AFIT/DS/ENG/97-04

Extracting Functionally Equivalent Object-Oriented Designs from Legacy Imperative Code

DISSERTATION

Ricky E. Sward
Major, USAF

AFIT/DS/ENG/97-04

AFIT/DS/ENG/97-04

Extracting Functionally Equivalent Object-Oriented Designs from Legacy Imperative Code

DISSERTATION

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy in Computer Engineering

Ricky E. Sward, B.S.C.S, M.S.C.S, M.S.I.S.

Major, USAF

September, 1997

AFIT/DS/ENG/97-04

Extracting Functionally Equivalent Object-Oriented Designs from Imperative Legacy Code

Ricky Eugene Sward, B.S.C.S, M.S.C.S., M.S.I.S.

Major, USAF

Approved:

Thomas C. Hartrum	14 Aug 97
Dr Thomas C. Hartrum, Chairman	
Robert P. Graham Jr	14 Aug 97
Dr Robert P. Graham, Jr.	
[illegible]	14 Aug '97
Dr Aihua W. Wood	
[illegible]	14 Aug 97
Dean's Representative	

[illegible]

Dr Robert A. Calico, Jr.
Dean, Graduate School of Engineering

Acknowledgements

Thank you, Renee, for the dedication and devotion you have shown me during this process. I love you. I could not have done this work without your support on the home front. Thank you, Amanda, for the fun and games you seem to come up with endlessly. I can always count on you for a break. You wan'na play "cups"? Thank you, Mom and Dad, for always being there for me. Thanks for your inspiration and support. Thank you, Dr Hartrum, for your guidance and levity. We do make a good team. The things I forget, you remember and vice versa. Thank you, Paul, for your guidance on my "big picture". That sure is a big pencil. Thank you, Robert, for all the "focus checks". I think we found the perfect way to spoil a trip to the beach. Thank you, Dr Wood, for your guidance and encouragement. I appreciate it. Thanks also goes to the KBSE research group: Dr Hartrum, Paul Bailor, Mark Gerken, Frank Young, Scott DeLoach, Robert Graham, Jerry Nutter, and Tom Schorsch. Thanks for all the bullets. It's always best to be told you're an idiot by your best friends. Thanks, Jerry, for being a bud. Best of luck with everything. Did you ever find your LaTeX book? I have an extra one in my car if you need it. To the guys in Room 231, good luck and keep that popcorn going! Thanks, Jimbo, for the rocket diversions. Sure hope the Big Kahuna fits in my van. Thanks, Cookie, for the trip to Salt Lake City. Squatters was great, but I could have done without Antelope Island. If I've told you once, I've told you a thousand times...stop exaggerating. Finally, I'd like to thank and praise the Lord Jesus Christ for guiding me through these years. Thank you for making me part of your plan and making my way perfect!

Ricky E. Sward

God is my strength and power:
And He maketh my way perfect.
2 Samuel 22:33

Call to me, and I will answer you,
And show you great and mighty things,
Which you do not know.
Jeremiah 33:3

Table of Contents

Page

Page

List of Figures

Figure Page

List of Tables

AFIT/DS/ENG/97-04

Abstract

The research presented in this document defines a methodology for automatically extracting functionally equivalent object-oriented designs from legacy imperative programs. The Parameter-Based Object Identification (PBOI) methodology is based on fundamental ideas that relate programs written in imperative languages such as C or Cobol to objects and classes written in object-oriented languages such as Ada 95 or C++. The fundamental thesis that defines the PBOI methodology is that object attributes manifest themselves in imperative subprograms as data items passed between subprograms. The PBOI methodology converts each subprogram in an imperative design into a class and a method that implements the subprogram. During this process duplicate classes are eliminated, duplicate objects are identified, and overlapping classes are merged into one class. Transformations have been developed that formalize the PBOI methodology and a formal proof is provided showing the extracted object-oriented design is *functionally equivalent* to the legacy imperative system. To focus the task of re-engineering, generic models of imperative programming languages and object-oriented programming languages have been developed. The Generic Imperative Model (GIM) is programming language independent, programming construct independent, and canonicalizes simulated control flow constructs. Formal definitions for the semantics of each GIM construct have been defined using the *weakest precondition* notation. The Generic Object-Oriented Design model (GOM) is also programming language independent and programming construct independent. Formal definitions for the semantics of each GOM construct have also been defined using the *weakest precondition* notation. The formal tranformations convert imperative subprograms represented in the Generic Imperative Model into classes and objects represented in the Generic Object-Oriented Design Model. A taxonomy of imperative subprograms has also been developed which classifies all imperative subprograms into one of six categories. A proof of concept prototype has been developed and a 3000-line FORTRAN-77 system has been converted to an object-oriented design as a feasibility demonstration.

Extracting Functionally Equivalent Object-Oriented Designs from Legacy Imperative Code

I. Introduction and Background

1.1 Introduction

1.1.1 Overview.

The *object-oriented* paradigm with its promise of re-usability, extensibility, and maintainability has great appeal to organizations with aging legacy systems. Legacy systems are often complex, unstructured, and include no documentation. Making even the smallest change to a legacy system often creates unpredictable side-effects. Legacy systems are valuable assets to an organization and should be preserved, not thrown away [49]. Re-engineering imperative legacy systems into object-oriented systems provides a way for organizations to modernize their aging systems without losing the investment that these systems represent [50, 67].

The research fields of *re-engineering* and *reverse engineering* have recently emerged within the field of software engineering. The goal of software engineering is to improve the products and practices used to develop software. However, the majority of the software development effort is spent on maintaining legacy systems as opposed to developing new systems [61]. Boehm [8] estimates the proportion of resources and time devoted to maintenance ranges from 50% to 80%. The implication is that in order to improve the software development process, the software maintenance process must be examined. Since maintaining an object-oriented system is inherently easier than maintaining an imperative system [32, 33], re-engineering legacy code to the object-oriented paradigm improves the maintenance process. Furthermore, re-engineering to the object-oriented paradigm is desirable to organizations because it improves the overall software development process.

The research presented in this document defines a methodology for automatically identifying objects from legacy imperative programs. The Parameter-Based Object Identification (PBOI) methodology is based on fundamental ideas that relate programs written in imperative languages such as C or Cobol to objects and classes written in object-oriented languages such as Ada 95 or C++. Transformations have been developed that formalize the methodology and a proof is provided showing the extracted object-oriented design is *functionally equivalent* to the legacy system. To focus the task of re-engineering, generic models of imperative programming languages and object-oriented programming languages have been developed. The formal transformations convert imperative subprograms represented in the imperative model into classes and objects represented in the object-oriented model. A taxonomy of imperative subprograms has also been developed which classifies all imperative subprograms into one of six categories.

The remainder of this chapter defines re-engineering terms in Section 1.2 and presents related work in the field of re-engineering in Section 1.3. A summary of research contributions is provided in Section 1.5 and brief descriptions of the following chapters is presented in Section 1.6. The author assumes the reader has fundamental knowledge about both the imperative paradigm [16] and the object-oriented paradigm [62].

