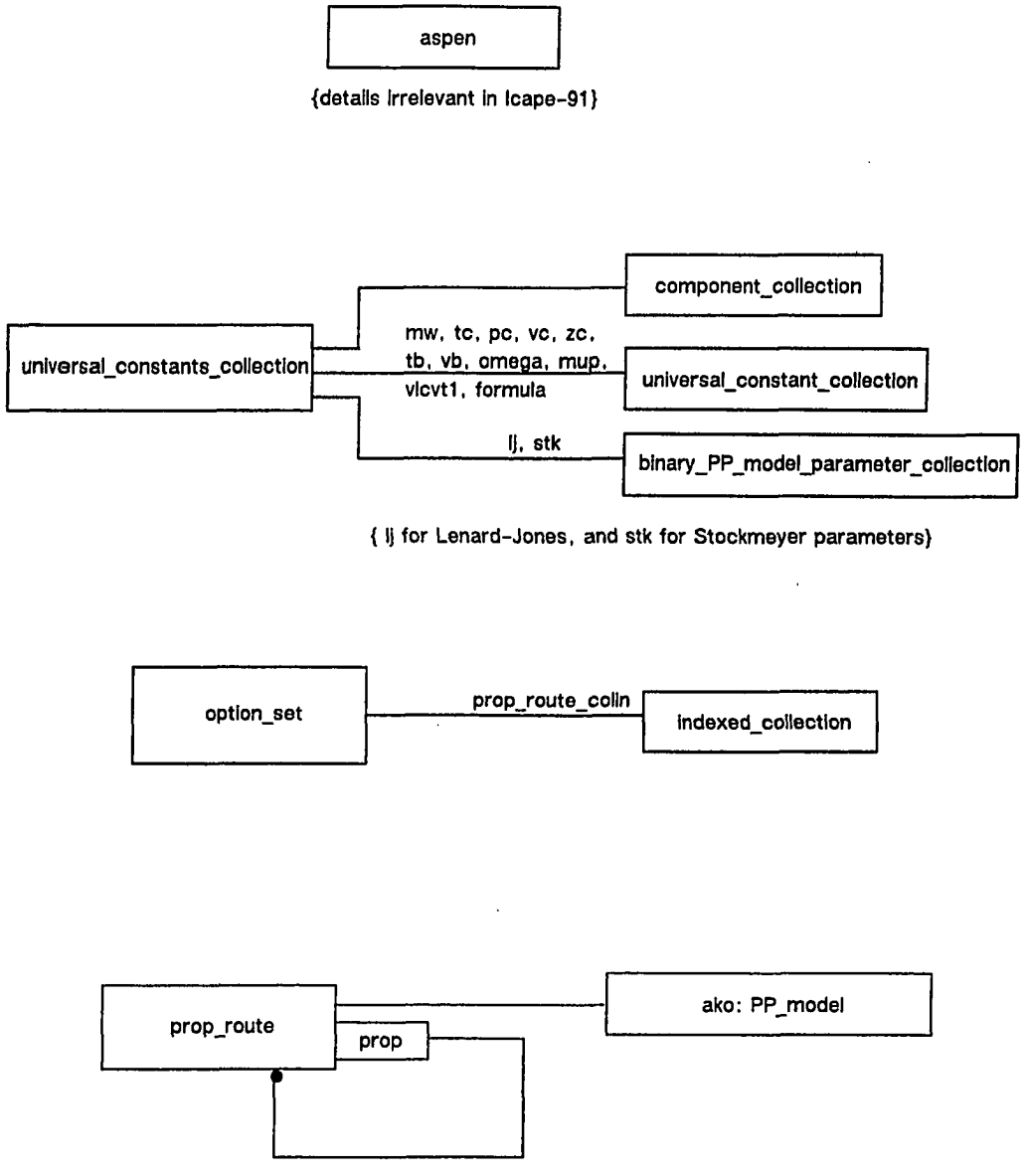


Figure A.6 Inter-class Relationship Diagram from the COMMON blocks

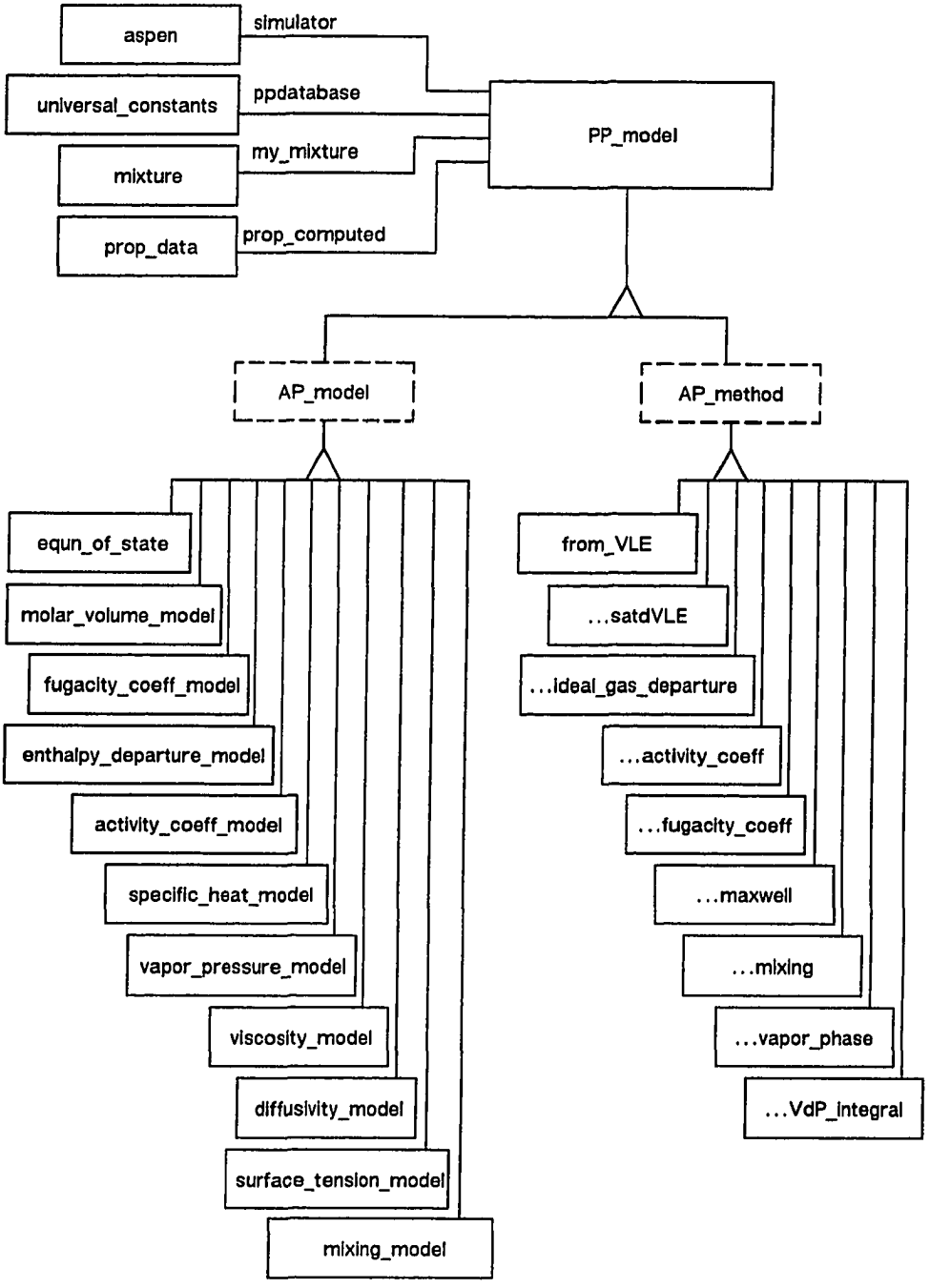


Figure A.7 Inter-class Relationship Diagram from the PP Model Routines

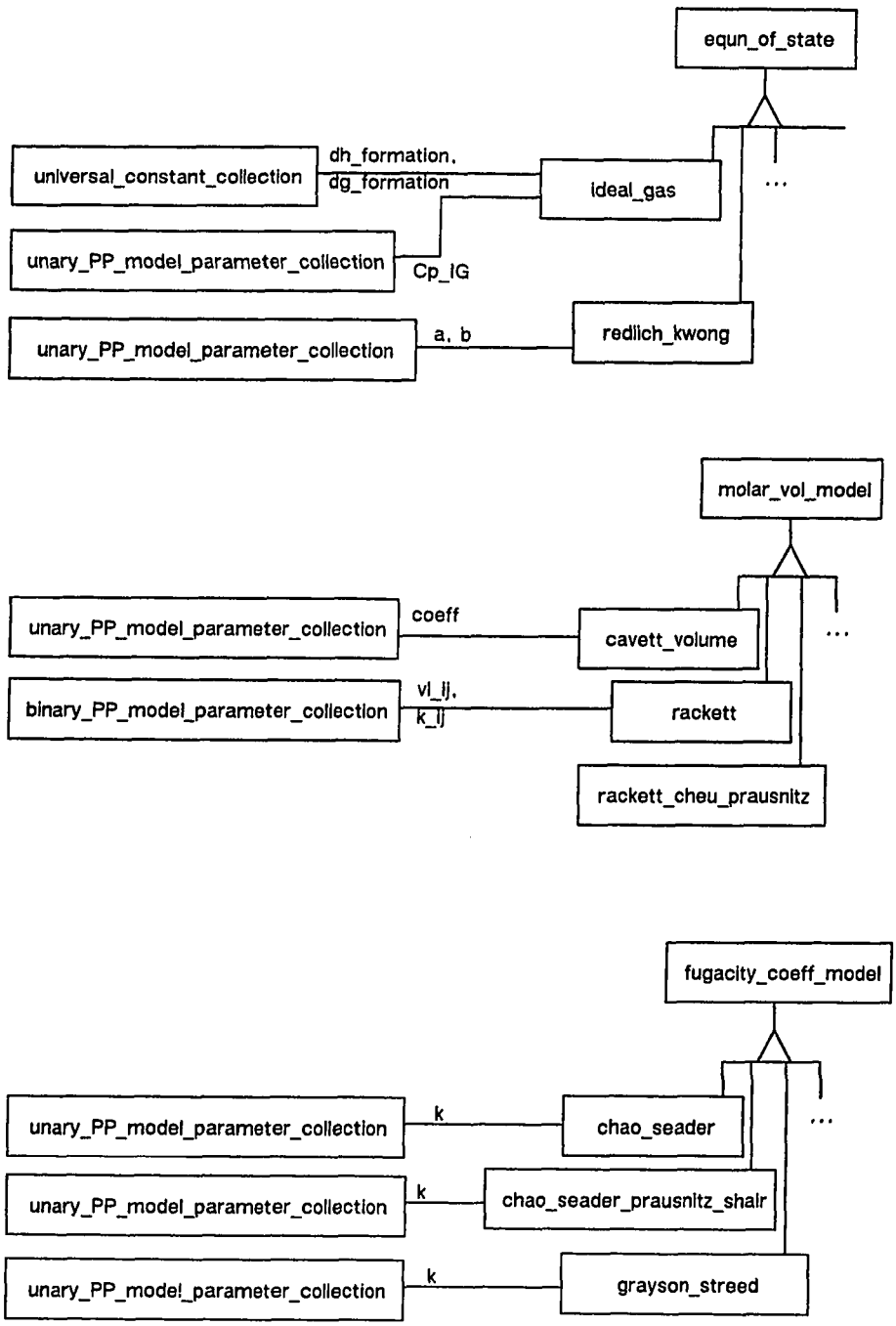


Figure A.8 Inter-class Relationship Diagram from Some of the AP Model Routines

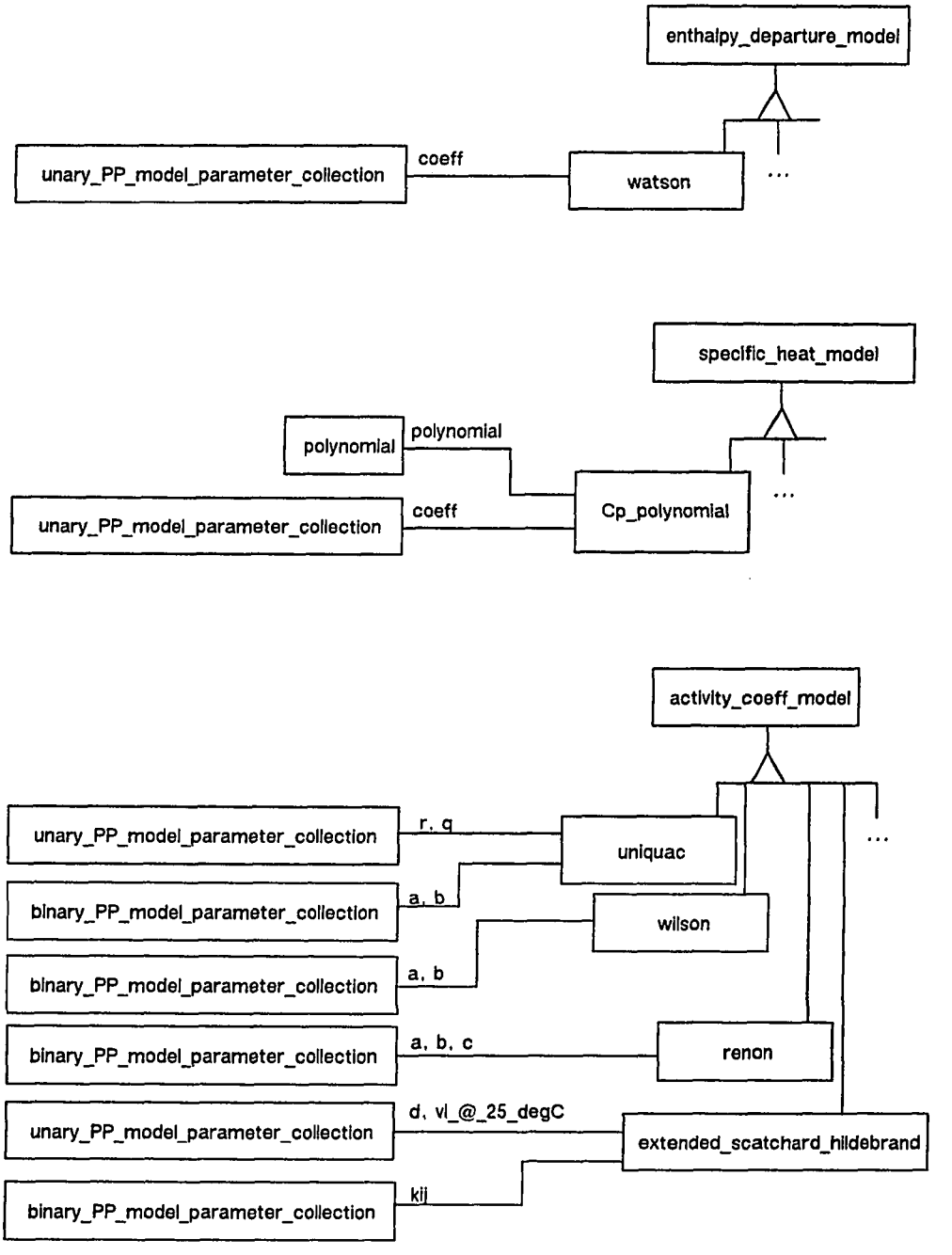


Figure A.9 Inter-class Relationship Diagram from Some of the AP Model Routines

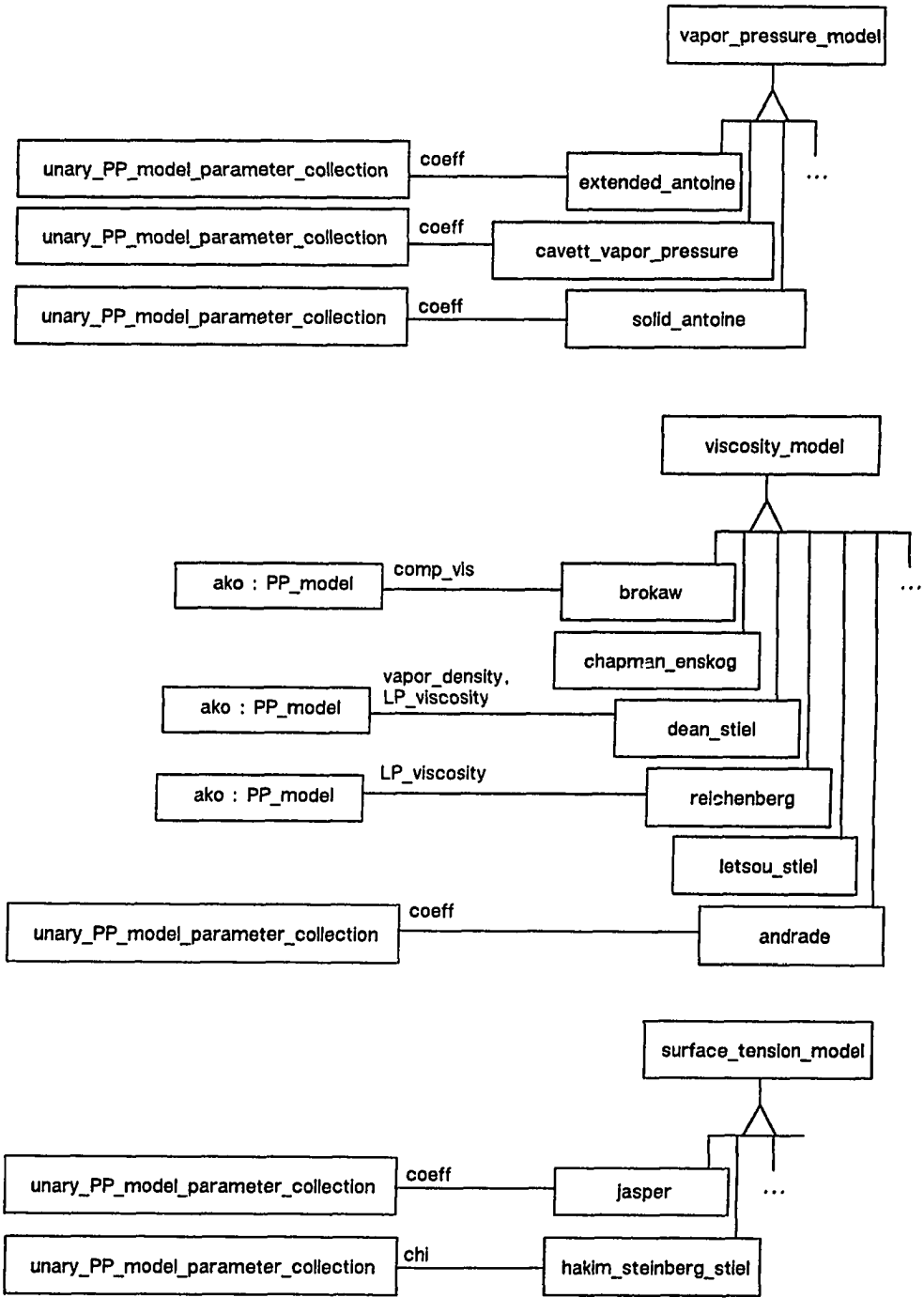


Figure A.10 Inter-class Relationship Diagram from Some of the AP Model Routines

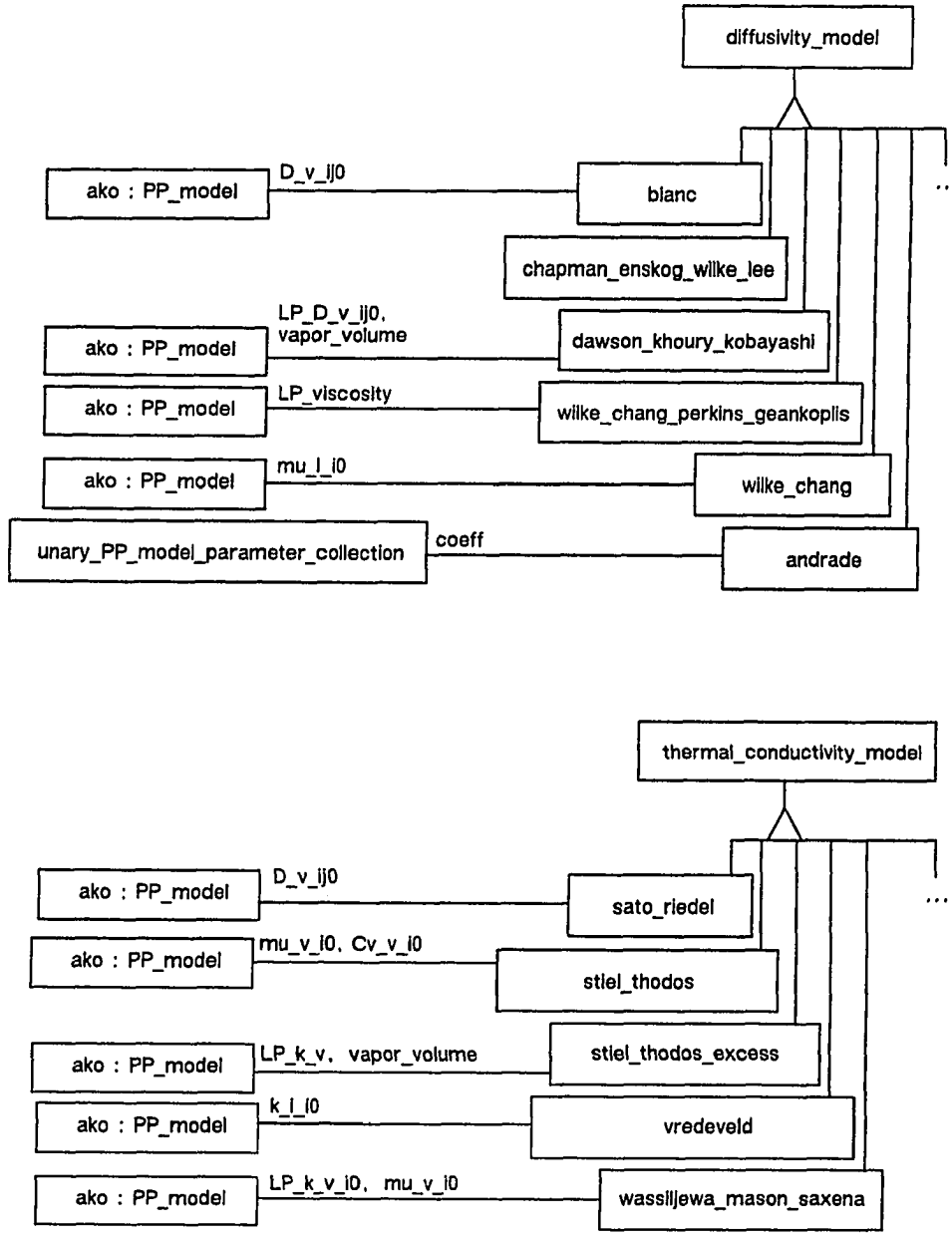


Figure A.11 Inter-class Relationship Diagram from Some of the AP Model Routines

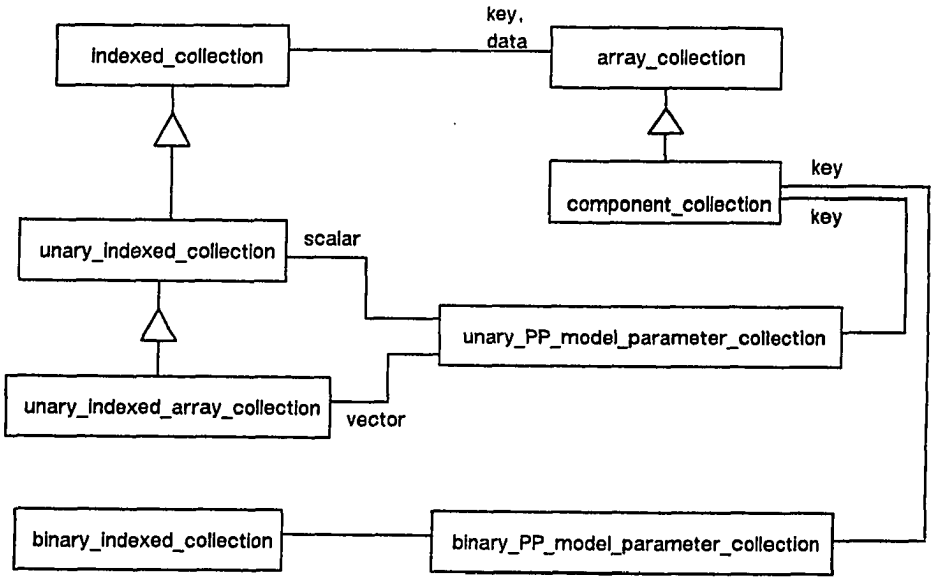


Figure A.12 Inter-class Relationship Diagram of Utility Objects from the COMMON Blocks

Appendix B. VSM Design of Icape-91

A design of Icape-91 in VSM consists of GDS's and vsm modules, as discussed in Section 4.5 and presented in the following pages. A GDS design describes the essential elements of its structure, elements and superclasses, and behavior, operations and constraints on operations that are "visible" to the user. Figure B.1 presents a template for design. The structural elements are specified along with their types. The operations or methods of a GDS are specified as a method signature consisting of a method name, argument types, and the order of arguments (argument names are only for internal use). Each method has a unique integer identifier (these are needed to show inter-relationships and different specializations of methods) in the family, class and all its subclasses. By default a subclass GDS inherits all methods from its superclass. However, it may either supplement (indicated by the sign "+"), modify (indicated by the sign "*"), or drop (indicated by the sign "x") the inherited method. For some methods, an informal description rather than detailed design is given. The constraints on methods are specified in terms of *before* and *after* relationships between methods or a collection of methods.

A vsm module design has three parts (recall discussions in Section 4.5): a foreign tool in as-is condition, an implementation of an interface module for VSM, and a definition of the interface module for the linker-loader in VSM. Figure B.15 presents a template for design. Of course, the implementation of an interface module is not an important element of the design; nonetheless, it is shown for completion but most of the details are omitted. In some cases, design description would have required more than afforded by one page; in these cases,

only portions of the design descriptions are shown. More details can be found at the Center for Computer Aided Process Engineering.

name of GDS
IS name of superclass (list, if multiple inheritance)
attributes type