1.2 Definitions

The following definitions are provided for the terms *re-engineering*, *reverse engineering*, and *program understanding.*

1.2.1 Re-Engineering. According to Chikofsky [13],

> *Re-engineering* is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

Typically, a system being re-engineered is written in an outdated programming language or built specifically for an outdated hardware platform. Computer systems that were built many years ago and have undergone several maintenance modifications often become what are known as *legacy systems.* A legacy system is hard to maintain because of the lack of knowledge about the system and because of the side effects when making even a simple change to the system. The computer program in a legacy system is called *legacy code.* Re-engineering often focuses on revamping legacy systems using new programming

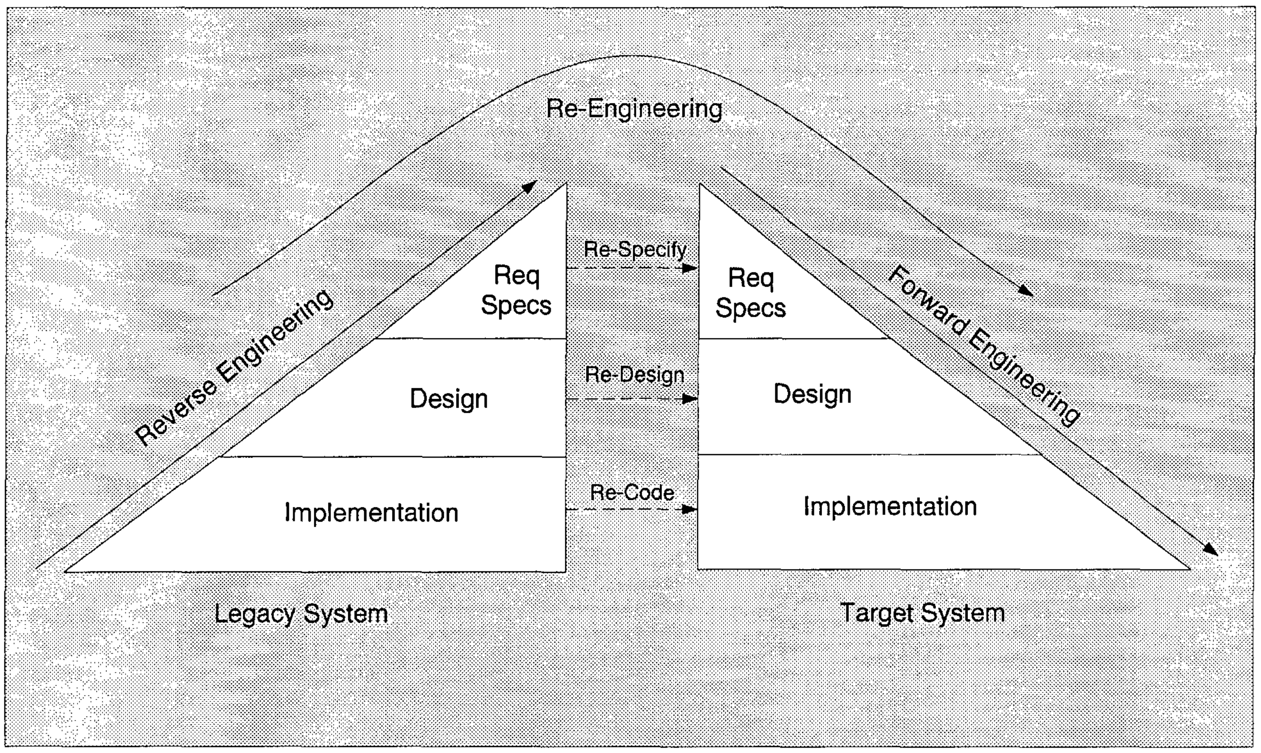


Figure 1 Re-Engineering Process

paradigms, languages, or hardware platforms. Figure 1 shows a generalized view of the process of re-engineering legacy code as developed by Byrne [12]. In order to do effective re-engineering, the legacy code must be expressed at a higher level of abstraction than the programming language in which it was written [75]. This process of expressing the legacy code in a higher level of abstraction is *reverse engineering* as shown on the left hand side of Figure 1. The different levels of abstraction in re-engineering include *implementation*, *design*, and *requirements specifications.*

Legacy code can be re-engineered at each of these levels of abstraction. At the implementation level, it is possible to *re-code* a program from one programming language to another. At the design level, it is possible to *re-design* a program changing the design of the legacy code into a design for the target system. At the requirements specification level, it is possible to *re-specify* the requirements for a program. Each of these processes is discussed in more detail in Section 1.2.2.

Once these transformations are done at the different levels of abstraction, *forward engineering* can be used to build the target system in the new paradigm or language. Re-engineering is the overall process of reverse engineering followed by forward engineering. This process does not require the target system to be functionally equivalent to the legacy system.

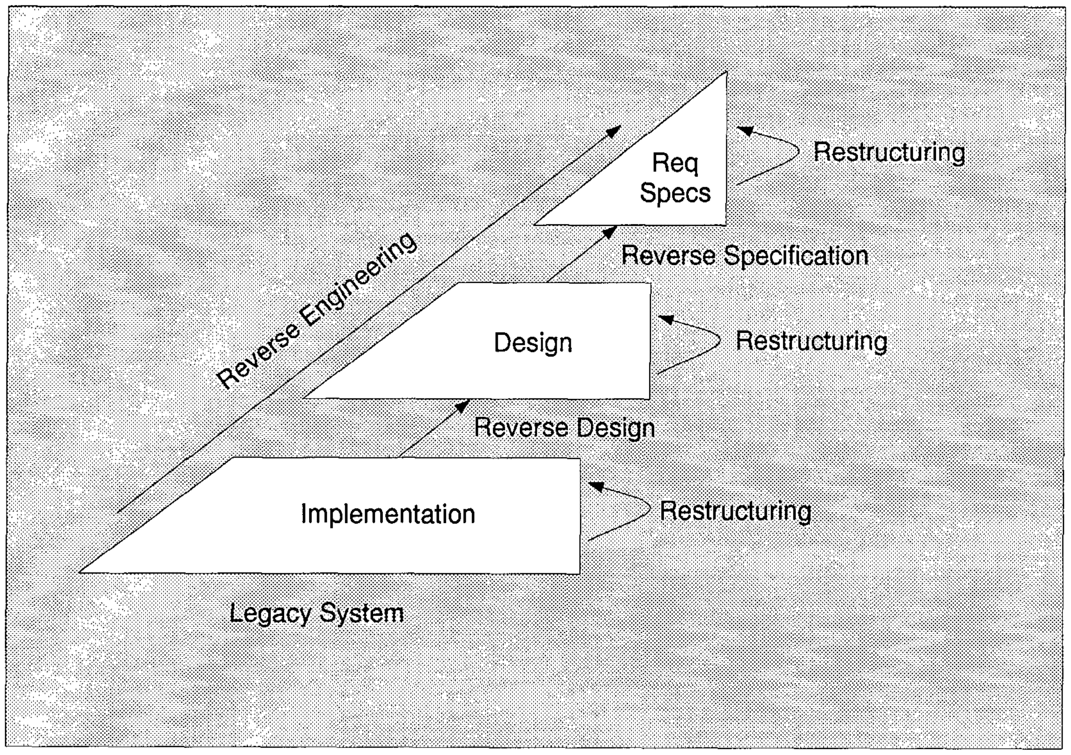


Figure 2   Reverse Engineering Process

1.2.2 Reverse Engineering. Figure 2 shows the process of reverse engineering in more detail. To abstract legacy code, *implementation* information must be available. This is typically the programming language code such as FORTRAN or Cobol code. It is possible to *restructure* the implementation information during reverse engineering without re-engineering the legacy code to a new target system. A restructuring at the implementation level could possibly include eliminating `GOTO` statements from the legacy code.

The process of *reverse design* abstracts the implementation information up to the design level. This process extracts information such as structure charts showing the calling hierarchy of the legacy code, data flow diagrams showing the flow of data in and between legacy code routines, or control flow diagrams showing the flow of control for the legacy code. Restructuring can also be done at the design level without doing re-engineering. For example, the structure chart could be reorganized to improve the number of connections between different legacy code routines, i.e., improve the *coupling* [64] between routines.

The design level information is abstracted up to the requirements specification level by the *reverse specification* process. This process extracts the specifications for the legacy code from the design information. Restructuring can also be done at this level of abstraction. For example, the specifications could be split apart and reorganized to improve understandability.

Often in reverse engineering, a system undergoes a program understanding process in order to create a more robust and understandable abstraction of the program. *Program understanding* is a special form of reverse engineering. An overall view of program understanding is presented in the following section.