No. & type) method signature (method name, argument types and order) where, type is one of the following: "+" if subclass is responsible for modification or addition to the method "x" if subclass drops inherited method "*" if subclass modifies inherited method else new method
Constraints no.) *before* or *after* relationship between methods or a set of methods

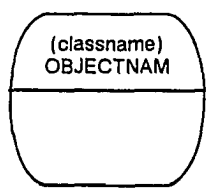


Figure B.1 Template for design of GDS on the following pages.

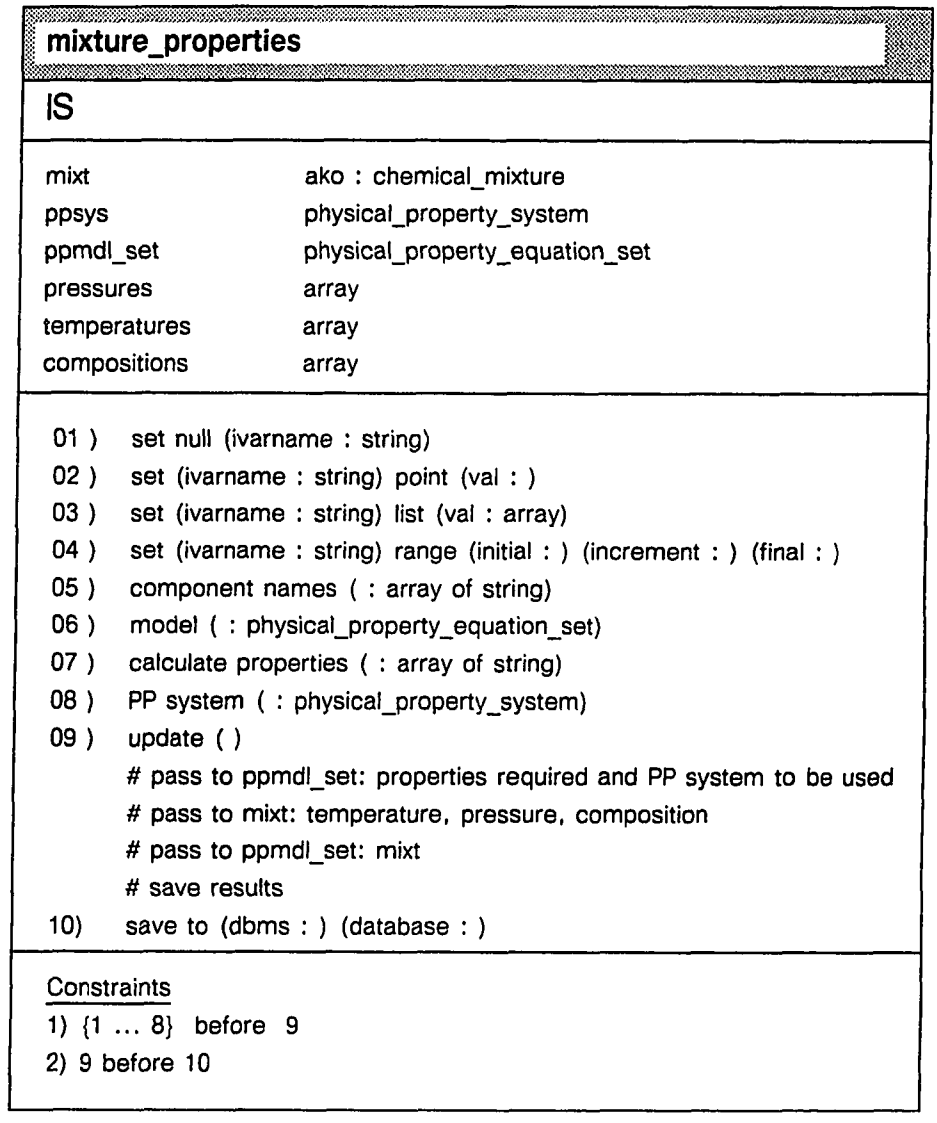


Figure B.2 Design of the GDS mixture_properties

physical_property_system

IS

simItr	aspen # or, other process simulator
ppmdl_colln	indexed_collection
ppdb_reader	pp_model_parameter_reader
ucdb	universal_constants_collection

01) PP model data reader (: pp_model_parameter_reader)
    # to retrieve data for ucdb and objects in ppmdl_colln
02) add components (: array of string)
    # update universal_constants_collection and objects in ppmdl_colln, for the given components
03) mixture of components (cs: array of string)
    # create an object for cs and that is compatible with PP model objects in ppmdl_colln
04) get model named (: string)
    # for the given GDS name retrieve its definition, instantiate it, and save it in ppmdl_colln
    # to this new object add universal_constants_collection, pp_model_parameter_reader, components, simulator
    # the new object should be able to execute its update method

Constraints
1) 1, 2 before 3

Figure B.3 Design of the GDS physical_property_system

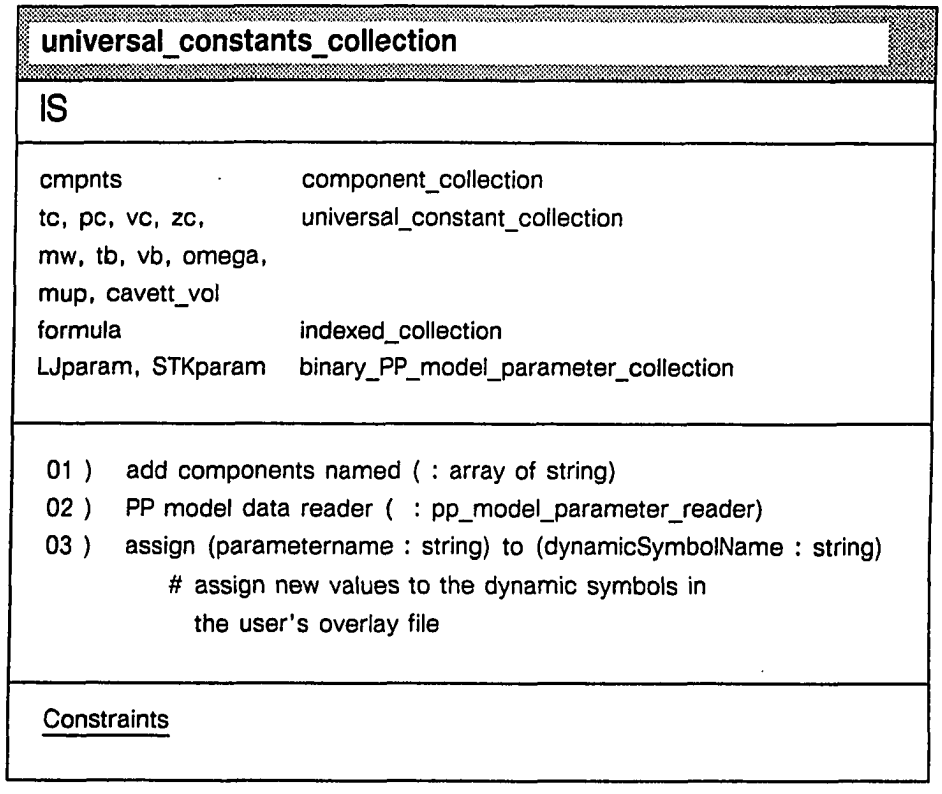


Figure B.4 Design of the GDS universal_constants_collection

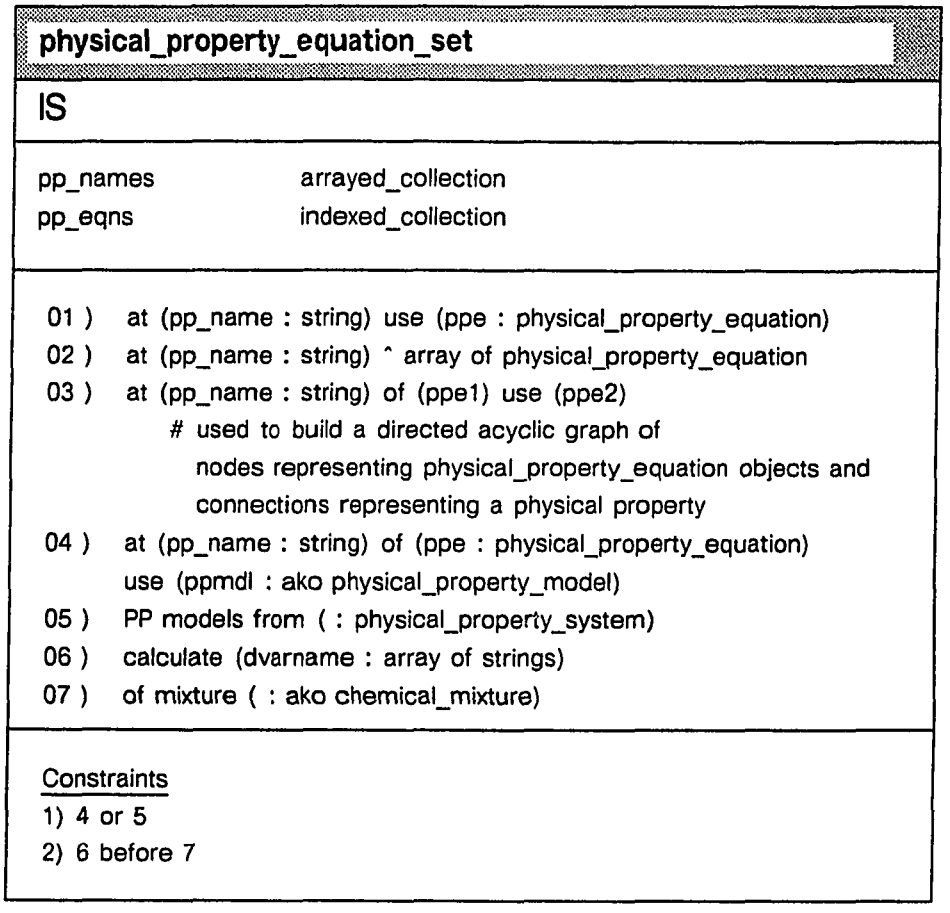


Figure B.5 Design of the GDS physical_property_equation_set

physical_property_equation

IS

prop_calculated	PP_name
ppmdl	ako : physical_property_model
input_ppe	array of physical_property_equation

```
01 )  to calculate ( : PP_name | string)
02 )  model ( : ako physical_property_model)
03 )  model name ( : string) # GDS/class name
04 )  at input ( : PP_name | string) use ( : physical_property_equation)
05 )  calculate ( : array of ddTcode)
          # propagate to input_ppe objects
          # link up ppmdl with those of input_ppe objects
          # request calculation from ppmdl
```

Constraints

1) 1 before all of {2, 3, 4, 5}

Figure B.6 Design of the GDS physical_property_equation

physical_property_model

IS

simltr	aspen
ppdb	universal_constants_collection
mixt	aspen_mixture
data	prop_data

```
01 )   simulator ( : aspen)
02 )   universal constants data ( : universal_constants_collection)
03 )   parameters of ( : array of components)
       from ( : pp_model_param_reader)
04 )   at ( submodelnm : string)
       use ( : ako physical_property_model)
05+)   assign dynamic
           # objects to COMMON blocks
06 )   mixture ( : aspen_mixture)
07+)   update ( : prop) ( : phase) ( : mixd) ( : array of ddTcode)
08 )   get ( : prop) ( : phase) ( : mixd) ( : array of ddTcode)
```