1.2.3 Program Understanding. This section describes the current cognitive science notion of human program understanding as found in Tiemens [73]. Figure 3 shows an

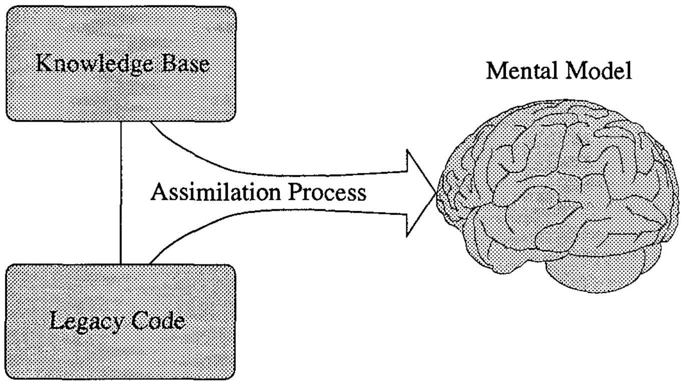


Figure 3 Program Understanding Theory

overall notion of how humans understand programs. The *knowledge base* stores concepts the person already understands such as analysis and design techniques, programming architectures, specific algorithms, and specific programming language constructs. This knowledge ranges from abstract notions such as architecture to concrete notions such as assignment statements in FORTRAN.

The *legacy code* is the programming language code the person is trying to understand. This code is typically in the form of a listing the person can examine. There is typically no pre-processing of this code before the person examines it.

The *assimilation process* involves the person looking at the code and trying to find concepts in the knowledge base implemented in the legacy code. This process is aptly called "assimilation" because the person is trying to fit the processing being done by the legacy code into the knowledge he or she already has in the knowledge base. The process builds new knowledge that is linked to the knowledge base and stored in the mental model.

The *mental model* is a series of links created in the assimilation process by the person understanding the legacy code. The mental model is the person's understanding of how the processing of the legacy code fits into what he or she already knows. The links between the knowledge base and the legacy code are rich semantic links from what was known before to what is being understood now. Understanding is a process of recognition, abstraction, assimilation, and storage [73].

This overall view of program understanding is by no means complete because the area of program understanding is an open field in cognitive science. However, the ideas presented here are an adequate presentation of the current literature in this field.

1.3 Related Work

This section reviews previous work in the area of reverse engineering. This review includes significant contributions in program understanding, extracting software architectures, extracting objects, and maintaining functional equivalence. The overall landscape of these contributions is presented below followed by detailed descriptions of selected systems.

Research contributions thus far in the area of reverse engineering can be loosely categorized based on three criteria:

Knowledge The level of the knowledge extracted from the legacy code.
Strategy The approach used for reverse engineering.
Goal The end result or product.

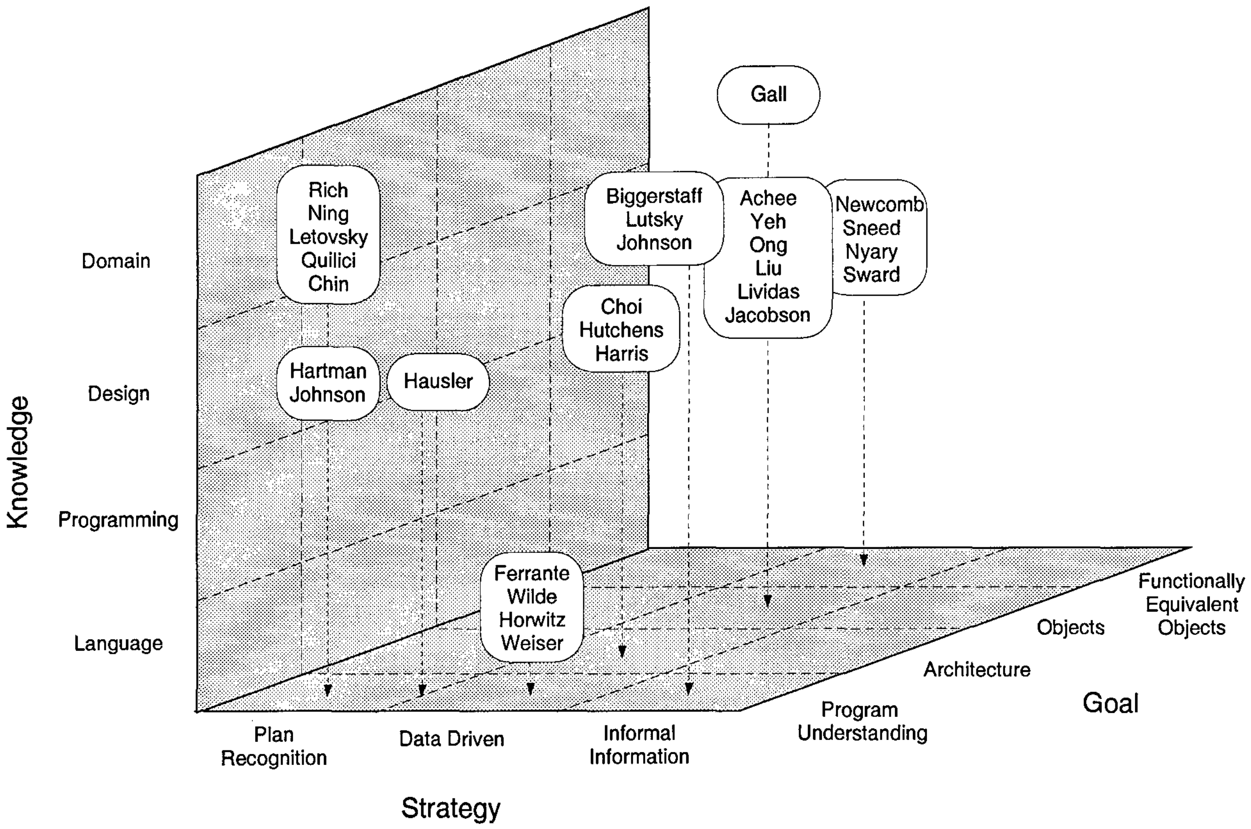


Figure 4 Reverse and Re-Engineering Systems

Figure 4 shows these criteria on three axes. The axis labeled **Knowledge** shows the different levels of knowledge that can be extracted. *Language knowledge* is low-level knowledge about a programming language such as FORTRAN or Cobol. *Programming knowledge* includes knowledge about common ways to implement algorithms such as sorting or searching. *Design knowledge* is knowledge about the architecture or overall design of a legacy code. *Domain knowledge* is knowledge about the application domain being implemented by the legacy code. The axis labeled **Strategy** shows the different strategies used in reverse engineering. *Plan Recognition* strategies rely on pre-defined plans or clichés that describe the concepts to be recognized. *Data Driven* strategies rely on information from the legacy code such as data-flow, control-flow, and program slices in order to extract knowledge. *Informal Information* strategies extract information from documentation and comments from the legacy code.

The axis labeled **Goal** shows the goals of the reverse engineering systems. *Program Understanding* systems produce a mapping between concepts and the legacy code which represents an understanding of the legacy code. The *Architecture* goal refers to systems that recover the architecture used in the legacy code. This ranges from low-level products such as module interconnection diagrams to high-level products such as recognizing a client-server architecture. The *Objects* goal refers to systems that recover objects and classes from the legacy code. Systems in this category recover as many objects as possible, but are not focused on duplicating the functionality of the legacy code with these objects. The *Functionally Equivalent Objects* systems recover objects and classes with the specific goal of maintaining the functionality of the legacy code.

1.3.1 Plan Recognition Systems. Work in the area of program understanding has been dominated thus far by the plan (or cliché) recognition systems [28,48,51,60,70]. Plan recognition systems were inspired by forward engineering systems based on plans including the Programmer's Apprentice [56–59, 74]. Plan recognition systems aid in understanding legacy code by matching the code to a pre-defined *plan* with concepts and constraints between the concepts. Once the plan is found in the code, the concept being represented by the plan is considered to be understood. These plans can be organized into a hierarchy representing low-level programming knowledge up to high-level domain knowledge.