Constraints

1) 1, 2 before 4
2) 2 before 3
3) 5, 6 before 7

Figure B.7 Design of the GDS physical_property_model

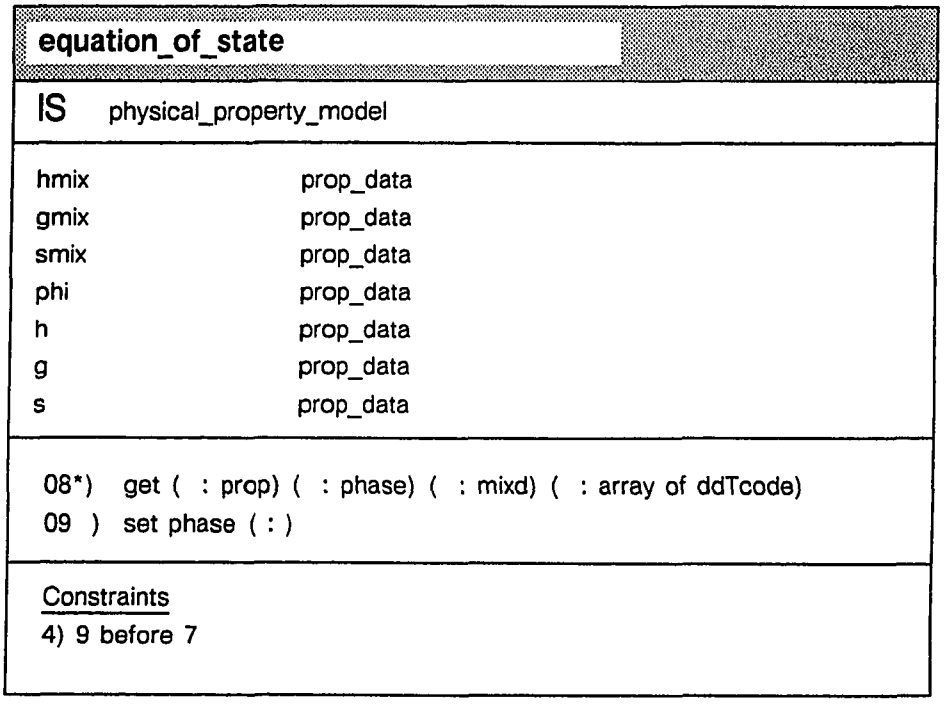


Figure B.8 Design of the GDS equation_of_state

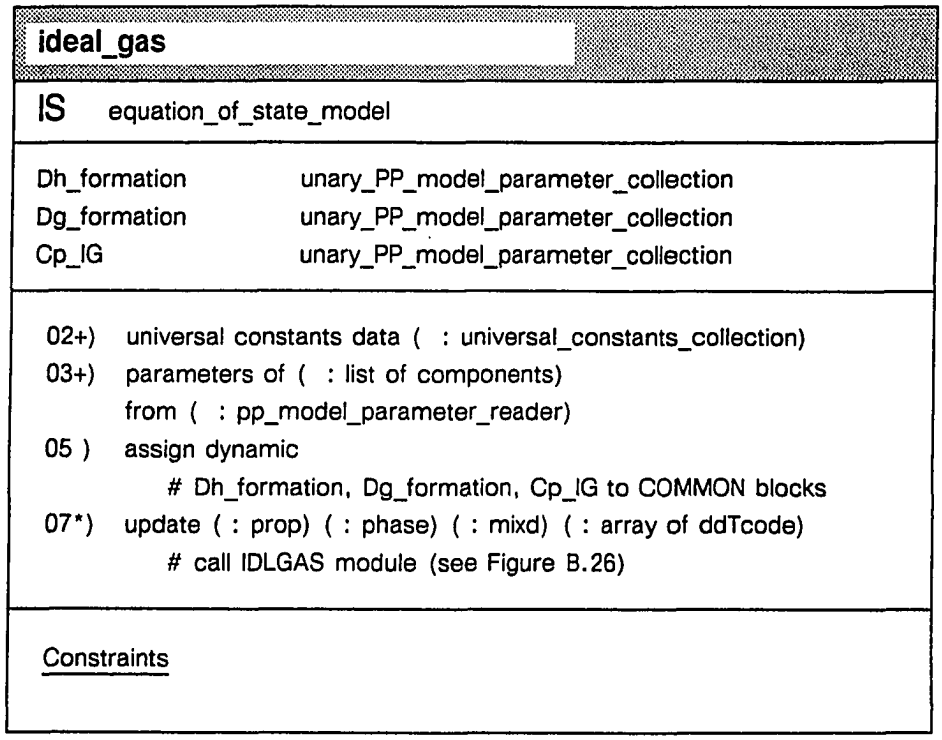


Figure B.9 Design of the GDS ideal_gas

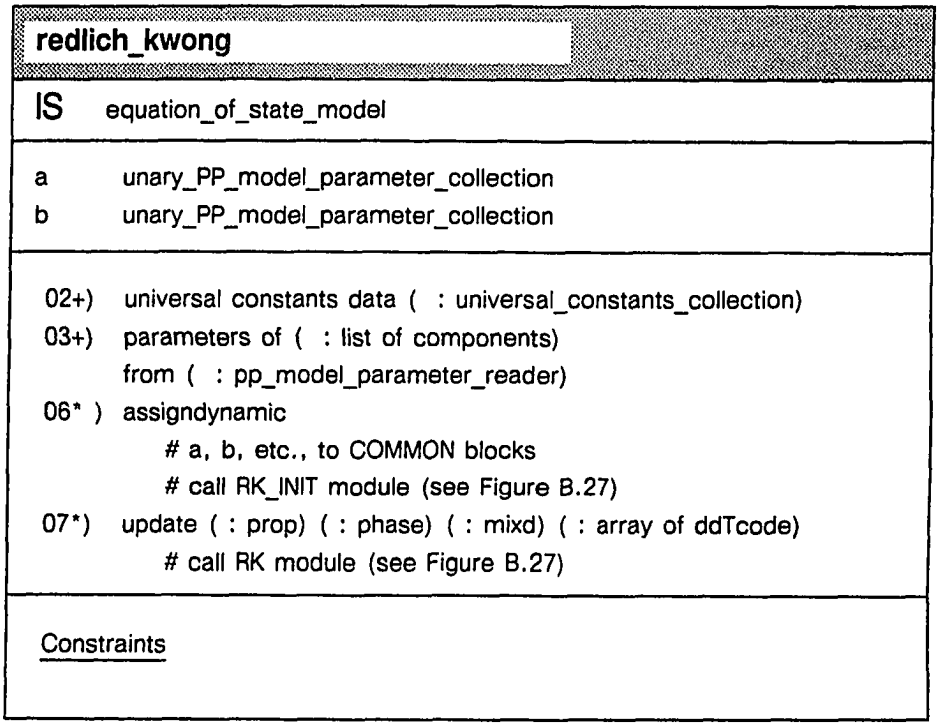


Figure B.10 Design of the GDS redlich_kwong

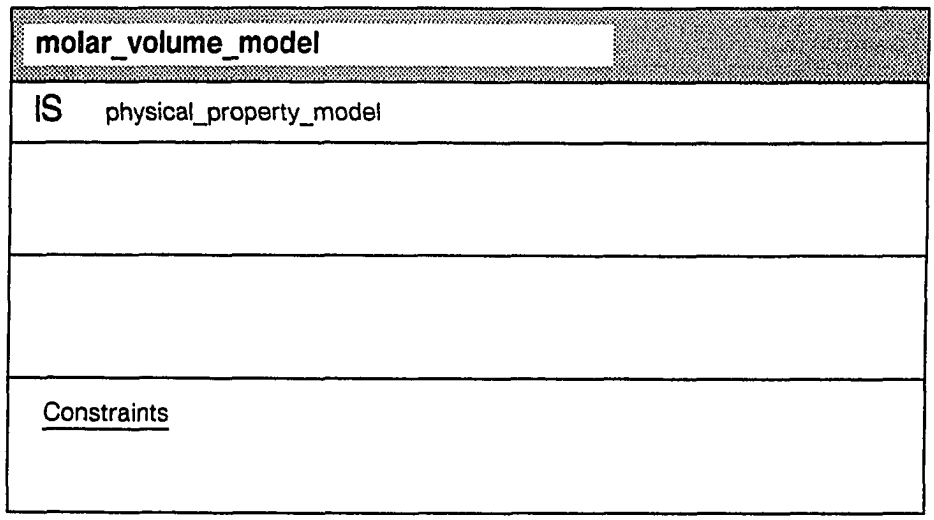


Figure B.11 Design of the GDS molar_volume

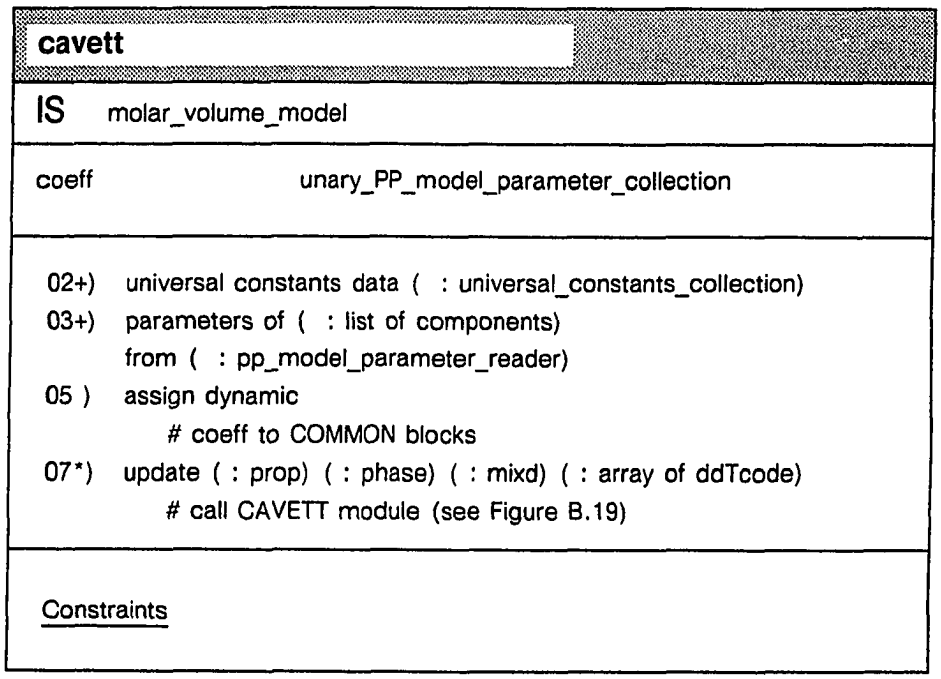


Figure B.12 Design of the GDS cavett

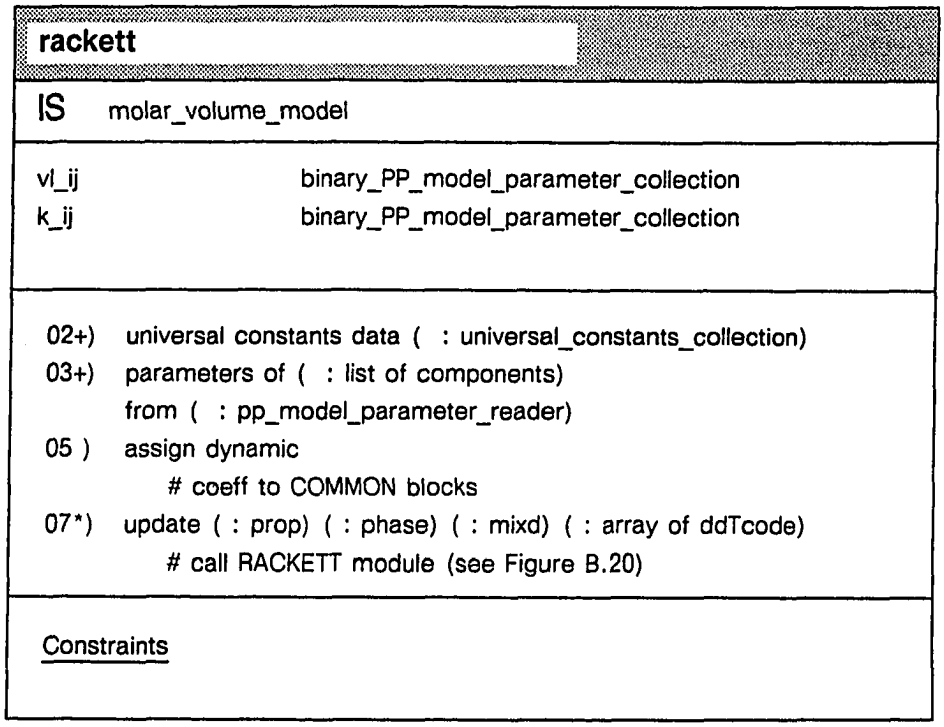
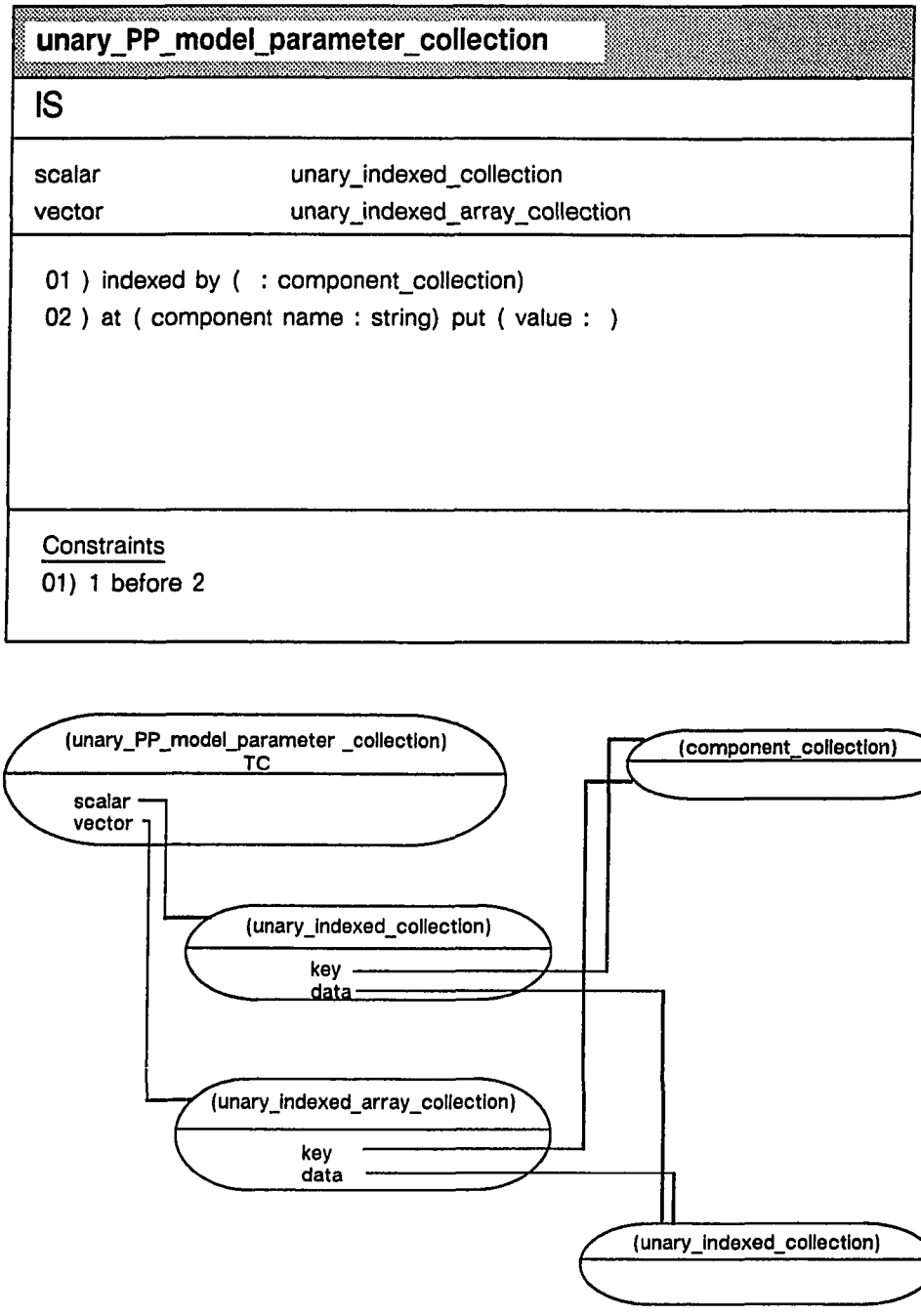
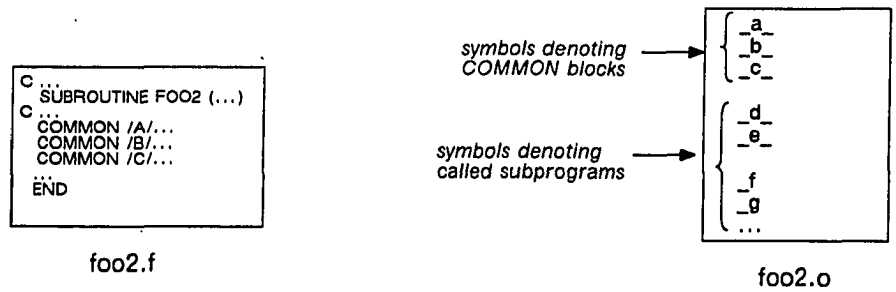


Figure B.13 Design of the GDS rackett



Note: Similar design for the GDS binary_PP_model_parameter, but with different methods of access and update (binary indexed) of symmetric parameters.

Figure B.14 Design of GDS for PP model unary parameters

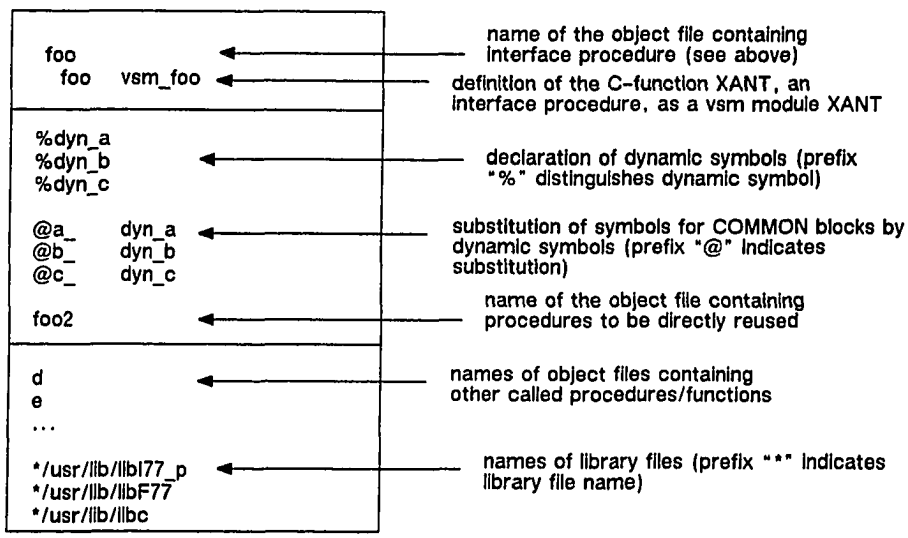


foreign routine

```
external int dyn_a[], dyn_b[];
external float dyn_c[];

foo( )
{
   pop_...( );       /* obtain the necessary data communicated from vsm */
   foo_(X,Y,...);  /* X, Y, etc., macros for expressions evaluating data addresses */
   return 1;
}
```

foo.c

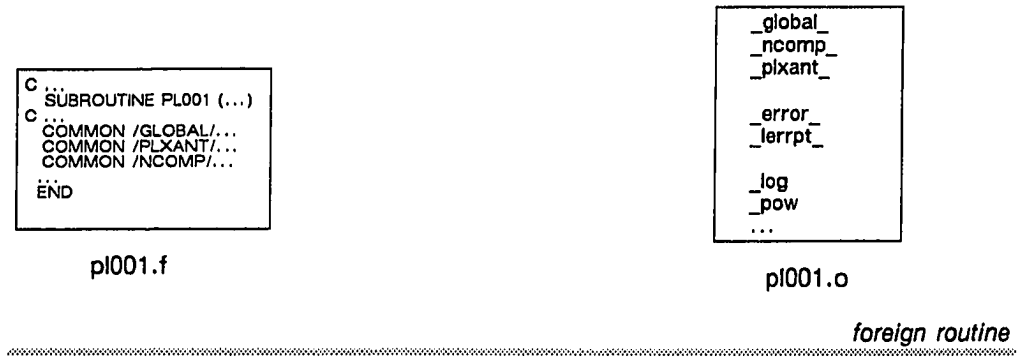
interface routine

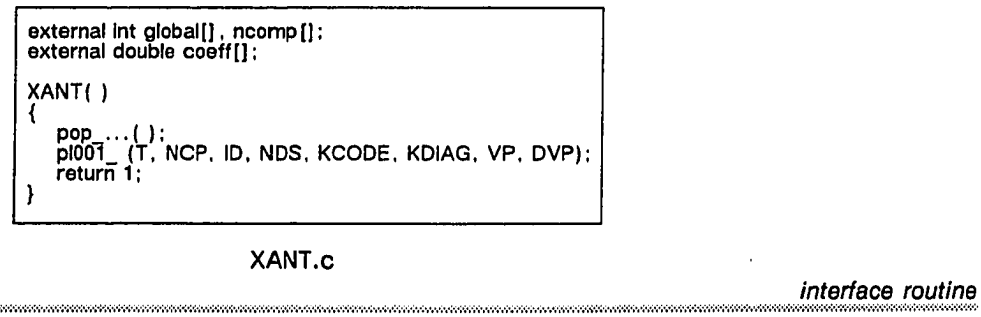


foo.cat

module definition

Figure B.15 Template for design of vsm modules on the following pages





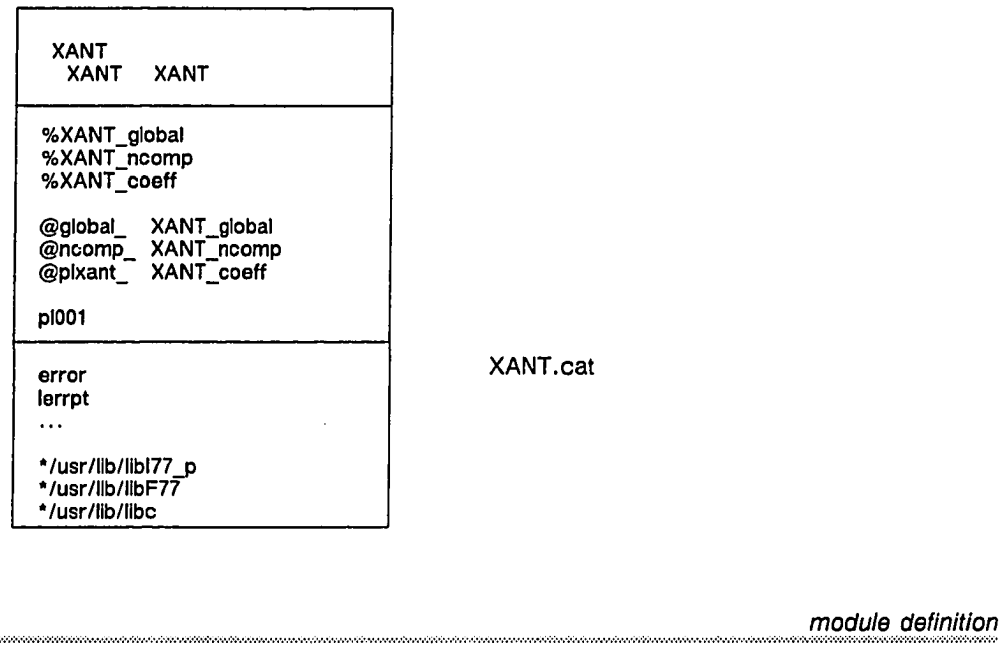


Figure B.16 Vsm Module for the Program Unit PL001

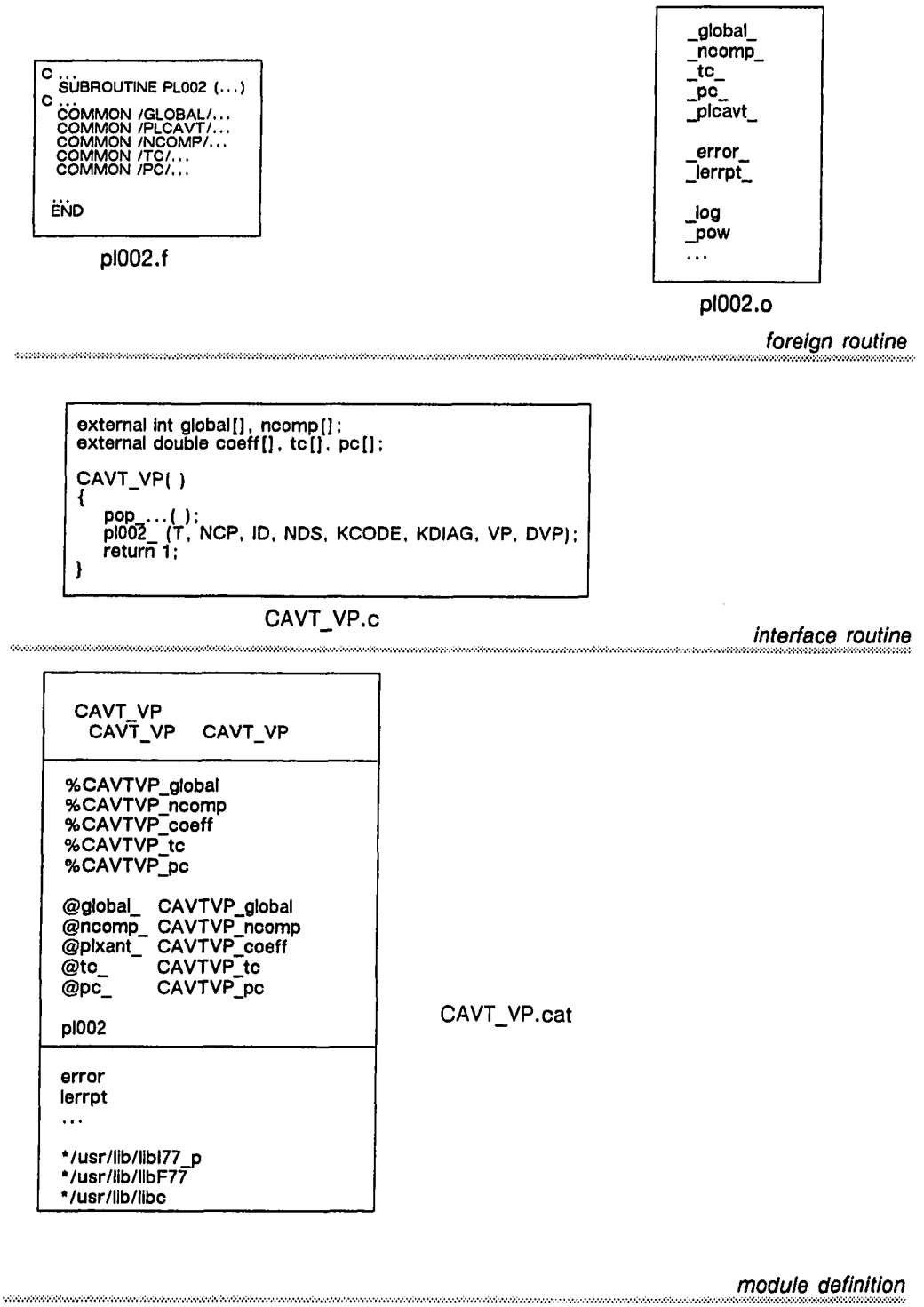


Figure B.17 Vsm Module for the Program Unit PL002

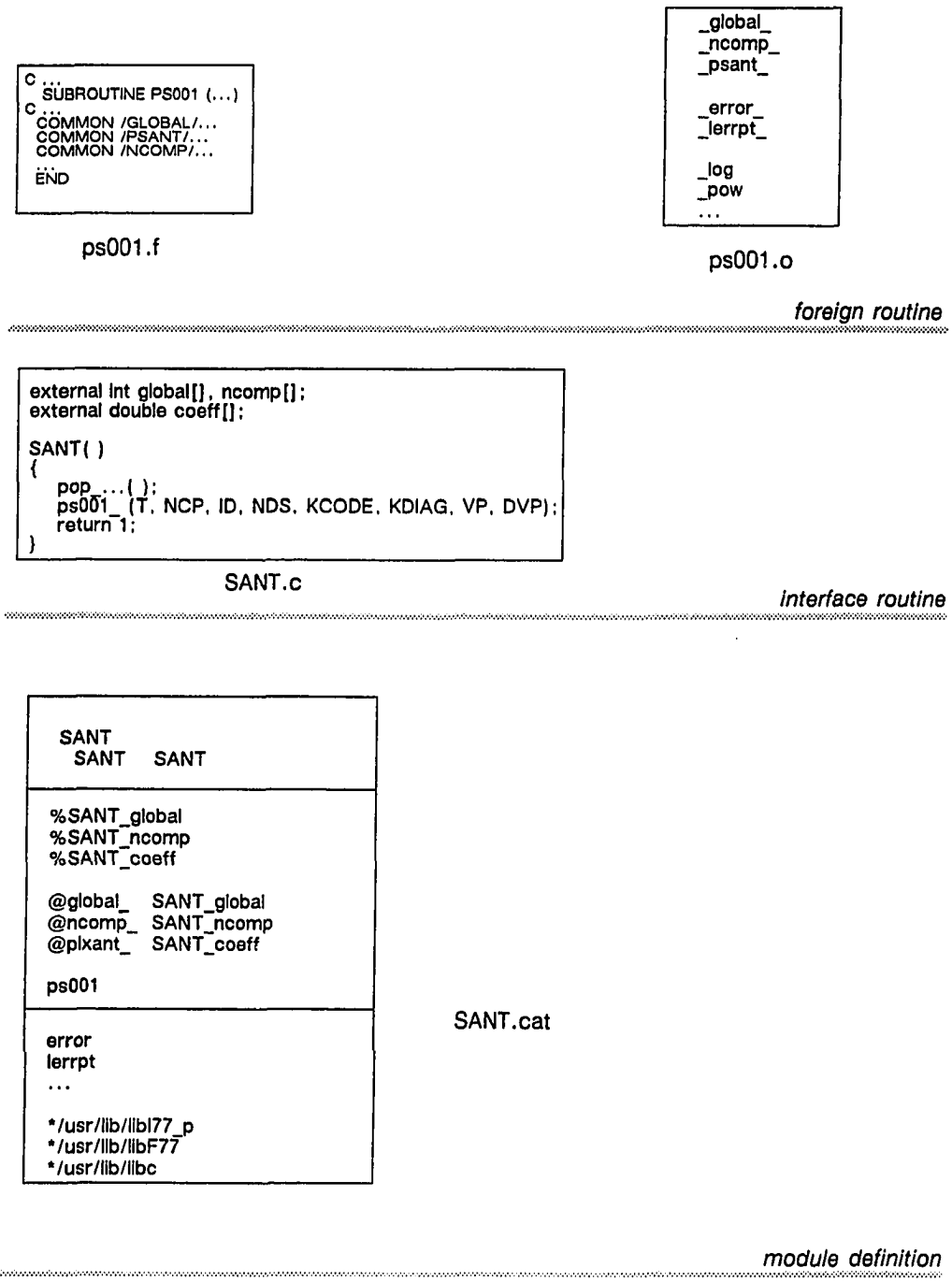


Figure B.18 Vsm Module for the Program Unit PS001

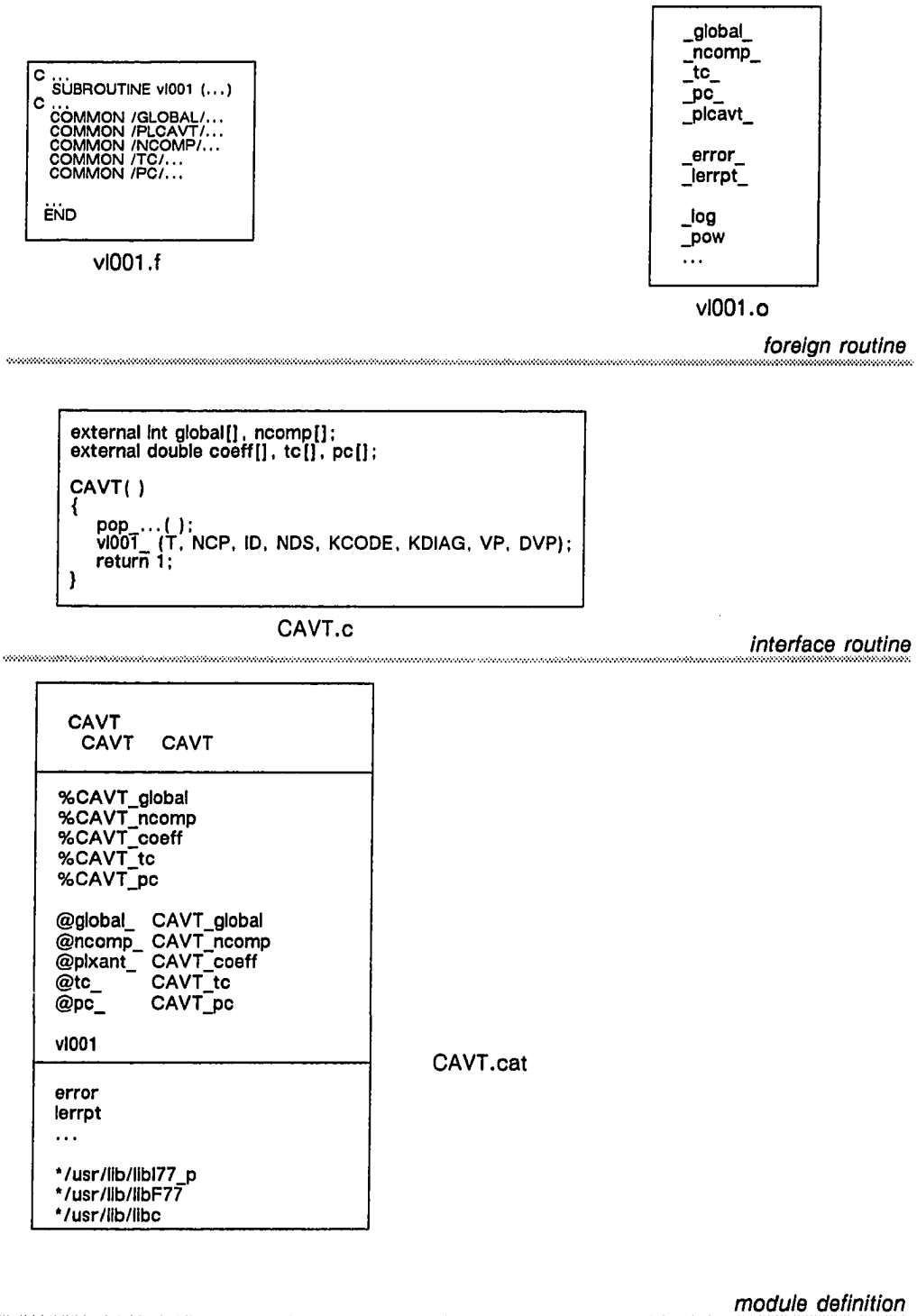


Figure B.19 Vsm Module for the Program Unit VL001

foreign routine

```
external int global[], ncomp[];
external double coeff[];

RKT( )
{
   pop_...( );
   vl004_ (T, NCP, ID, KV, KDIAG, NDS, V, DV, KER);
   return 1;
}
```

RKT.c

interface routine

```
RKT
  RKT    RKT

%RKT_global
%RKT_ncomp
%RKT_coeff

@global_   RKT_global
@ncomp_    RKT_ncomp
@plxant_   RKT_coeff

vl004

error
lerrpt
...

*/usr/lib/libI77_p
*/usr/lib/libF77
*/usr/lib/libc
```

RKT.cat

module definition

Figure B.20 Vsm Module for the Program Unit VL004

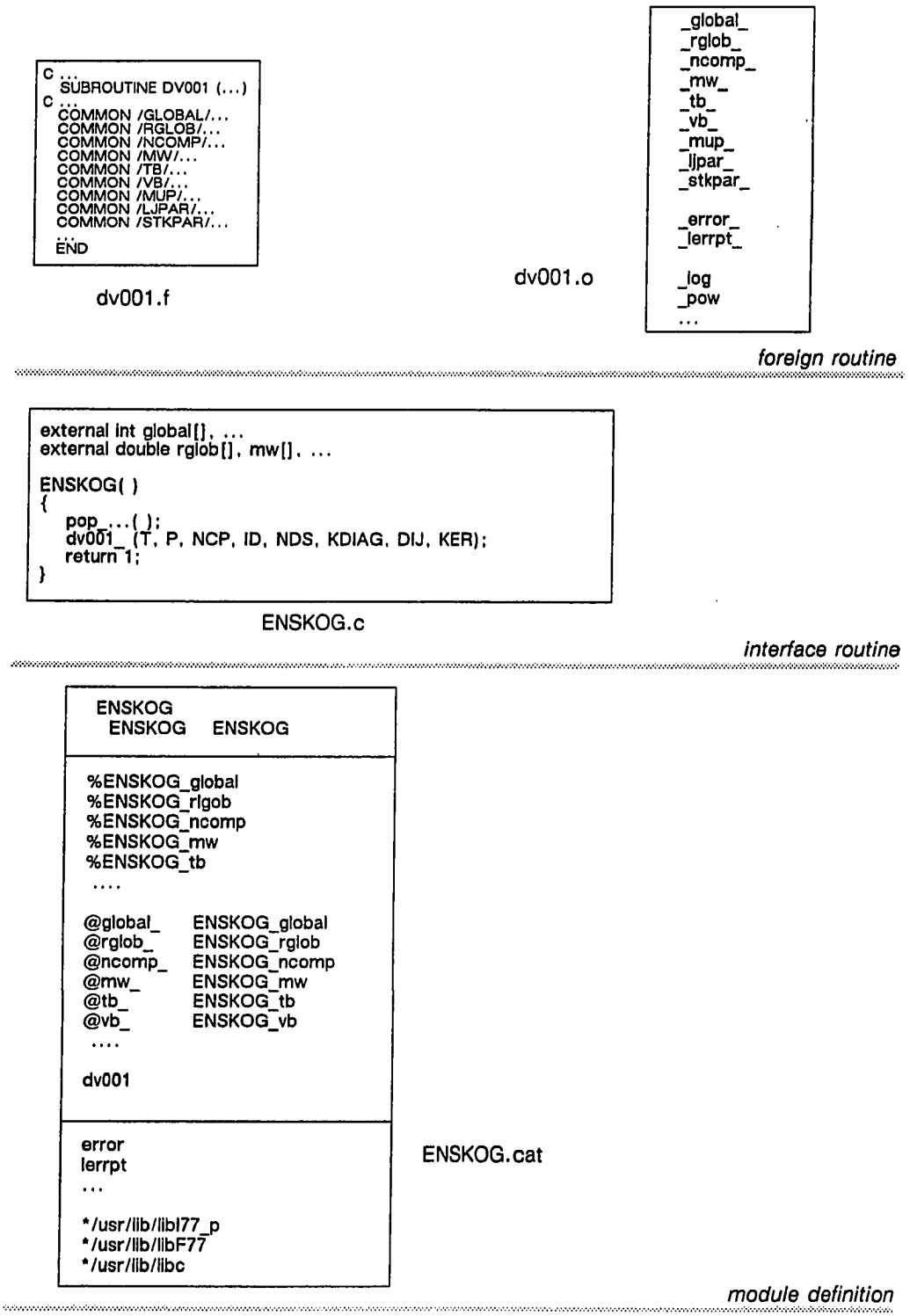


Figure B.21 Vsm Module for the Program Unit DV001


```
external int global[];
external double mw[], vc[];

DAWSON( )
{
    pop_...( );
    dv002_ (X, NCP, ID, RHO, DIJLP, KDIAG, DIJ, KER);
    return 1;
}
```

DAWSON.c

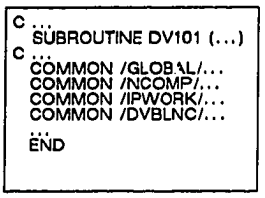
interface routine

```
DAWSON
  DAWSON    DAWSON

%DAWSON_global
%DAWSON_mw
%DAWSON_tb
 ....

@global_     DAWSON_global
@mw_         DAWSON_mw
@vc_         DAWSON_vc
 ....

dv002


error
lerrpt
...

*/usr/lib/libI77_p
*/usr/lib/libF77
*/usr/lib/libc
```

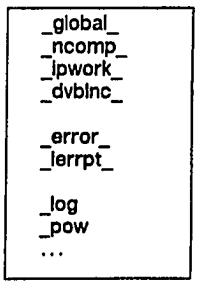
DAWSON.cat

module definition

Figure B.22 Vsm Module for the Program Unit DV002



dv101.f



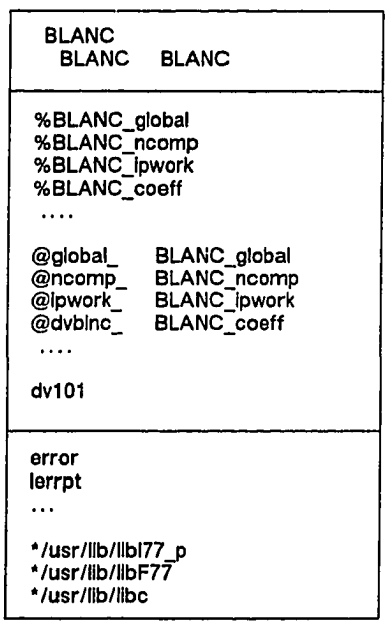
dv101.o

foreign routine

```
external int global[], ...
external double coeff[], work[], ...

BLANC( )
{
   pop_...( );
   dv101_ (T, P, NCP, ID, NDS, DIJ, KER);
   return 1;
}
```