Figure 4 shows where plan recognition systems fit in the overall landscape of re-engineering systems. As shown in the figure, some plan recognition systems (Wills, Ning, etc.) can recognize domain knowledge while others (Hartman and Johnson) recognize design knowledge. The goal of plan recognition systems has, to this point, been program understanding.

1.3.1.1 Rich's Work [60]. The Recognizer system built by Rich and Wills automatically finds all occurrences of a concept in a program. This system relies on the idea of clichés for concept recognition and uses the *plan calculus* [57] to represent programs and clichés. The Recognizer builds a hierarchical representation of the concepts it finds in the programs and associates a natural language description with each of the concepts recognized.

Rich addresses five problems in legacy code that are handled by using clichés to represent knowledge:

Syntactic variation The same net flow of data and control can be achieved in many ways.

Non-contiguousness Parts of a cliché can be scattered through the program text; they do not have to be contiguous.

Implementation variation An abstraction can be implemented in many ways.

Overlapping implementation Program optimization can merge the implementations of distinct abstractions.

Unrecognizable code It is hard to find "good" concepts in code if the code was not built from a template such as a cliché.

The Recognizer system transforms programs into the plan calculus to overcome the problems of syntactic variation and non-contiguousness. The Recognizer also includes a hierarchy of clichés represented in the plan calculus. By representing both programs and clichés in the plan calculus, Rich has made the matching process much simpler. The process of matching clichés to programs is treated as a graph parsing problem. The Recognizer demonstrates understanding of the program by producing documentation automatically. It does this by using generic natural language templates stored with each cliché and filling in the appropriate fields from the code. The documentation has an artificially formal flavor to it, but demonstrates shallow understanding of design decisions made in producing the code.

Overall, Rich's research raises some important issues. First, for program understanding, there must be some representation of the concepts to be recognized. Second, the representation of concepts must be compatible with the representation of the program. Using the plan calculus to represent both the program and the clichés is a good common representation for the Recognizer. Using the plan calculus overcomes the problem of syntactic variation and non-contiguousness. It also casts the problem as a graph parsing problem which taps into a rich resource for problem solving. Rich calls this approach a *representation shift.*

1.3.1.2 Ning's Work. Ning [26] developed the Program Analysis Tool (PAT) to aid program understanding. The system uses a plan hierarchy and rule-based inferencing system to recognize low-level constructs in the code and combine these to form higher-level abstract concepts. Ning [39] discusses automating the process of concept recognition by transforming programs to different levels of abstraction:

Text-level transformations treat a program as a sequence of characters and transform the program using character substitution.

Syntactic-level transformations treat programs as abstract syntax trees and transform programs using rules about the abstract syntax descriptions.

Semantic-level transformations add meaning to the syntax of programs and transform programs using the semantic information found in control flow and data flow diagrams.

Concept-level transformations rely on knowledge of programming, problem solving, and application domains to transform programs at the conceptual level.

Ning [39] also discusses the general problem of concept recognition. Programs contain many different kinds of information which can be divided into *language* concepts and *abstract* concepts. Language concepts are the low-level syntactic programming concepts from the legacy code. They include modules, language statements and other concepts defined by the programming language syntax. Abstract concepts refer to language-independent concepts such as programming concepts and problem solving methods such as searching and sorting.

A programming language compiler recognizes language concepts and builds them into an Abstract Syntax Tree for a program. To recognize abstract concepts, a concept classification hierarchy called a *concept model* is used to define the knowledge to be recognized. This model will not be complete since abstract theories of general programming are hard to develop. Thus, a *partial recognition* of the program is possible. This means it may not be able to recognize the program as representing a single concept, but it may be possible to recognize *islands* of concepts in the program. The concept model also defines the attributes of the concepts to be recognized. Ning develops the idea of a *plan* to represent a concept. A plan contains components (or sub-concepts) of the concept and the constraints between the sub-concepts. This information is encoded in the form of concept recognition rules. Plans represent the abstract concepts to be recognized, and through the constraints of the plans, they represent how these concepts will be recognized.

Ning's work is important because it exemplifies plan recognition systems. In any program understanding system, all levels of abstraction need to be recognized. Ning shows how a hierarchy of concepts can be recognized using a hierarchy of plans. Ning's work was the basis for continued work in plan recognition by Letovsky [41], Quilici [51], Chin [52], and others as discussed in the next section.

1.3.1.3 Other Plan Recognition Systems. The Cognitive Program Understander (CPU) system by Letovsky [41] uses plans as correctness preserving transformations for rewriting programs into semantically equivalent yet more abstract representations. Quilici [51] enhanced Ning's plan hierarchy to include indices into the plan hierarchy, a specialization mechanism, and links to other related plans. These improvements were intended to speed up the plan recognition process. The DECODE system from Chin [52] builds up a concept base through interaction with the programmer. DECODE uses an enhanced plan recognition system to recognize as much of the legacy code as possible. It increases understanding of the program by allowing the programmer to enter new conceptual design primitives and linking them to the program. The UNPROG system by Hartman [28] uses plans to recognize control designs such as *read-process* loops in Cobol programs. The PROUST system from Soloway and Johnson [70] uses plans in a top-down method in order to understand novice programmer's code.

Overall, the plan recognition systems are limited by the number of plans required to fully understand legacy code. If the exact plan that represents the concept being implemented by the legacy code is not present in the plan hierarchy, the plan recognition system is not able to understand the legacy code. This implies large plan hierarchies are required to understand even simple programs. The plan recognition process does not scale up well to large production-level legacy code. Quilici and Chin write [52]:

> Applying this paradigm to reverse engineering real-world legacy systems is problematic, as it appears to require enormous libraries of code patterns, relies on program-understanding algorithms that are not guaranteed to scale, and fails completely on existing idiosyncratic code that does not fit well into patterns [17, 40, 69].

The plan recognition systems suffer from problems that limit their applicability to real-world systems [52]. The methodology presented in this document does not use libraries of code patterns. Instead, it uses a data-driven approach similar to the systems described in the next section.

1.3.2 Data-Driven Systems.

Plan recognition systems are contrasted with the data-driven systems as shown in Figure 4. Data-driven systems do not rely solely on the process of matching plans for recognizing concepts in legacy code. Some data-driven systems use low-level knowledge about the legacy code to limit the amount of code being considered when matching plans to the code. In general, data-driven systems use data-flow and control-flow dependencies, program slicing, and definition/usage information to extract design and programming knowledge. The data-driven systems have varying reverse engineering goals including extracting architectures and objects. The following sections present relevant data-driven systems.

1.3.2.1 Newcomb's Work [47].

This section describes Newcomb's work on recovering functionally equivalent collections of objects from legacy code. The following terms are defined by Newcomb and help explain the process.

Scope analysis relates the occurrence of a variable to its declaration.

Program unit analysis describes the signature of routines, i.e., the number, order, and type of parameters for all routine declarations and invocations.

Collision former is a unique template of a data structure based on record length, field offset, field length and field type.

Alias analysis is the process of finding the names of the records and subfields in legacy code that match the collision former.

Alias map represents a relation whose domain is a collision former and whose range is a set of records that match the collision former. The criteria for matching the collision former can be changed.

Program slice is computed using the transitive closure of a *usage* to *definition* arc with respect to a variable.

Newcomb uses the idea of a collision former to hold the low-level declarative knowledge in his system. Each Cobol `01` level record from the legacy code becomes a collision former. The collision formers do not describe algorithms or designs, as would a plan. They hold the structure of the data in the system. The procedural knowledge is stored in control flow and data flow diagrams derived from the code.

The collision formers are used to do alias analysis, i.e. used as patterns to search through the other record structures for matches Newcomb calls *aliases.* Alias analysis is done on the data items in the legacy code instead of on the routines in the code (as in plan recognition systems). Newcomb creates an alias map that maps from each collision former to the set of record definitions that match the collision former.

Classes in the object model are formed by alias analysis and collision formation. The classes are built from a generalized version of the set of record definitions, and the instances of the classes are the records in the set. The fields from the 01 records are assigned to the objects as attributes. Currently, Newcomb's object modeling process does not construct class hierarchies.

Once objects are found through alias analysis, Newcomb takes a program slice focusing on one object. This indicates what procedures in the code modify or use the object. Newcomb presents a thorough explanation of how these procedures are built as methods and assigned to classes of objects.

Overall, Newcomb's system is important because of the work in extracting functionally equivalent objects. Newcomb claims the Object-Oriented Model (OOM) extracted is *functionally faithful* to the legacy code, but does not provide a proof of this claim.

1.3.2.2 Sneed's Work. The system developed by Sneed [67] is a pre-cursor to Newcomb's system [47] and also extracts functionally equivalent objects from legacy code. Sneed uses Cobol record structures to extract objects. Every record type is identified as an object and every field as an object attribute. The methods for these objects are extracted by using a program slicing technique Sneed calls *phasing.* A phase is an executable sequence of statements that follows the data flow path starting at an input from a file and ending at an output to a file. These program slices are attached to the objects to which they refer and become methods in the class that describes these objects.

An object-oriented specification is built for each extracted object. The specification language used is Z++, an object-oriented version of the Z specification language. At the end of the reverse engineering process, there is a specification for each method and the imperative code is distributed among the objects in the target system. Sneed claims the objects extracted are functionally equivalent to the legacy code, and uses *dynamic profiling* to validate this claim. Dynamic profiling is a technique that inserts statements in the legacy code to print out the values of variables at certain points in the code. The values of the variables are collected during execution of the legacy system. The values of variables in the re-engineered system are also collected during execution of the re-engineered system. The systems are functionally equivalent if the values for each of the variables in the legacy system are equal to the values for that variable in the re-engineered system.

The system from Sneed is an interesting data-driven system because of the program slicing and the extraction of functionally equivalent objects. The use of dynamic profiling requires execution of both the legacy system and the extracted object system, which limits the practicality of Sneed's proof of functional equivalence. Furthermore, errors in the transformation of the legacy system are not discovered until the re-engineered system is executed.

1.3.2.3 Nyary's Work. The system described by Nyary [68] automatically extracts object-oriented design documentation from legacy code. Nyary claims the design extracted is functionally equivalent to the legacy code, but does not prove this claim. The first step in Nyary's approach is to identify several different types of objects including user interface, information, file, record, view, work, and link objects. Nyary gives heuristics for finding each type of object. The seven types of objects are identified and stored with the names and descriptions of their attributes.

The system extracts operations for these objects by using data-flow and control-flow information. The connections between objects are identified by examining the parameters of routines in the legacy code and identifying any foreign variables in the object operations. *Foreign variables* are variables used by an operation that are not included in the object associated with the operation. Nyary claims the extracted operations are functionally equivalent to the legacy code. The sequence of these operations is expressed as a state driven model (as defined by Shlaer and Mellor [63]).

Nyary presents another example of a system that claims to extract functionally equivalent objects. Nyary does not provide any formal proof of this claim.

1.3.2.4 Achee and Carver's Work. The methodology developed by Achee and Carver [1] uses statistical analysis of subroutine parameters to extract objects from legacy FORTRAN systems. The frequency with which pairs of parameters are used in different subroutines is used to construct the "strongest cohesive unit" [1]. Objects are extracted by grouping these data items together (to form the cohesive unit) and then attaching methods to these objects.

Achee and Carver do not claim to extract functionally equivalent objects from legacy code. Their methodology is interesting because it uses subroutine parameters to extract objects using a statistical method rather than the methodology presented in this document. Their analysis focuses more on how parameters are used in a single subroutine than on the parameters passed between several subroutines.

1.3.2.5 Liu, Lividas, and Johnson's Work. Lividas and Johnson [43] present three techniques that use data-driven methods to extract objects from legacy code. The first two techniques were developed by Liu [42]; Lividas and Johnson introduce a third. Lividas and Johnson express all three methods for object identification using formal definitions. These definitions are presented below.

Let P be an imperative program, F the set of all routines in P, T the set of all types in P, and D the set of all data items in P. A *candidate object* [43] is a triple $C_M^P = (\varphi, \tau, \delta)$ where $\varphi \subset F$, $\tau \subset T$, $\delta \subset D$, and M indicates the method used for object identification.

Global-based Object Identification (GBOI) defines a triple $C_g^P = (\varphi, \emptyset, x)$ where $\varphi \subset F$ and x is a global variable with respect to φ.

Type-based Object Identification (TBOI) defines a triple $C_t^P = (\varphi, \tau, \emptyset)$ where $\varphi \subset F$ and $\tau \subset T$. Here,

$$\tau = \bigcup_{i=1}^{n} \{a_i\} \ \cup \ \{b\}$$

where a_i represent the types of the parameters and b the type of the returned variable (if any).

Receiver-based Object Identification (RBOI) defines a triple $C_r^P = (\varphi, \tau, \emptyset)$ where $\varphi \subset F$ and $\tau \subset T$. Here,

$$\tau = \bigcup_{i=1}^{n} \{r_i\} \ \cup \ \{b\}$$

where r_i represent the types of the parameters that are changed in φ.

These three techniques identify candidate objects in imperative programs based on relationships between the data items.

1.3.2.6 Other Data-Driven Systems. Figure 4 shows several other data-driven systems. These systems are described briefly in this section. The system proposed by Hausler [29] is a hybrid of the plan recognition systems and the data-driven systems. Hausler uses program slicing to isolate pieces of legacy code and then uses pattern matching to recognize plans in the program slices. This is interesting because the knowledge extracted by data-driven means is being used to limit the amount of code considered when matching the plans to the code. The systems from Ferrante [20] and from Wilde [77] capture dependencies in legacy code. Horwitz [30] and Weiser [76] present data-driven methods for program slicing. These four contributions provide building blocks for data-driven reverse engineering tools. The knowledge being represented is at the language level. The systems presented by Choi [14] and by Hutchens [31] use data-driven methods to analyze system structure of legacy code. Choi presents a method for extracting various module interconnection diagrams based on various criteria. The system built by Harris [27] uses program slicing and recognition rules to recognize architectural aspects of legacy code. The *recognizers* used are similar to plans because they structure the knowledge being recognized, but the recognizers do not require an exact match in the legacy code. Program slicing is used to limit the amount of code considered when matching recognizers to code.

The system proposed by Gall [21] uses data flow diagrams and structure charts to cluster procedures and data together into objects. Semantic knowledge is added to the objects found by comparing them to an independently developed object-oriented model of the underlying system. The OBAD system built by Yeh [79] (a sub-system of the Harris system [27]) extracts Abstract Data Types (ADTs) from legacy code. This system first builds a graph where the procedures and the data structure types are the nodes and the references from the procedures to the internal fields of the structures are the edges. The ADTs are extracted from the connected components of this graph. This contribution is included under the *objects* rubric because of the similarity between ADTs and objects [11].

The system developed by Ong [50] extracts objects from FORTRAN code. The system uses information from the COMMON block of the FORTRAN code to organize data and procedures into objects. The work presented by Jacobson [33] describes an incremental approach for re-engineering legacy code to the object paradigm. An interface is built from the new object-oriented part of the system to the part of the system not being re-engineered.

1.3.3 Informal Information Systems. The final group of contributions extract knowledge from the informal information found in legacy systems. This information includes any available user's manuals, programmer's manuals, or testing manuals. The informal information also includes the comments in the legacy code associated with the piece of code being analyzed. The DESIRE system from Biggerstaff [4–6] uses informal information to extract domain level knowledge, as indicated in Figure 4. DESIRE uses the informal information to aid program understanding. The work done by Lutsky [45] uses informal information in legacy system documentation to generate test cases for legacy systems. The I-DOC work by Johnson [34] automatically answers user's questions about a legacy system from a hyper-media database of knowledge. I-DOC automates the informal information associated with a software system.