BLANC.c

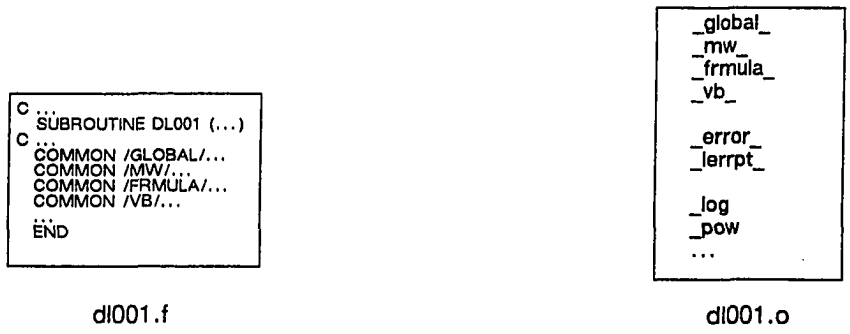
interface routine



BLANC.cat

module definition

Figure B.23 Vsm Module for the Program Unit DV101



foreign routine

```
external int global[], ...
external double coeff[], work[], ...

WILKECH( )
{
   pop_...( );
   dl001_ (T, X, NCP, ID, MUL, KDIAG, DIJ, KER);
   return 1;
}
```

WILKECH.c

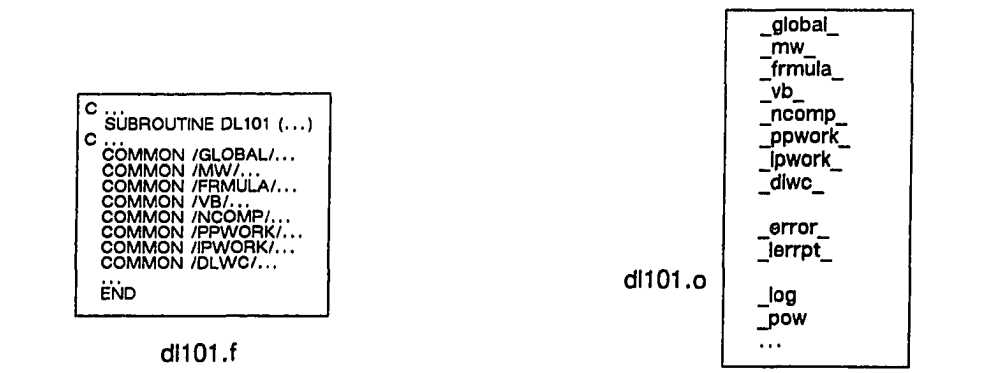
interface routine

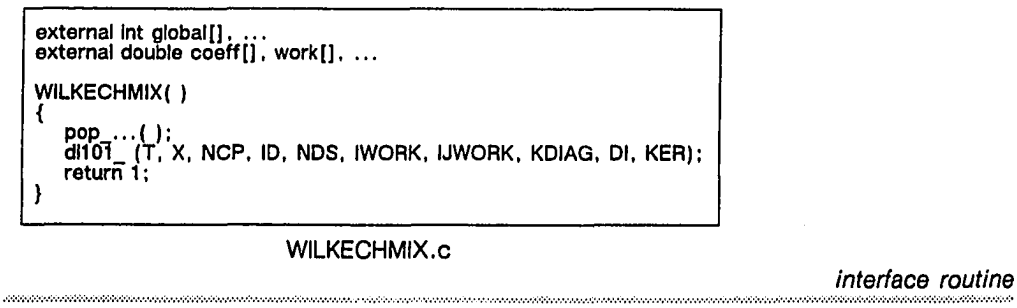
```
WILKECH
  WILKECH    WILKECH

%WILKECH_global
%WILKECH_mw
%WILKECH_formula
%WILKECH_vb
 ....

@global_     WILKECH_global
@mw_         WILKECH_mw
@frmula_     WILKECH_formula
@vb_         WILKECH_vb
 ....

dl001

error
lerrpt
...

*/usr/lib/libI77_p
*/usr/lib/libF77
*/usr/lib/libc
```

WILKECH.cat

module definition

Figure B.24 Vsm Module for the Program Unit DL001





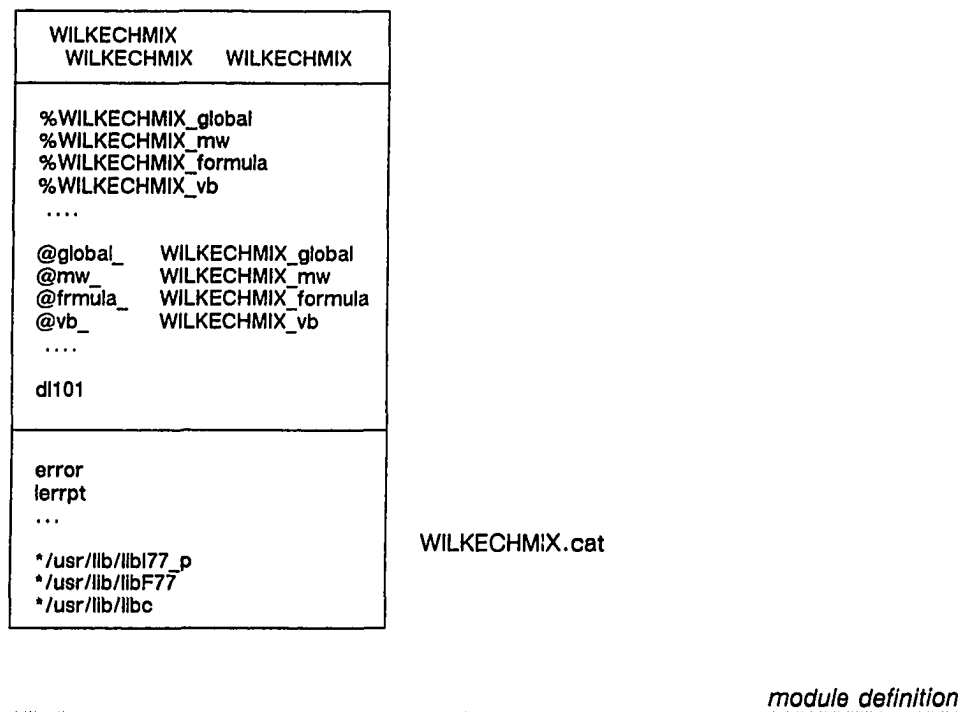
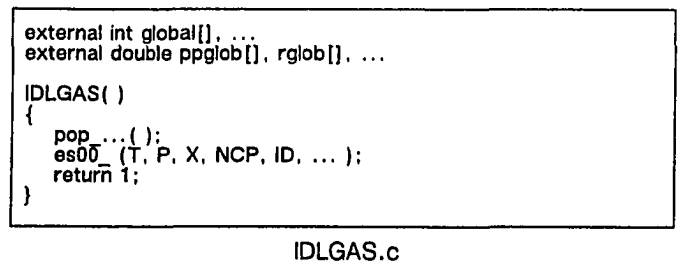


Figure B.25 Vsm Module for the Program Unit DL101



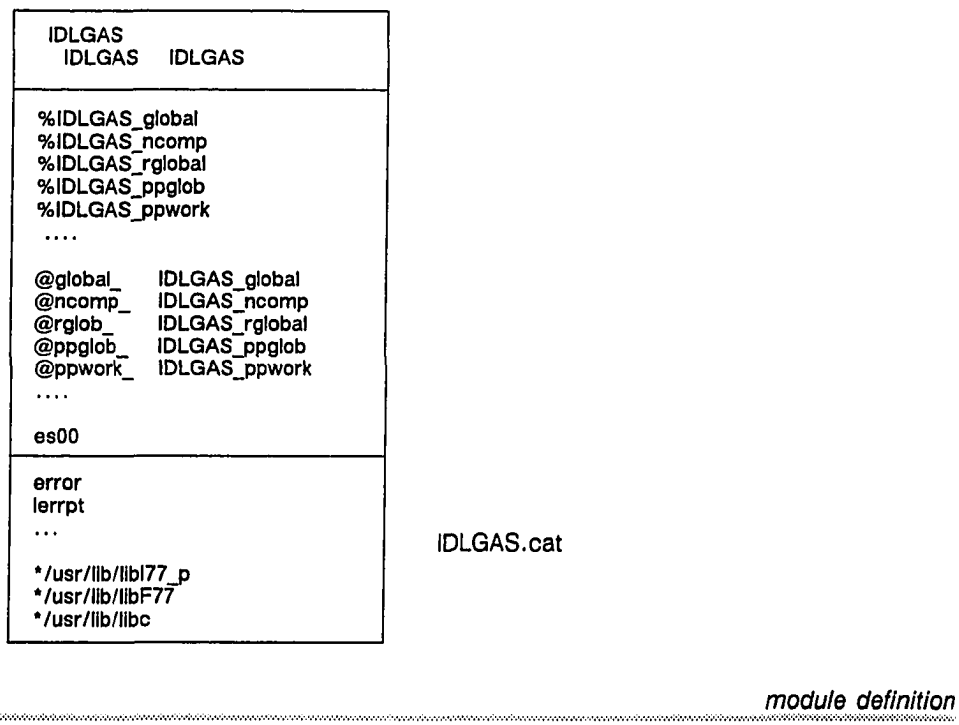


Figure B.26 Vsm Module for the Program Unit ES00

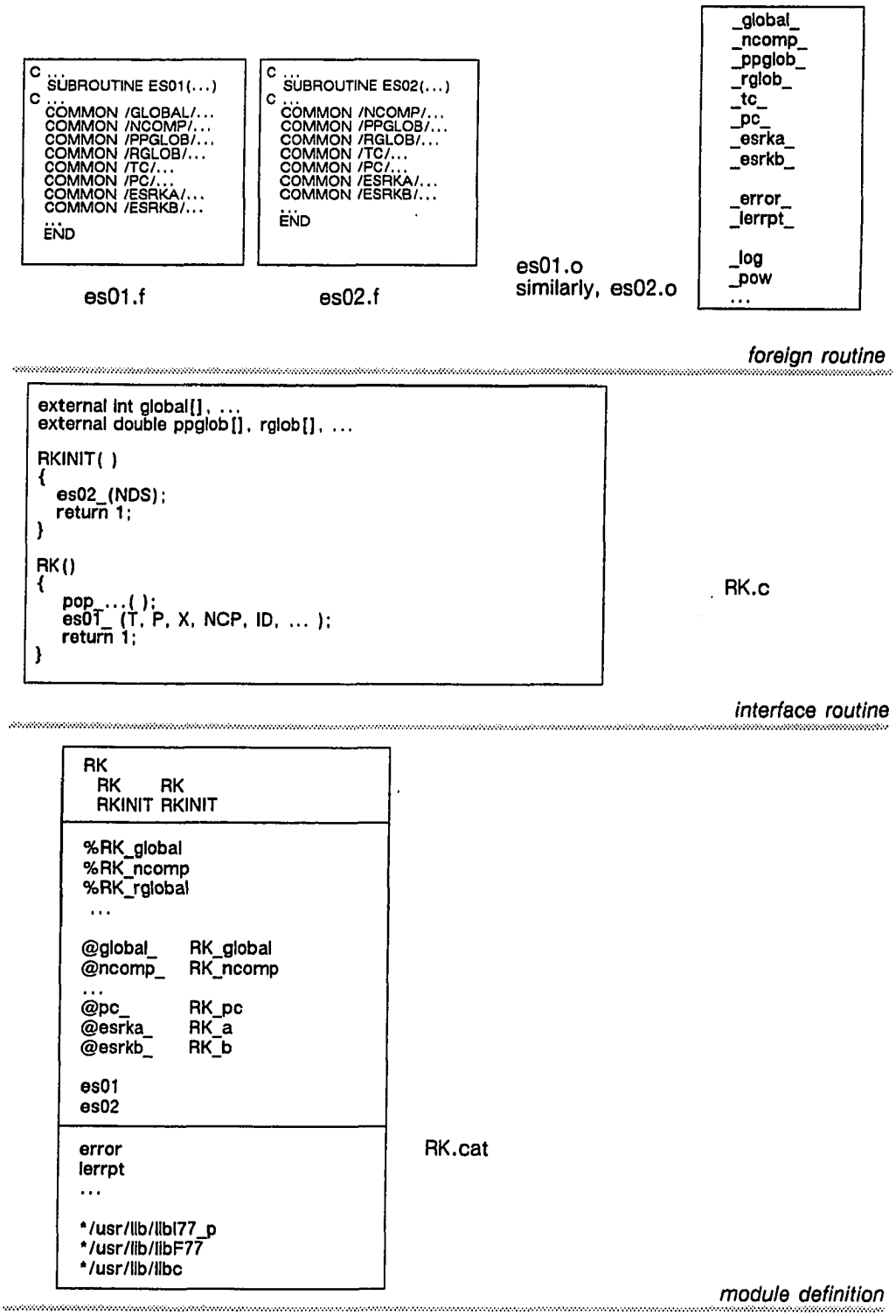
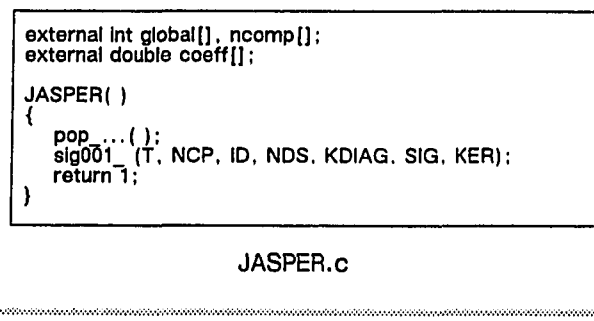


Figure B.27 Vsm Modules for the Program Units ES01 and ES02



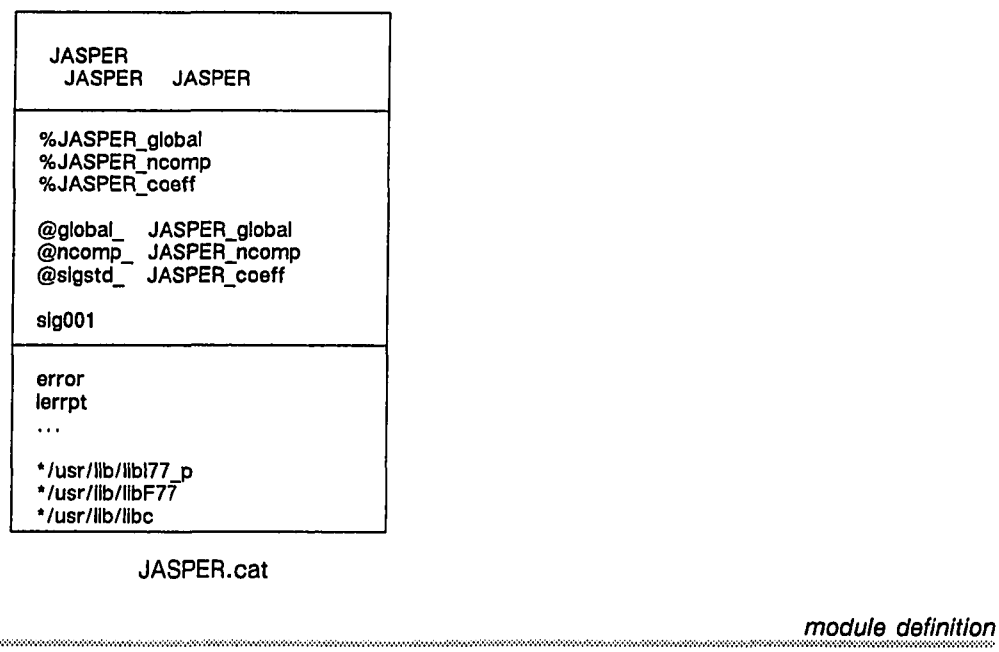


Figure B.28 Vsm Module for the Program Unit SIG001

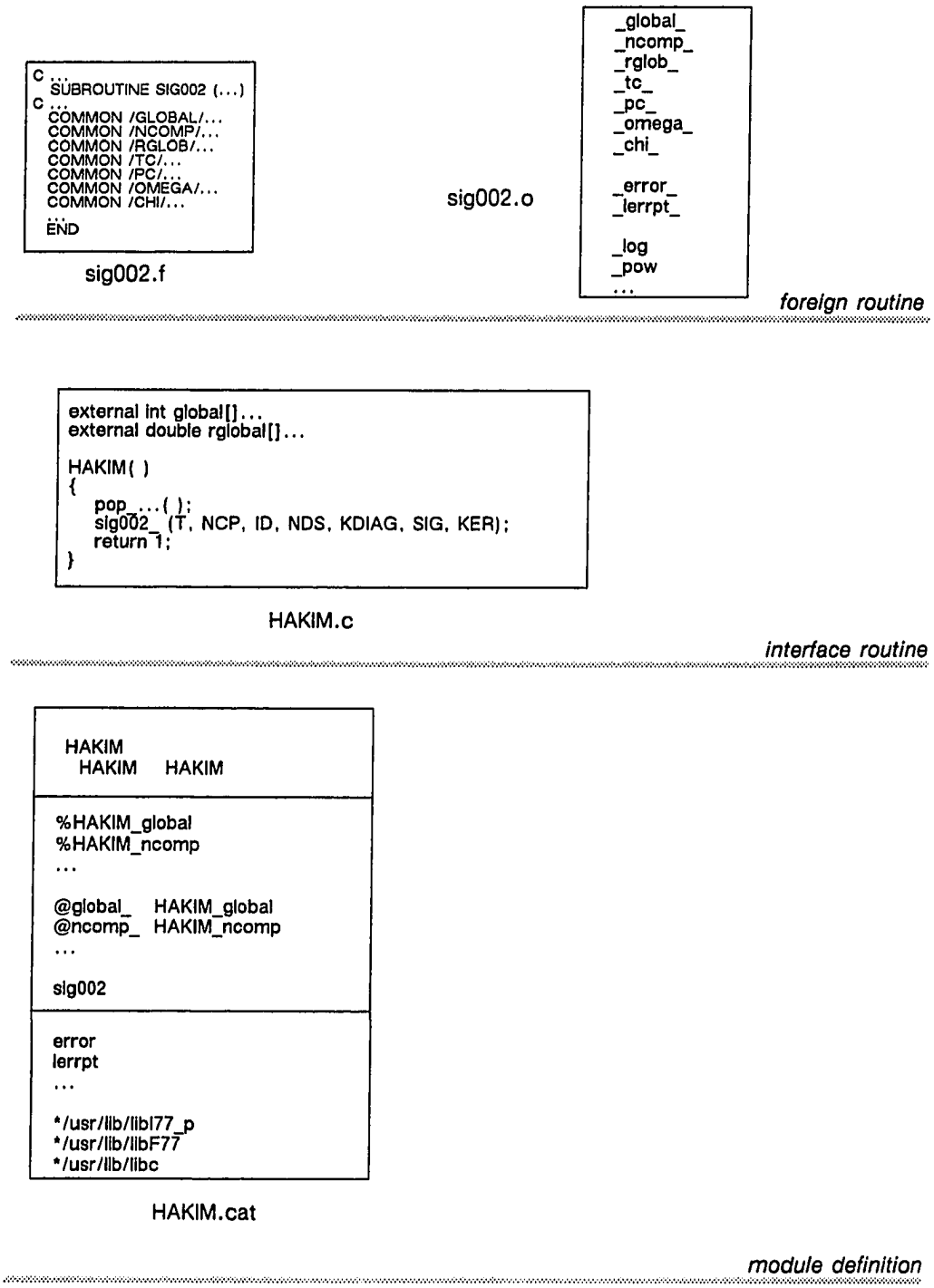
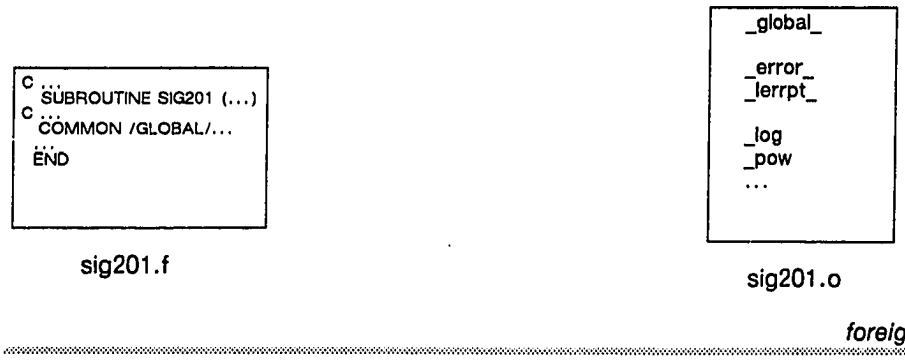
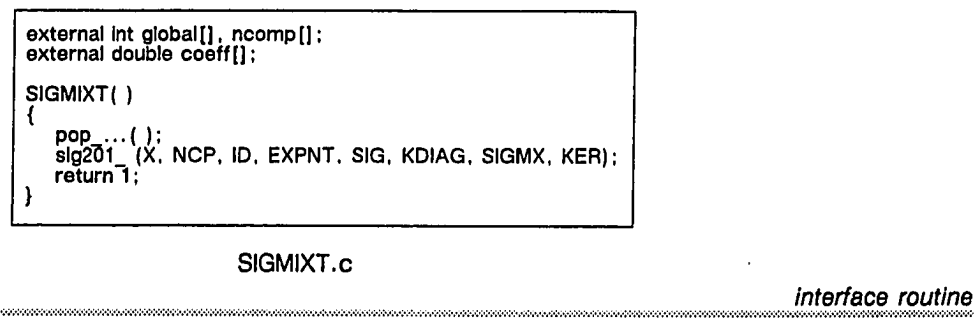