1.4 Summary

The data-driven systems presented in this chapter use approaches for reverse engineering that are more promising than the approaches used by plan recognition systems. Newcomb's system [47], for example, has been demonstrated on legacy systems as large as 168,000 lines of Cobol code. The data-driven systems are more scalable because they either limit the amount of code being matched against plans or they don't use plans at all.

Several approaches were presented that extract objects from legacy code. Some of these systems claim to extract functionally equivalent objects, but there are no proofs of these claims. Sneed [67] validates his claim using dynamic profiling, but this is not practical because it requires the extracted object-oriented design to be implemented. Any error introduced in the transformation would not be uncovered until the entire design was implemented.

The research presented in this document is similar to the work done by Newcomb [47], Sneed [67], and Nyary [68], as indicated in Figure 4. This research uses data-driven methods to extract an object-oriented design that includes functionally equivalent objects.

1.5 Summary of Contributions

Given the overview of re-engineering and the related work that has been done in this area, the research presented in this document makes the contributions shown below. Each of these contributions is discussed in subsequent chapters and a more detailed summary of the contributions is provided in Chapter IX.

1. An object identification methodology that extracts functionally equivalent objects from legacy imperative code.
2. A taxonomy of imperative subprograms.
3. Formal transformations that define the object identification methodology.
4. A proof of functional equivalence between the legacy imperative code and the extracted object-oriented code.
5. A canonical form for representing legacy imperative code.
6. A surface syntax for the imperative canonical form.
7. A canonical form for representing object-oriented designs and code.
8. A surface syntax for the object-oriented canonical form.
9. Definitions of the formal semantics for an imperative function call.
10. Definitions of the formal semantics for an object-oriented message.

1.6 Overview of Chapters

Chapter II defines in detail the problem of re-engineering imperative systems to the object-oriented paradigm. Chapter III presents the generic model of imperative programming languages and Chapter IV presents the generic model of object-oriented languages. Chapter V presents the taxonomy of imperative subprograms, the methodology for extracting objects from subprograms, and explains how *program slicing* [76] is used to simplify the methodology. Chapter VI presents transformations that formalize the methodology.

Chapter VII presents a proof that the extracted design is functionally equivalent to the legacy system. Chapter VIII presents a feasibility demonstration using a prototype implementation of the methodology. Chapter IX describes the contributions this research makes to the field of re-engineering. Chapter X discusses future research and presents the conclusions.

II. Problem Statement

2.1 Introduction

This chapter presents a summary of the overall re-engineering methodology developed by this research. Each part of the methodology is explained briefly here and the more significant parts are explained in greater detail in the following chapters.

2.2 The Re-Engineering Methodology

The goal of this research is to develop a methodology for re-engineering legacy systems into functionally equivalent object-oriented designs. Figure 5 shows the overall view of this methodology. In the figure, the large rectangles are groupings of processes and products within certain paradigms. The large box on the left shows the re-engineering processes and products within the imperative paradigm. The large box on the right represents the forward engineering processes and products within the object-oriented paradigm. The large box in the middle represents versions of processes and products in the object-oriented paradigm that can be manipulated automatically with a computer. The "rounded-rectangles" inside the large boxes represent products within a paradigm. The arrows between rounded-rectangles represent processes for converting one product to another. Each of the rounded-rectangles is discussed in the sections below. The dashed rectangle shows the scope of this research.

2.2.1 Legacy Code. The input to any re-engineering methodology is the collection of programming language code to be transformed. As explained in Chapter I, this code is termed *legacy code.* An assumption of this research is that the legacy code is available in some format, the least desirable of which is a program listing. The prototype developed to implement this research further assumes the legacy code can be accessed by a computer.

2.2.2 Canonical Form. The first step in the re-engineering methodology is to transform the legacy code into a *canonical form.* A canonical form allows code that does the same to look the same. A canonical form is language independent, programming

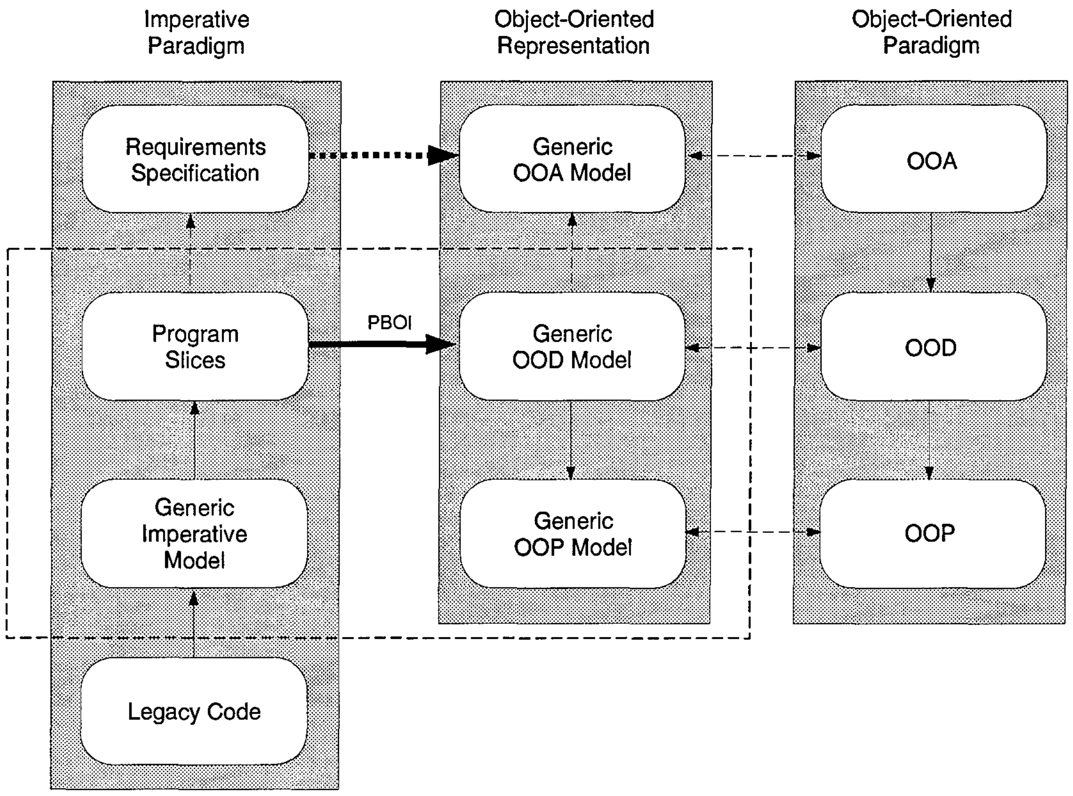


Figure 5 Re-Engineering Methodology

```
IF(ITYP.EQ.1) THEN
ALT =  -5.71D0
ELSEIF(ITYP.EQ.2) THEN
ALT = -1.57D0
ELSEIF(ITYP.EQ.3) THEN
ALT = -9.46D0
END IF
```

Figure 6 FORTRAN elseif structure

construct independent, and control flow construct independent. Each of these aspects is explained in the following sections.

2.2.2.1 Language Independent. A canonical form is *language independent*, i.e., the representation is not tied to any one specific programming language. This allows the re-engineering methodology to operate at a higher level of abstraction, encompass more diverse legacy code, and be built independent of the nuances used in the legacy code.

2.2.2.2 Programming Construct Independent. A canonical form allows different programming language constructs to be recognized as the same entity. Certain constructs can have a different *surface syntax* but provide the same control flow. These constructs are easily identified because they have equivalent control flow graphs. Wills [60] refers to this as the *syntactic variation* problem as discussed in Section 1.3.1.1. For example, Figure 6 shows the FORTRAN `elseif` construct. Not all programming languages have this control construct, so the `elseif` is converted to embedded `if-then-else` statements in the canonical form. Embedded `if-then-else` statements provide the same control flow as the `elseif` statement, thus providing the canonical form.