Figure B.29 Vsm Module for the Program Unit SIG002





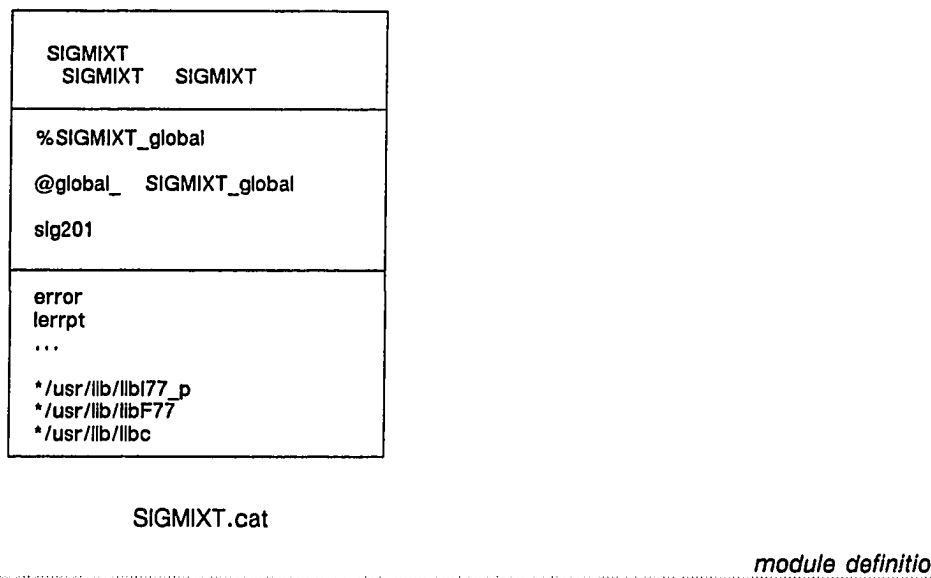
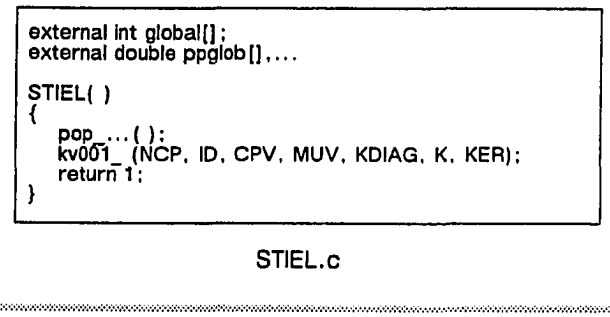


Figure B.30 Vsm Module for the Program Unit SIG201



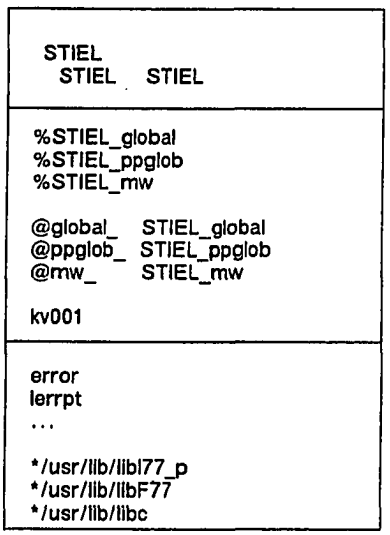


Figure B.31 Vsm Module for the Program Unit KV001

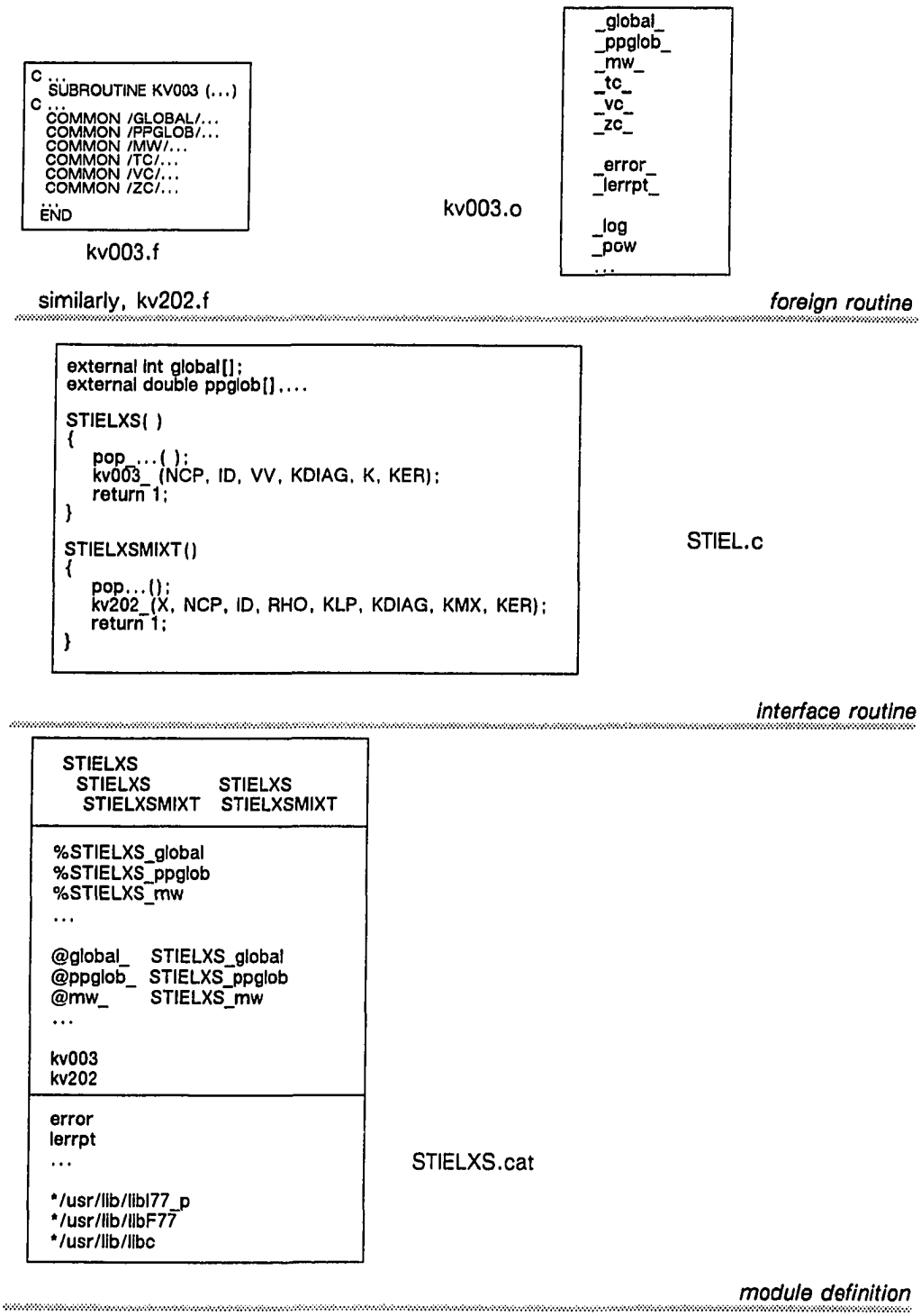
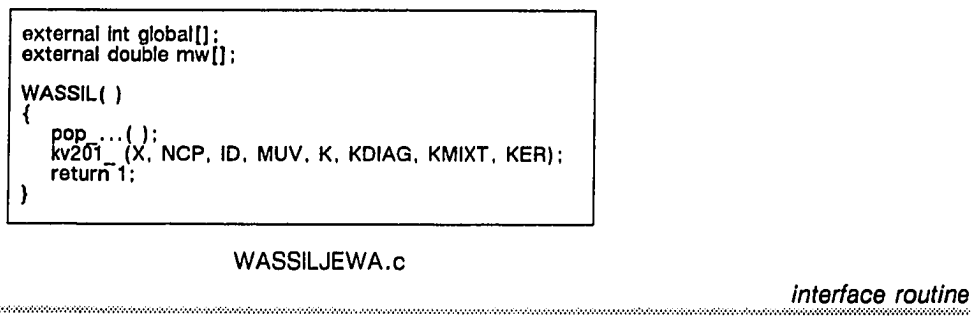


Figure B.32 Vsm Modules for the Program Units KV003 and KV202



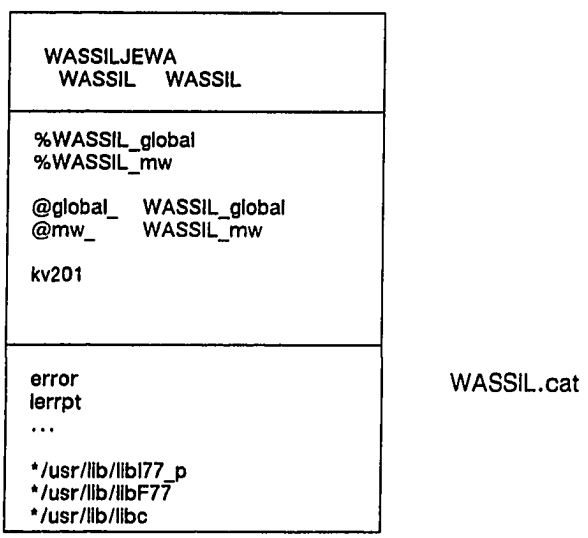
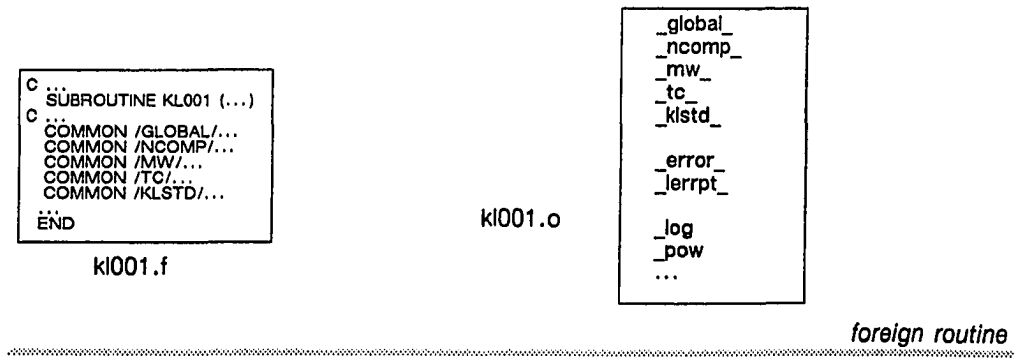
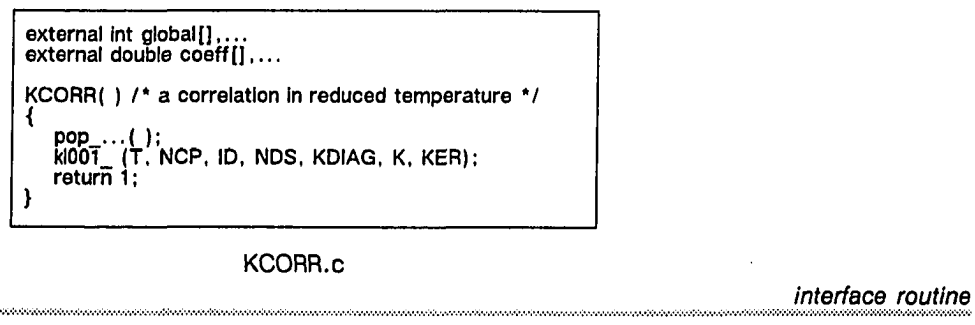


Figure B.33 Vsm Module for the Program Unit KV201





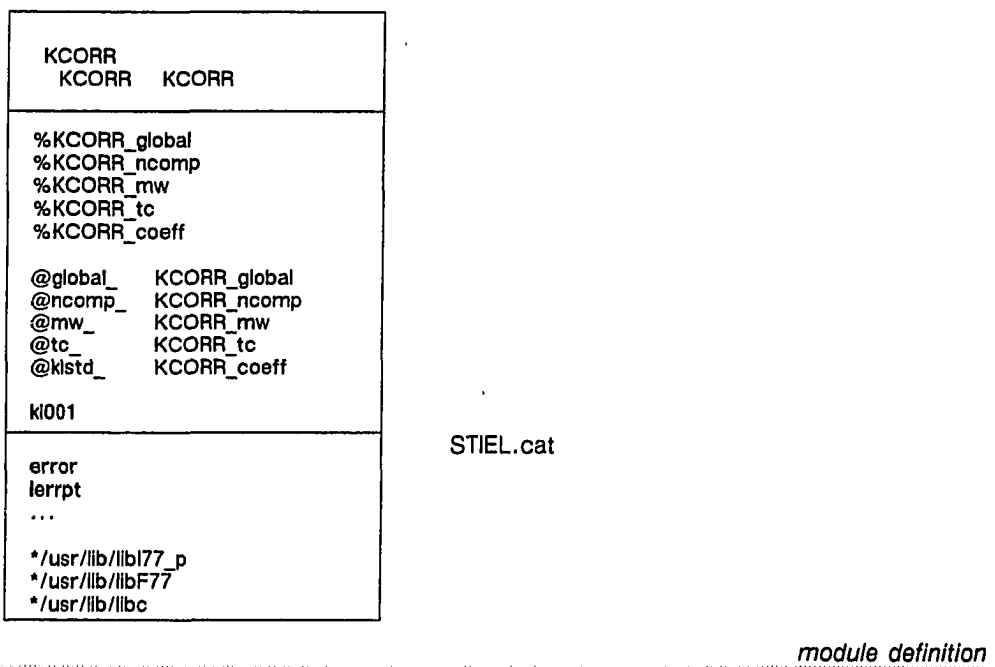


Figure B.34 Vsm Module for the Program Unit KL001

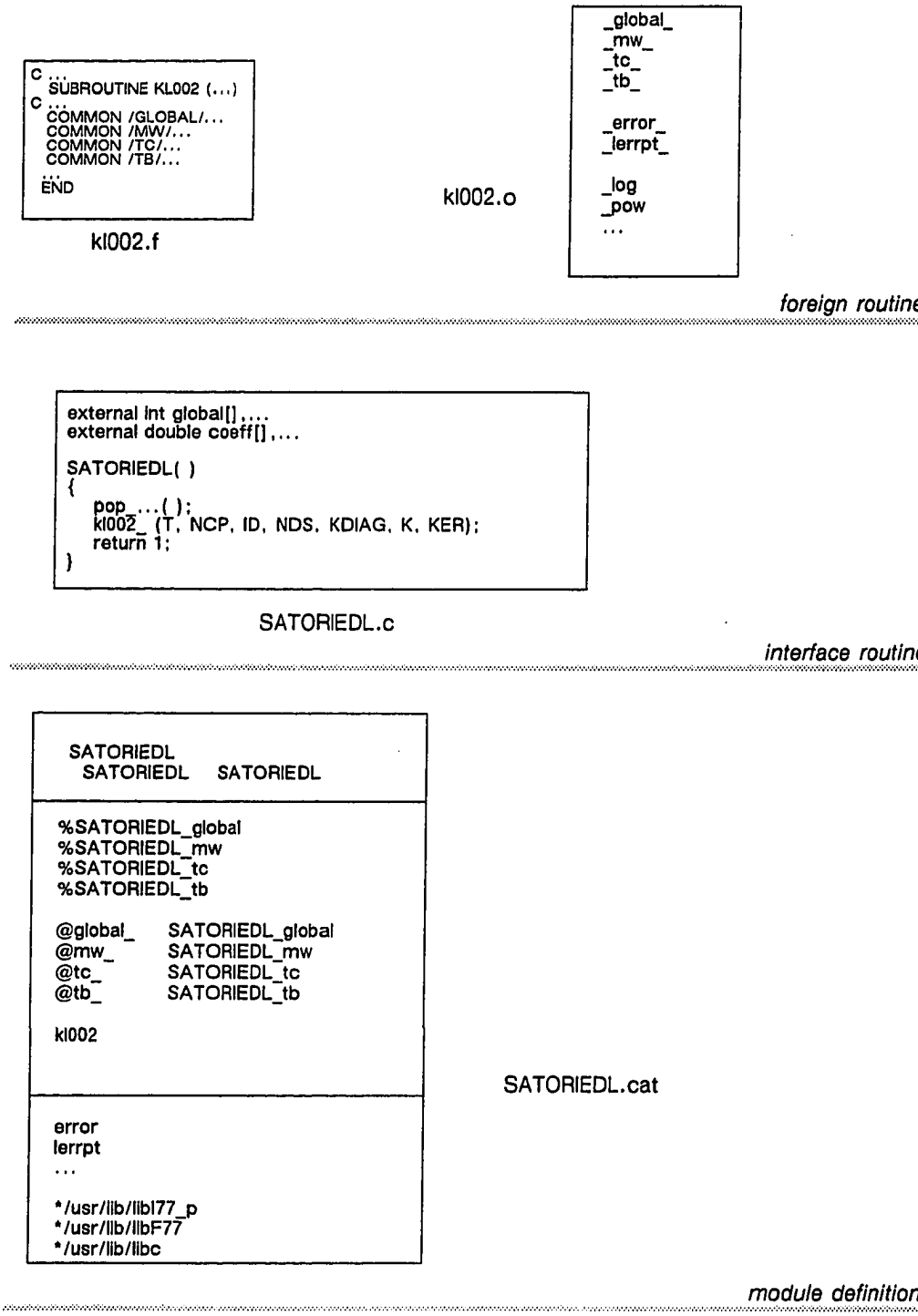
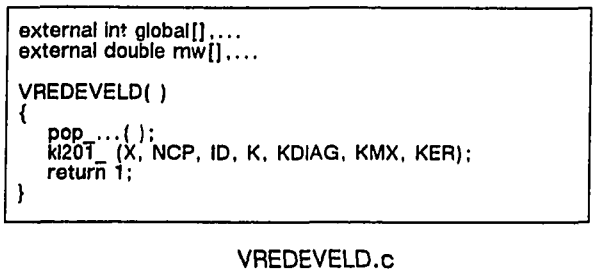


Figure B.35 Vsm Module for the Program Unit KL002



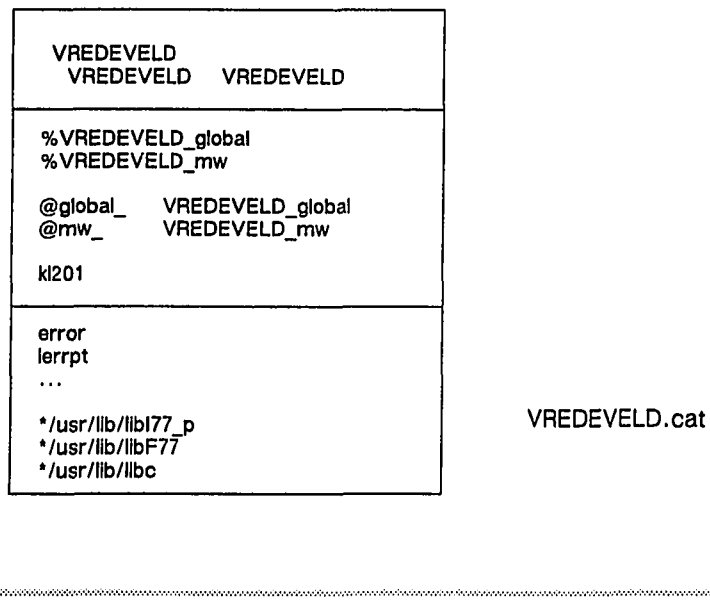


Figure B.36 Vsm Module for the Program Unit KL201

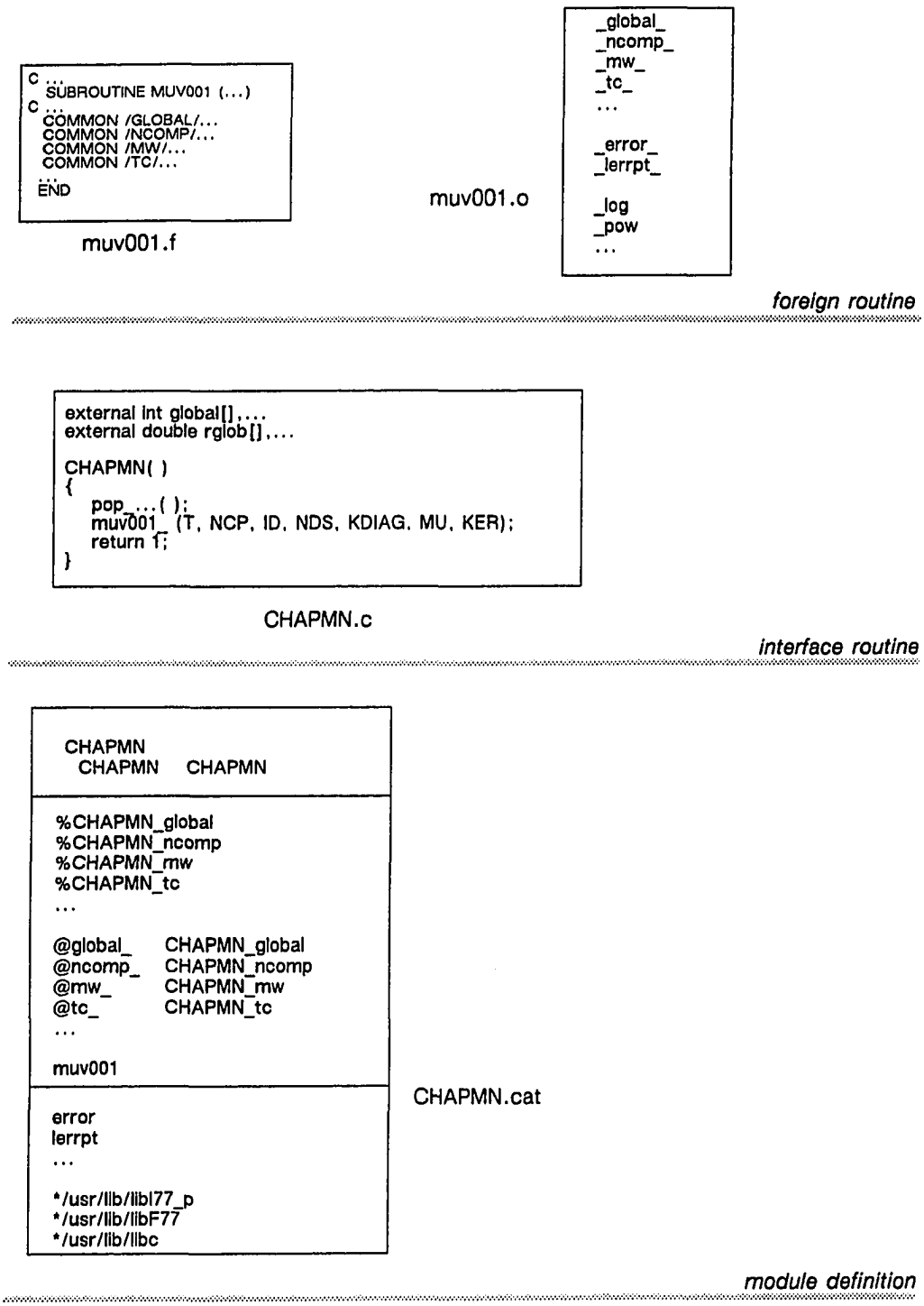


Figure B.37 Vsm Module for the Program Unit MUV001

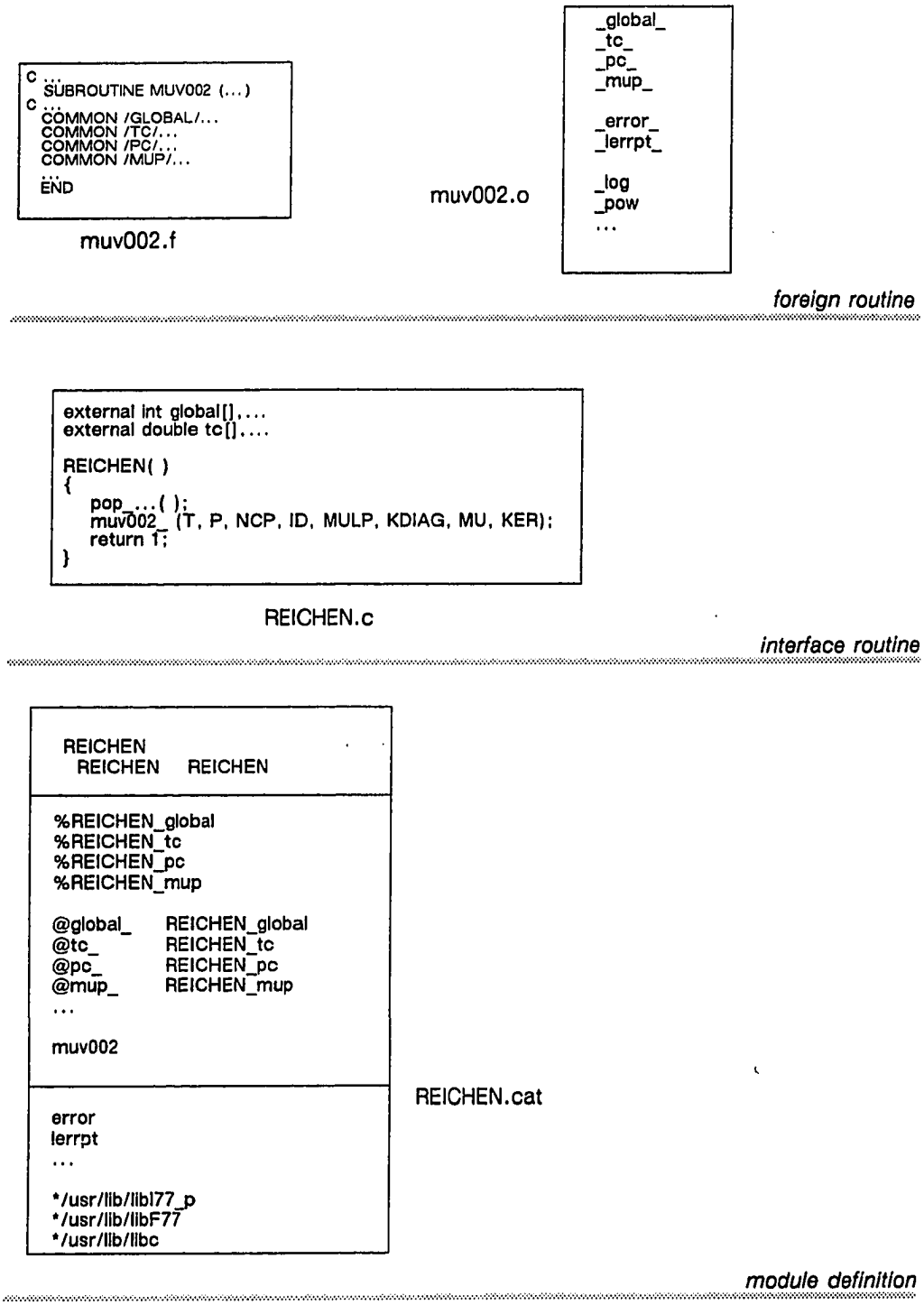


Figure B.38 Vsm Module for the Program Unit MUV002

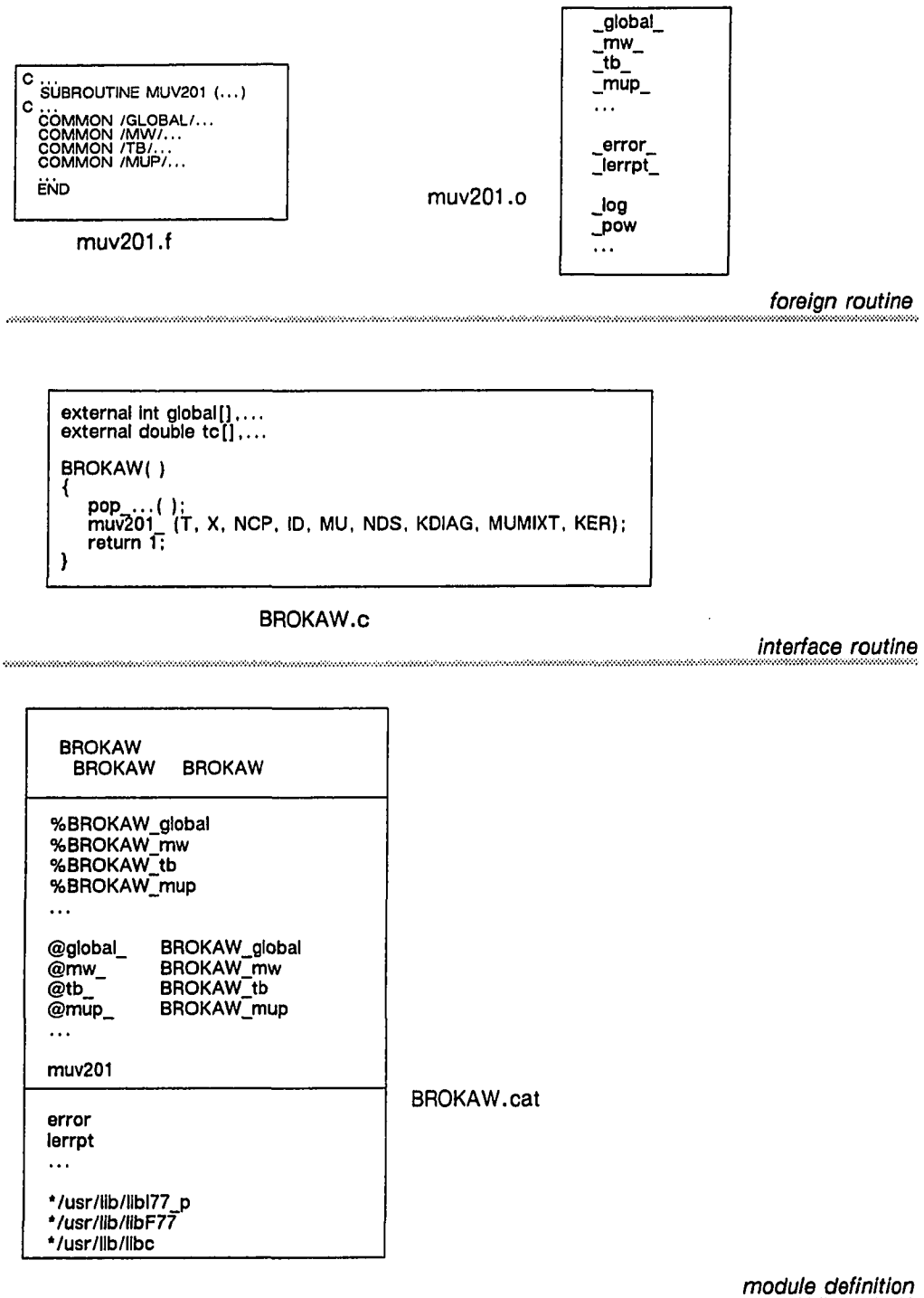


Figure B.39 Vsm Module for the Program Unit MUV201

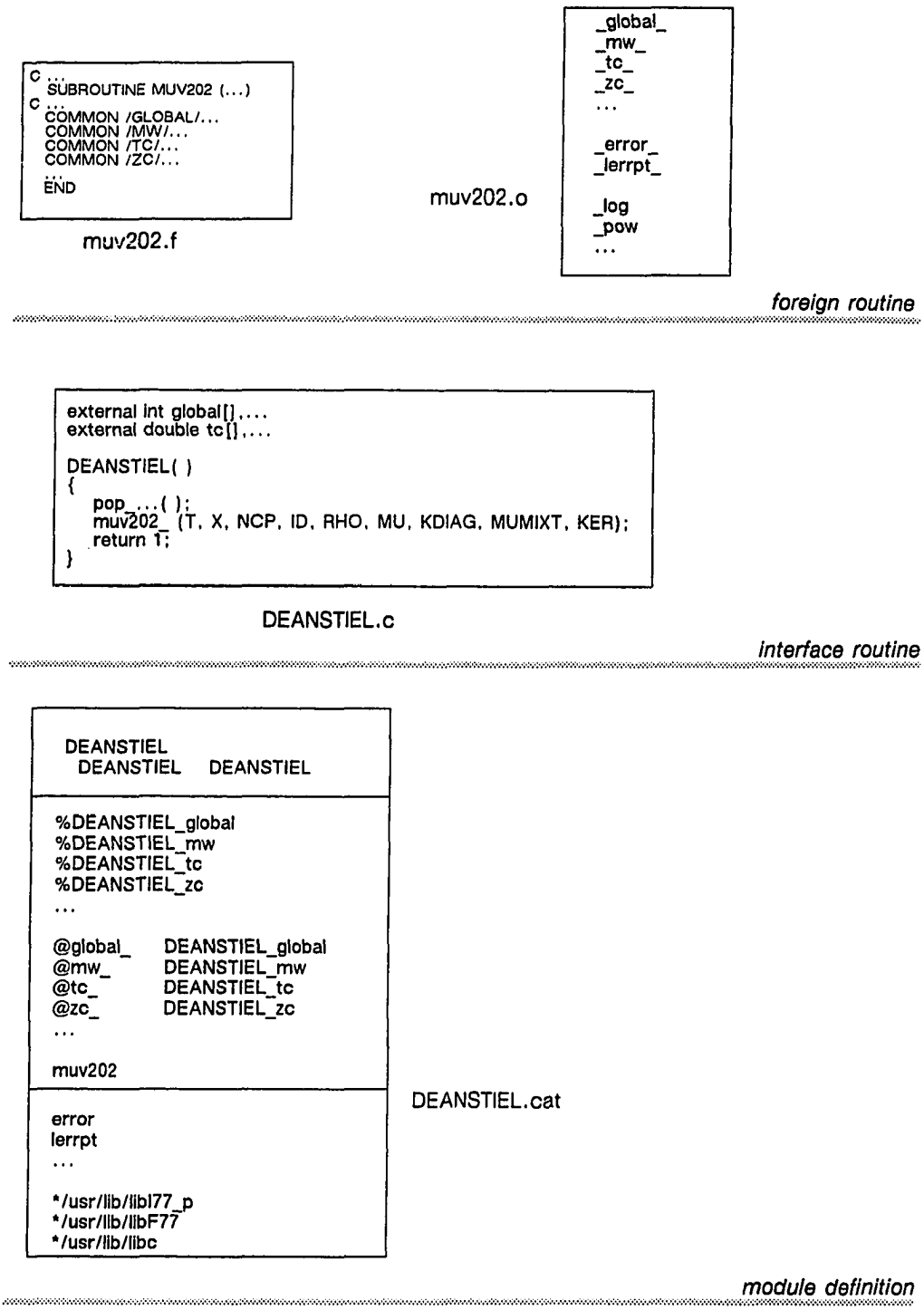


Figure B.40 Vsm Module for the Program Unit MUV202

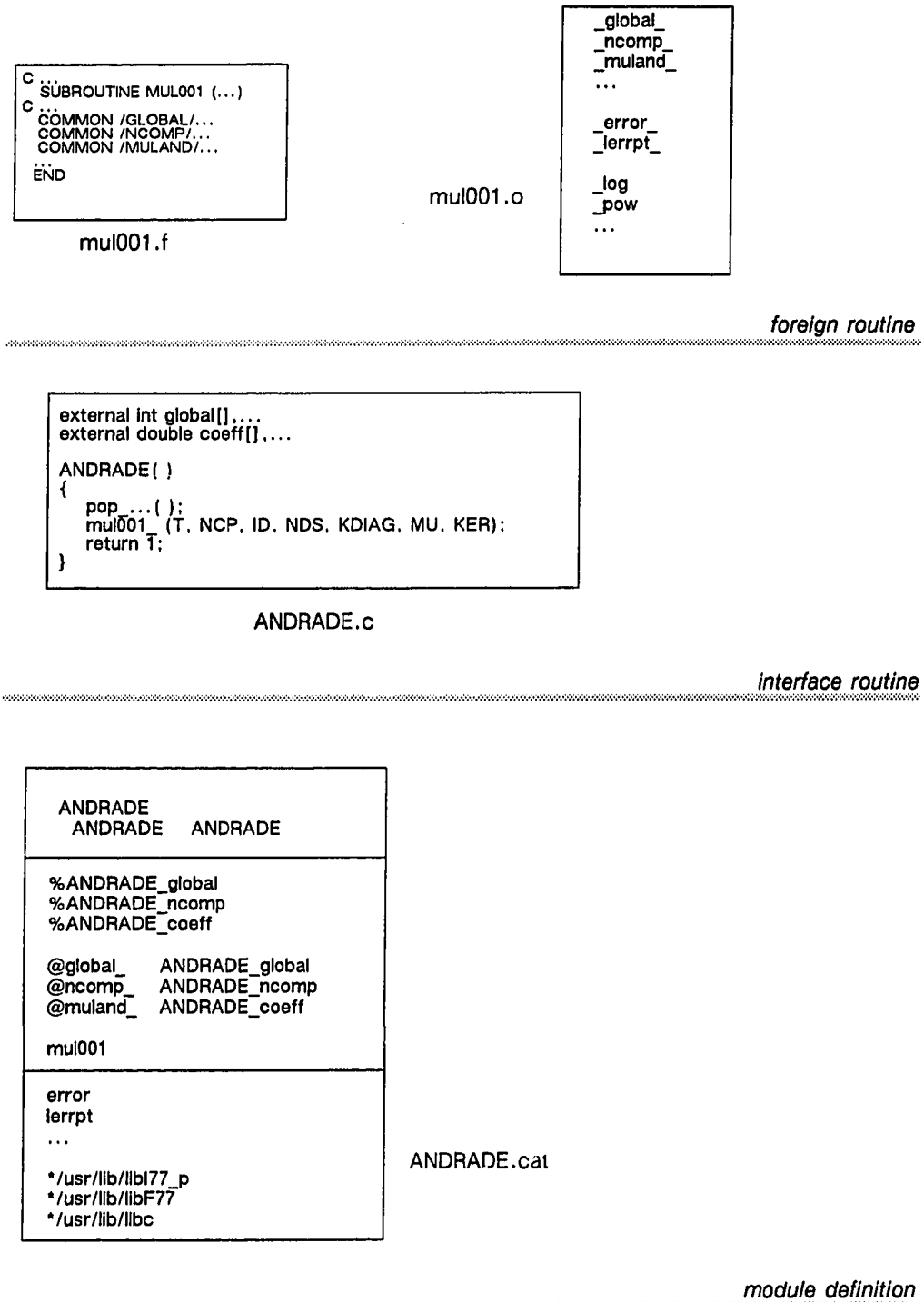
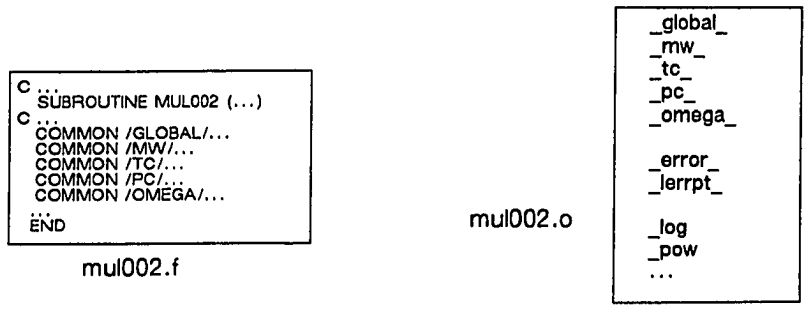


Figure B.41 Vsm Module for the Program Unit MUL001



foreign routine

```
external int global[],...
external double mw[],...

LETSOU( )
{
   pop_...( );
   mul002_ (T, NCP, ID, KDIAG, MU, KER);
   return 1;
}
```

LETSOU.c

interface routine

```
LETSOU
  LETSOU    LETSOU

%LETSOU_global
%LETSOU_mw
%LETSOU_tc
%LETSOU_pc
%LETSOU_omega

@global_    LETSOU_global
@mw_        LETSOU_mw
@tc_        LETSOU_tc
@pc_        LETSOU_pc
@omega_     LETSOU_omega

mul002

error
lerrpt
...

*/usr/lib/libI77_p
*/usr/lib/libF77
*/usr/lib/libc
```

LETSOU.cat

module definition

Figure B.42 Vsm Module for the Program Unit MUL002

References

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). Compilers, principles, techniques, and tools. Reading, MA: Addison-Wesley.