2.2.2.3 Simulated Control Flow Constructs. Another use of the canonical form is to recognize control flow constructs not implemented in the legacy code programming language. For example, there is no `while` loop in FORTRAN, but it can be simulated with the proper combination of an `if-then` statement and a `goto` statement. This is referred to as the *implementation variation* problem by Wills [60].

```
150 IF(C2.GT.1.0D0) THEN
    C1 = 1.0D0 - E**2
    C2 = C1/C2GAM
    GO TO 150
    END IF
```

Figure 7 Simulated while control flow

Figure 7 shows FORTRAN `if-then` and `goto` statements that simulate a `while` loop. By recognizing these simulated control flow structures in the legacy code, they can be represented in the canonical form as the control flow entity they emulate. For example, the `if-then` and `goto` structure from Figure 7 are transformed into an equivalent `while` structure in the canonical form.

2.2.2.4 Generic Imperative Model. The Generic Imperative Model (GIM) has been developed as the canonical form for the re-engineering methodology, as shown in Figure 5. The GIM includes fundamental aspects of imperative programming languages such as FORTRAN, C, Pascal, Cobol, and Ada 83. Chapter III presents the GIM in detail.

2.2.3 Program Slices. Once the legacy code is converted to the canonical form, a process of *program slicing* is used to split the imperative legacy code into *program slices* based on functionality of the routines. A *program slice* [76] is the collection of imperative programming statements extracted from a legacy program that are required to produce a single value. Program slicing over entire legacy systems has been used in such applications as software maintenance and debugging [22] to isolate sections of code. The approach taken in this research is to use program slicing on a single legacy subprogram, as discussed in more detail in Chapter V.

The rationale for using program slicing in the re-engineering methodology is to split the legacy code into separate imperative subprograms that resemble object-oriented methods. In his Object Modeling Technique, Rumbaugh [62] defines a specific type of object-oriented method called *queries.* A query of an object returns a specific value from the object. By using program slicing, an imperative subprogram that returns multiple values can be split into multiple subprograms that each return a single value. These new subprograms now resemble query methods and can be built into an object class.

2.2.4 Parameter-Based Object Identification (PBOI). This section describes the arrow shown in Figure 5 labeled **PBOI**. The goal of this step is to organize the data and associated program slices into objects and classes that comprise the extracted object-oriented design. Chapter V describes the Parameter-Based Object Identification (PBOI) methodology developed by this research that extracts objects based on the data items being passed throughout the legacy system. As each object is extracted, a class is built in the object-oriented design that defines the attributes and operations of the object. As a final step in this methodology, *duplicate* object instances are eliminated and *overlapping* classes are merged. This is explained in detail in Chapter V. In order to scope this research, generalization and specialization hierarchies are not identified for the extracted classes. Each class is built as a sub-class of an overall super-class resulting in a flat hierarchy of classes.

2.2.5 Generic Object-Oriented Design Model. The Generic Object-Oriented Design Model (GOM) is a canonical form that models any object-oriented design. The GOM has been developed as part of this research and is used to represent the extracted design. The design is built by creating entities from the GOM such as objects, classes, messages, and methods. The GOM also defines a design entity that is a collection of the extracted classes. Chapter IV defines the GOM in detail.

2.2.6 Generic Object-Oriented Program Model. The Generic Object-Oriented Program Model (GOM-P) is a canonical form that models any object-oriented program. The program differs from the design in that the object-oriented methods include object-oriented programming statements. This research assumes the statements of each method are from the *imperative* programming language paradigm. Part of the re-engineering process is to transform GIM statements to GOM-P statements. The GOM-P models languages such as C++, Java, and Ada 95. Chapter IV defines the statements that are modeled in the GOM-P.

2.2.7 Requirements Specification. Figure 5 shows a dashed arrow from the program slices to the requirements specifications. Although this specific process is outside the scope of this research, it will be addressed briefly. It may be possible to extract specifications from the partitions that describe the functionality of the imperative code. These requirements specifications might be used to create an equivalent Object-Oriented Analysis (OOA). Being able to recover accurate specifications of the legacy code would clearly allow a more extensive and robust analysis of the legacy system. However, recovering specifications from code is considered a hard problem [44] and is left as future research.

2.2.8 Generic Object-Oriented Analysis Model. The Generic Object-Oriented Analysis Model (GOM-A) is a canonical form that models any object-oriented analysis. According to Rumbaugh [62], Object-Oriented Analysis (OOA) differs from Object-Oriented Design (OOD) in that the OOA object model includes ***associations*** between objects and that the functional and dynamic models are separate in OOA. In OOD, the dynamic and functional models are incorporated into the object model. For these reasons, the GOM-A includes entities for the object, dynamic, and functional models. It also includes entities for the associations between objects. It is hypothetically possible to build a GOM-A covering any object-oriented analysis. The GOM-A is not included as part of this research. DeLoach [15] has done work in this area and developed a generic model of the Rumbaugh Object Modeling Technique (OMT) [62].

It is possible to convert a requirements specification recovered from the imperative legacy code to an object-oriented analysis as indicated in Figure 5, but this research does not include such processing. The arrow in Figure 5 from the GOM to the GOM-A shows a hypothetical abstraction process. This is left as future research.

2.2.9 Object-Oriented Program. Once the OOD has been built by the reengineering methodology, it is possible to convert it to an object-oriented program. This program can be implemented using any object-oriented language such as C++ [71] or Ada 95 [3]. This research has done only rudimentary prototyping of this step, as explained in Chapter VIII.

2.2.10 Object-Oriented Products and Processes. The rounded-rectangles in Figure 5 labeled OOA, OOD, and OOP represent the products of Object-Oriented Analysis, Object-Oriented Design, and Object-Oriented Programming, respectively. There are currently several methodologies for analysis, design, and programming in the object paradigm [9,10,62]. These rounded-rectangles are included in the figure to imply a correlation between the object-oriented design produced by the re-engineering methodology and an object-oriented design produced through forward object-oriented analysis and design.

2.2.11 Maintaining Functional Equivalence. After the object extraction methodology has been used on an imperative legacy system, the question remains about the utility of the transformation. How can one judge that the object-oriented design produced from the methodology is of any value? This question can be interpreted in at least two different ways. First, the question could be asking about the quality of the design and whether the recovered design is a *good* object-oriented design. Second, the question could be asking about the value of the design in general and whether representing the legacy system in the object-oriented paradigm adds value.

The former view requires a metric to determine the quality of an object-oriented design. This is an open research issue in the area of object-oriented analysis and design, i.e. there is currently no metric that accurately judges the quality of an object-oriented design. There is only an informal attempt in this research to answer the question from this view.

Instead, the value of the object-oriented design recovered is judged from the latter perspective. If it can be shown that the object-oriented design recovered from the imperative legacy code returns the same output as the legacy system given the same input, then the design is quite valuable. Such an object-oriented design can replace an aging legacy system and provide the same functionality as the original system while allowing the maintenance of the system to be done in the object-oriented paradigm. Chapter VII provides a proof that the design extracted using this methodology is functionally equivalent to the legacy code.

2.3 Summary

This chapter has presented a summary of the re-engineering methodology defined in this research. The methodology transforms imperative legacy code into the GIM, uses program slicing on the subprograms, extracts objects from the data items in the legacy system, and represents the extracted design using the GOM and GOM-P. The rationale for each of these steps has been discussed briefly and more detailed explanations are provided in the following chapters. Several other entities, such as the requirements specifications and the GOM-A, were presented and left for future research.

III. The Generic Imperative Model

This chapter defines the Generic Imperative Model (GIM) developed to model the variables, expressions, assignment statements, and control flow typically built into imperative programming languages. The imperative paradigm is discussed first, followed by detailed descriptions of how these traits of imperative programming languages are modeled using Abstract Syntax Trees (ASTs). Rumbaugh's notation [62] for classes and objects is used to present the objects and classes that define these ASTs. Formal semantics are defined for each GIM entity using the weakest precondition notation [18,19,24,25].

3.1 The Imperative Paradigm

According to Dershem [16], there are currently four different programming language paradigms: imperative, logic, functional, and object-oriented. Tennent [72] and Ghezzi and Jazayeri [23] describe the imperative programming language paradigm as a style of programming based on the following concepts.