Encyclopedia Americana. (1989). (Vol. 13). Danbury, CT: Grolier.

Anderson, K. J., Beck R. P., & Buonanno, T. E. (1988). Reuse of software modules. AT&T Technical Journal, 67 (1), 71–76.

Bachman, C. W. (1988). A CASE for reverse engineering. Datamation, 34 (13), pp. 49–56.

Bachman, C. W. (1990). A personal chronicle: Creating better information systems, with some guiding principles. IEEE Transactions on Data and Knowledge Engineering, 1 (1), 17–32.

Benayoune, M., & Preece, P. E. (1987). Review of information management in computer–aided engineering. Computers and Chemical Engineering, 11 (1), 1–6.

Biggerstaff, T. J., & Perlis, A. J. (1984). Software reuse [Special Issue]. IEEE Transactions on Software Engineering, SE–10, (5).

Biggerstaff, T. J. (1991). Design recovery for maintenance and reuse. Computer, 22 (7), 36–49.

Blaha, M. R. (1984). Application of data management technology to process engineering. Unpublished doctoral dissertation, Washington University in St. Louis.

Blaha, M. R., Yamashita Y. & Motard R. L. (1985). Database management systems for the process engineer. Chemical Engineering Progress, 81 (9), pp. 45–49.

Booch, G. (1991). Object-oriented design. Redwood City, CA: Benjamin Cummings.

Bray, O. H. (1987). Data management for engineers: Current capabilities, future directions. Computers in Mechanical Engineering, 5 (5), pp. 20–24.

Bushnell, M. L. (1988). Design automation: Automated full-custom VLSI layout using the ULYSSES design environment. Boston: Academic Press.

Caldiera, G., & Basili, V. R. (1991). Identifying and qualifying reusable software components. Computer, 24 (2), 61–70.

Cattell, R. G. G. (1991). Object data mangement: Object-oriented and extended relational database systems. Reading, MA: Addison–Wesley.

Chikofsky, E. J., & Cross II, J. H. (1990). Reverse engineering and design recovery: A taxonomy. IEEE Software, 7 (1), pp. 13–17.

Cifeuntes L. (1987). Selecting process modeling software. Chemical Engineering, 94 (16), pp. 97–99.

Codd, E. F. (1970). A relational model for large shared data banks. Communication of the ACM, 13 (6), 377–387.

Cox, B. J. (1986). Object-oriented programming: An evolutionary approach, Reading, MA: Addison–Wesley.

Dahl, O., & Hoare, C. A. R. (1972). Hierarchical program structure. In O. Dahl, E. Dijkstra, & C. A. R. Hoare, Structured programming (pp. 175–220). New York: Academic Press.

Date, C. J. (1986). An introduction to database systems. (4th ed.) Reading, MA: Addison–Wesley.

Deltz, D. (1988). Tools for total quality. Computers in Mechanical Engineering, 7 (1), pp. 8–13.

Dittrich, K. R., & Dayal. U. (Eds.) (1986). Proceedings of the international

workshop on object-oriented database systems. Los Alamitos, CA: IEEE Computer Society Press.

Durek, T., & van Horne F. (1988). Systems software development: Building canonical libraries. Signal, 42 (8), pp. 89–93.

Eastman, C. M. (1981). Database facilities for engineering design. Proceedings of the IEEE, 69 (10), 1249–1263.

Edmunds, R. A. (1985). The Prentice-Hall Standard Glossary of Computer Terminology. Englewood Cliffs, NJ: Prentice Hall.

Engelke, W. D. (1987). How to integrate CAD/CAM systems. New York: Marcel Dekker.

Fulton, R. E. (1987). A framework for innovation. Computers in Mechanical Engineering, 5 (5), pp. 26–40.

Gadient, A. J. (1987). Engineering information systems: Implementation approaches and issues. In Proceedings of the third international conference on data engineering, (pp. 567–578). Los Alamitos, CA: IEEE Computer Society Press.

Golay, M. E. (1990, April). Advanced light-water reactors. Scientific American, 262, pp. 82–89.

Goldberg, A., & Robson, D. (1983). SmallTalk-80: The language and its implementation. Reading, MA: Addison-Wesley.

Graham, S. A. Jr., & Giambelluca, R. (1987). Information systems--tools for project management. Chemical Engineering Progress, 83 (1), pp. 52–59.

Graham, L. E. (1982a). ASPEN User Manual (Vol. 2). Washington, DC: U.S. Department of Energy. (NTIS Publication No. DE82020196)

Graham, L. E. (1982b). ASPEN System Administrator Manual (Vol. 1). Washington, DC: U.S. Department of Energy. (NTIS Publication No. DE82020196)

Graham, L. E. (1982c). ASPEN System Administrator Manual (Vol. 2). Washington, DC: U.S. Department of Energy. (NTIS Publication No. DE82020196)

Gundersen, T. (1991). Achievements and future challenges in industrial design applications of process systems engineering. In Proceedings of the 4th international sysposium on process systems engineering. New York: American Institute of Chemical Engineers.

Gupta, A. P., Holland, M., Siewiorek, D. P., Anayadufresne, M., Prinz, F., Nigen, J., & Amon, C. (1991). Life Cycle Concerns in Designing Computer Systems. Presented at the AIChE summer national meeting, Pittsburgh.

Harrison, D. S., Newton, A. R., Spickelmier, R. L., & Barnes, T. J. (1990). Electronic CAD frameworks. Proceedings of the IEEE, 78 (2), 393–417.

James, A. J. (1984). Specification and evaluation of computer aided engineering (CAE) systems. [Personal communications].

Jones, T. (1984). Reusability in programming: A survey of the state of the art. IEEE Transactions on Software Engineering, SE–10, 488–494.

Jones, A. K. (1978). The object model: A conceptual tool for structuring software. In R. Bayer, R. M. Graham, & G. Seegmuller (Eds.), Operting systems--An advanced course, (pp. 7–16). Berlin: Springer Verlag.

Joseph, J. V., Thatte, S. M., Thompson, C. W., & Wells, D. L. (1991). Object–oriented databases: Design and implementation. Proceedings of the IEEE, 79 (1), 42–64.

Joyce, E. J. (1988). Reusable software: Passage to productivity. Datamation, 34 (18), pp. 97–100.

Katz, R. H. (1985). Information management for engineering design. New York: Springer–Verlag.

Kinnes, C. C., & Kappes, K. K. (1992). Protecting computer software--your legal

rights and responsibilities. Presented at the AIChE spring national meeting, New Orleans.

Kuhn, T. (1970). The structure of scientific revolution. (2nd ed.) Chicago: University of Chicago.

Love, T., (1988). The economics of reuse [of software]. In Proceedings of the thirty-third IEEE Computer Society international conference. Los Alamitos, CA: IEEE Computer Society Press.

Lyytinen, K., (1987). Different perspectives on information systems: problems and solutions. ACM Computing Surveys, 19 (1), 5-252.

Mehta, J., & Patakas, D. (1988). Database systems: Current research. St. Louis: Center for Computer Aided Process Engineering, Washington University in St. Louis.

Meyers, B. (1987). Reusability: The case for object-oriented design. IEEE Software, 4 (2), pp. 50-64.

Meyers, B. (1988). Object-oriented software construction. Englewood Cliffs, NJ: Prentice Hall.

Motard, R. L. (1987). [Personal communications].

Myers, G. J. (1978). Composite/Structured Design, New York: Van Nostrand Reinhold.

Patakas, D. (1988). Data Modeling for Process Design. Unpublished doctoral dissertation, Washington University in St. Louis.

Piela, P. C., Epperly, T. G., Westerberg, K. M., & Westerberg, A. W. (1991). ASCEND: An object-oriented computer environment for modeling and analysis: The modeling language. Computers and Chemical Engineering, 15, (1), 53-72.

Pountain, D. (1988). REKURSIVE: An object-oriented CPU. BYTE, 13 (12), pp. 341-349.

Power, L. R. (1990). Post-facto integration technology: New discipline for an old practice. In Proceedings of the first international conference on systems integration, Los Alamitos, CA: IEEE Computer Society Press.

Preece, P. E., & Stephens, M. B. (1989). PROCEDE--opening windows for design. In Computer Integrated Process Engineering, (pp 211-223). Rugby, UK: Hemisphere

Pyster, A., & Barnes, B. (1988). The Software Productivity Consortium reuse program. In Proceedings of the thirty-third IEEE Computer Society international conference. Los Alamitos, CA: IEEE Computer Society Press.

Rasdorf, W. J. (1987). Extending DBMS's for engineering applications, Computers in Mechanical Engineering, 5 (5), pp. 62-69.

Robertson, J. L. (1989). Ideal process simulator. Chemical Engineering Progress, 85 (10), pp. 62-66.

Rumbaugh, J., Blaha M., Premerlani, W., Eddy, F., & Lorenson, W. (1991). Object-oriented modeling and design. Englewood Cliffs, NJ: Prentice Hall.

Saunders, J. H. (1989). A survey of object-oriented programming languages. Journal of Object-Oriented Programming, 1 (6), 5-11.

Silberschatz, A., Stonebraker, M., & Ullman, J. (Eds.) (1991). The next-generation database systems. [Special Issue]. Communications of the ACM, 34 (10), 1110-1120.

Stefik, M. & Bobrow, D. (1986). Object-oriented programming: Themes and variations. Artificial Intelligence Magazine, 7 (1), pp. 40-62.

Stephanopolous, G., Johnston, J., Kriticos, T., Lakshmanan, R., Mavrovouniotis,

M., & Siletti, C. (1987). DESIGN-KIT: An object-oriented environment for process engineering. Computers and Chemical Engineering, 11 (6), 655-674.

Waligura, C. L. & Motard, R. L. (1977). Data management in engineering and construction projects. Chemical Engineering Progress, 73 (12), pp. 62-70.

Winograd, T. (1979). Beyond programming languages. Communications of the ACM, 22 (7), 391-401.

Yamashita, Y., & Motard, R. L. (1986). Object-oriented integration in process engineering computation. Paper presented at the AIChE Spring National Meeting, New Orleans.

Yamashita, Y. (1986). Object-oriented integration in process engineering computation. Unpublished doctoral proposal, Washington University in St. Louis.

Yamashita, Y. (1987). VSM 1.1 User's Manual. Department of Chemical Engineering, Washington University in St. Louis.

Glossary

This glossary consists of two sections. The first section consists of abbreviations, acronyms, and titles only. The second section briefly describes various terms of the REO methodology.

Abbreviations, Acronyms, and Titles

ASCEND. An acronym for Advanced System for Computations in Engineering Design. This software system is developed by the Engineering Design Research Center at Carnegie Mellon University, Pittsburgh, Pennsylvania.

ASPEN. An acronym for Advanced System for Process Engineering, a chemical process modeling and simulation system.

AP. An abbreviation for ASPEN's Physical Property Subsystem.

CAE. An abbreviation for Computer Aided Engineering.

CAPE. An abbreviation for Computer Aided Process Engineering.

CAD. An abbreviation for Computer Aided Design.

CAD/CAE. An abbreviation for Computer Aided Design or Computer Aided Engineering.

CAD/CAM. An abbreviation for Computer Aided Design or Computer Aided Manufacturing.

CFI. An acronym for CAD Framework Initiative. This is the name of a system architecture for integration of CAD tools in electronic CAD.

DBMS. An abbreviation for Data Base Management System.

DESIGN-KIT. Title of a software system developed in the Laboratory for Intelligent Systems for Process Engineering at Massachusetts Institute of Technology, Cambridge.

DELI. An acronym for Design Environment for Leonardo Investigators and Inventors.

FORTRAN. An acronym for FORMULA TRANSLATION, a programming language used mainly for writing scientific and engineering programs.

ICAE. An abbreviation for Integrated Computer Aided Engineering.

ICAPE. An abbreviation for Integrated Computer Aided Process Engineering.

IEEE. An abbreviation for the Institute for Electrical and Electronics Engineers, a professional society.

IPAD. An acronym for the project titled Integrated Program for Aerospace Vehicle Design. This project was undertaken by a group of leading aerospace and CAD/CAM companies in the U.S.

MCC. An abbreviation for Micro Electronics and Computer Technology Corporation. This is a consortium of U.S. companies in the computer industry, and is located in Austin, Texas.

OMT. An abbreviation for Object Modeling Technique. This is a methodology developed at General Electric, Corporate Research & Development.

PDF. An acronym for Problem Data File. A Problem Data File is a file that stores data for a specific simulation problem; it is created and managed by the ASPEN system.

PID. An abbreviation for Piping and Instrumentation Diagram.

PP. An abbreviation for Physical Property Subsystem. This subsystem of ASPEN is used for various computations concerned solely with thermophysical properties.

Proto-ICAPE Project. Title of the research project that is the subject of this dissertation. It is so named to indicate that it is a prototype ICAPE system.

PROCEDE. A software system for process design developed at the University of Leeds, U.K.

REO. Acronym for Reuse for object-orientation, a methodology that is described in this dissertation.

REO-TGS. An object-oriented model derived from the TGS subsystem of ASPEN by following the REO methodology.

TGS. An abbreviation for Table Generation System, a subsystem of the ASPEN system, for generating tables of thermophysical property data for various mixture of chemical components.

VLSI. An abbreviation for Very Large Scale Integrated as in VLSI circuits.

VSM. An abbreviation for Virtual Stack Machine. This is a software system, an object-oriented programming environment, that is used for implementation of Icape-91 system, a prototype ICAPE, in this research.

Terminology of REO Methodology

CODE. A method to derive object-oriented model that involves direct reuse of code or compiled program unit. (See page 51.)

Code. A program unit in object language generated by a compiler.

Compiled form. The form of a program unit that is generated by a compiler; it is expressed in an object language.

Cover. A software system that is subjected to software reuse following the REO methodology is said to be covered. (See page 40.)

DOCU. A method to derive object-oriented model from the descriptions in the manual. (See page 57.)

LANG. A set of methods to derive object-oriented model from language descriptions. (See pages 42, 43.) The set consists of only the LANG method.

Also, a method to derive object-oriented model from the syntax specifications of the context-free grammar of a programming language. (See pages 43–47.)

PROG. A set of methods to derive object-oriented model from program descriptions. (See page 47.)

Program unit. A unit of a program that is executable.

SIMP. A set of methods to simplify derived object-oriented models in REO. It consists of SIMP-1, SIMP-2, SIMP-3, SIMP-4, and SIMP-5 methods. (See pages 58–61.)

SIMP-5. A method to simplify an object-oriented model that is derived by following the REO methodology. In this method, extraneous classes are eliminated. (See page 61.)

SIMP-4. A method to simplify an object-oriented model that is derived by following the REO methodology. In this method, equivalent classes are eliminated. (See page 60.)

SIMP-1. A method to simplify an object-oriented model that is derived by following the REO methodology. In this method, a class attribute that serves solely to identify uniquely an instance of the class is dropped. (See page 59.)

SIMP-3. A method to simplify an object-oriented model that is derived by following the REO methodology. In this method, a class with no attributes is dropped; instead, one uses simple integral constants or enumerated data types. (See page 60.)

SIMP-2. A method to simplify an object-oriented model that is derived by following the REO methodology. In this method, a class with only one attribute is eliminated. (See page 59.)

SORC. A method to derive object-oriented model from the source form of the program unit. (See page 54.)

Source. The source form of a program unit.

Jaimin A. Mehta

VITA

Date of Birth:	9/29/61
Place of Birth:	Baroda, India
Undergraduate Study:	Indian Institute of Technology Bombay, Bombay B. Tech. 1984
Graduate Study:	Washington University, St. Louis, Missouri, 1985-present D. Sc. expected May, 1992
Professional Societies:	IEEE Computer Society American Institute of Chemical Engineers
Honors/Awards:	Graduate Fellowship, Washington University, 1986-present Runner-up in state-wide Ramanujan Math Olympiad, 1979
Scholastic and Professional Experience:	Graduate Research Assistant, Washington University, 1986-present Teaching Assistant, Washington University, 1985-1986, 1990-1991 Assistant Engineer, Indian Organic Chemicals Limited, Madras, India, 1984-1985

Publications:

[1] M. R. Blaha, J. A. Mehta and R. L. Motard, "Structure and Methodology in Engineering Information Management," for the AIChE Los Angeles Annual Meeting, November 17-22, 1991.

[2] J. A. Mehta, Y. Yamashita and R. L. Motard, "Integration Technology for Process Design and Simulation," for the AIChE Orlando Spring National Meeting, March 18–22, 1990.

[3] J. A. Mehta, Y. Yamashita and R. L. Motard, "Object-Oriented Modeling and Simulation," for the AIChE Houston Spring National Meeting, April 2–6, 1989.

[4] J. A. Mehta, D. Patakas and R. L. Motard, "Data Management in Computer Aided Engineering," for the AIChE New Orleans Spring National Meeting, March 6–10, 1988.

[5] J. A. Mehta and D. Patakas, "Database Systems: Current Research," Technical Report, Center for Computer Aided Process Engineering, Washington University, Saint Louis, Missouri, January 1988.

[6] J. A. Mehta, D. Patakas and Y. Yamashita, "A Relational Data Model for Aspen flowsheeting system," Technical Report, Center for Computer Aided Process Engineering, Washington University, Saint Louis, Missouri, October 1987.

May, 1992