Variables Variables hold state information during execution of the program.

Data Types Data types define the acceptable values for a variable and the operations that can be done on the variable.

Expressions Expressions are combinations of variables and operators used to express temporary intermediate values.

Assignment Statements Assignment statements change state by assigning new values to variables via expression evaluation.

Input/Output Input and output statements read and write to the standard input/output devices and to files.

Sequential Control In sequential control flow, a sequence of statements executes one after another.

Selective Control In selective control flow, a choice is made, based on the result of a boolean expression, between executing one sequence of statements versus another.

Iterative Control In iterative control flow, a sequence of statements is executed repeatedly while a boolean expression is true.

Procedural Abstraction A procedural abstraction collects a sequence of statements that are executed when the abstraction is referenced by name. A procedural abstraction can be passed parameters and may return values.

Main Program In *systems* of imperative subprograms, there is always one subprogram that is given the flow of control as the system begins execution. This special subprogram is termed the *main program.*

Imperative programming languages include FORTRAN, C, Pascal, Ada, Cobol, and others. In fact, any language where the majority of the language implements the concepts presented above is considered an imperative programming language. These are the concepts that distinguish the imperative paradigm from the other programming language paradigms.

These imperative programming language constructs are modeled in the GIM by building ASTs that store knowledge about the constructs. For a specific programming language such as Ada 83, it is possible that a construct in the language is not part of the imperative paradigm. For example, the `accept` and `entry` statements implement communication between tasks in Ada 83. These statements are not modeled in the GIM because they fall outside the definition of the imperative paradigm presented in this chapter. Overall, this means certain imperative languages can not be completely modeled by the GIM. The Feasibility Demonstration Chapter (Chapter VIII) discusses a method for determining which parts of a specific language can be modeled by the GIM.

The organization of this chapter emphasizes the importance of assignment and control flow in the imperative paradigm as opposed to the issues of data storage and expressions. Unfortunately, one cannot talk about assignment without variables and expressions. The reader is referred to Section 3.14 and Section 3.20 for the explanations of how variables and expressions, respectively, are modeled in the GIM.

For each programming language construct modeled in the GIM, formal semantics are provided using the *state model* [19] of programs. Preconditions and postconditions are used to define the semantics for each GIM representation of an imperative construct. Specifically, given a *postcondition* R that is guaranteed to be true after a statement S is executed, the *weakest precondition*, $wp(S,\ R)$, defines the weakest set of preconditions that must hold in order for the execution of S to establish R [19]. This research relies heavily on the previous work done by Djikstra [18], Gries [25], and Dromey [19] to define formal semantics for imperative programming language constructs.

3.2 The GIM Domain Model

This section presents a brief overview of the ASTs that are included in the GIM domain model. Figure 8 shows a partial representation of the GIM domain model. The

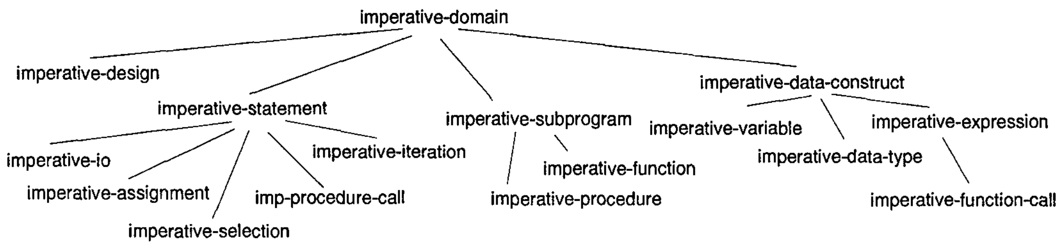


Figure 8 GIM Domain Model

overall superclass of the domain is the `imperative-domain` AST. The `imperative-design` class models collections of imperative subprograms as defined in Section 3.8. The abstract class `imperative-statement` is the superclass for all imperative programming statements modeled in the GIM. The `imperative-data-construct` class is the superclass of imperative expressions, data types, and variables modeled in the GIM. Each of the lower-level classes in the domain model are described in the rest of this chapter.

3.3 Imperative Assignment

An imperative assignment statement takes the general form $x := e$, where x is a variable and e is an expression of the same type. When this statement is executed, the expression e is evaluated and the result is assigned to x [19]. Assignment statements in the GIM are modeled using the `imperative-assignment` class. Figure 9 shows the

imperative-assignment
imp-assign-lhs : imperative-variable imp-assign-rhs : imperative-data-construct

Figure 9 Imperative Assignment Class

class description for the AST that models `imperative-assignment` in the GIM (using Rumbaugh's notation [62]). The `imp-assign-lhs` attribute models the variable being assigned a value. The `imp-assign-rhs` attribute models the expression to be evaluated and assigned to the variable. When variables appear on the right hand side of an assignment, they are considered expressions. The GIM is able to model both variables and expressions on the right hand side of the assignment statement.

As an example, consider the following C assignment statement.

```
mfpgrc = 0;
```

This assignment statement is modeled in the GIM by creating an instance of the class `imperative-assignment`, which is shown in Figure 9. Figure 10 shows the object instance

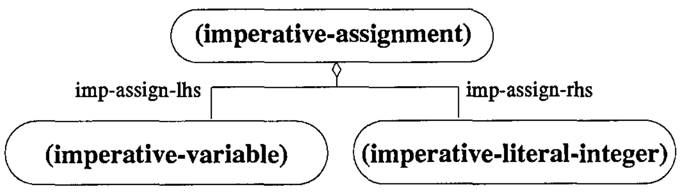


Figure 10 Imperative Assignment Object

that models this C assignment statement in the GIM. The variable `mfpgrc` is modeled as a variable in the GIM (see Section 3.14) and is stored in the `imp-assign-lhs` attribute. The value 0 is modeled as a literal expression (see Section 3.20) and is stored in the `imp-assign-rhs` attribute of the AST.

The semantics for the GIM assignment statement are defined formally using the *weakest precondition* notation. Let R_e^x denote the postcondition R, with all free occurrences of x simultaneously replaced by e [19]. The semantics for the general form of imperative assignment, $x := e$, are defined using the weakest precondition notation as shown below.

Definition III.1. $wp(x := e,\ R) = R_e^x$

In the GIM, the `imp-assign-lhs` models x and the `imp-assign-rhs` models e. By relating the general form of assignment to the specific representation of assignment in the GIM, the formal semantics for assignment in the GIM are now defined.

3.4 Imperative Sequential Control

The default method of program control in the imperative paradigm is sequential control flow where a sequence of statements is executed one statement after another. Program statements that are executed sequentially in an imperative programming language are modeled in the GIM by storing the statements in a sequence. The order of the statements in the sequence corresponds to the order of the statements from the imperative program.

For example, in the collection of statements shown below, `<Statement 1>` is executed followed by `<Statement 2>` followed by `<Statement 3>`.

```
<Statement 1>
<Statement 2>
<Statement 3>
```

This collection of statements is modeled in the GIM using the following sequence.

```
[<Statement 1>, <Statement 2>, <Statement 3>]
```

The semantics for sequential control in the GIM are based on the semantics of the *composition* command [25] and are defined using weakest precondition notation. Let $S1$ and $S2$ be imperative statements and $[S1, S2]$ represent the sequential composition of the two statements.

Definition III.2. $wp([S1, S2],\ R) = wp(S1,\ wp(S2, R))$

Definition III.2 leads to the claim that the GIM representation of sequential control flow is *associative*. Let $S1$, $S2$, and $S3$ be imperative statements.

Theorem III.1. $wp([S1, [S2, S3]],\ R) = wp([[S1, S2], S3],\ R)$

Proof.

$$\begin{aligned} wp([S1,[S2,S3]],\ R) &= wp(S1, wp([S2,S3],\ R)) \\ &= wp(S1, wp(S2, wp(S3,\ R))) \\ &= wp([S1,S2], wp(S3,\ R)) \text{ (by function composition)} \\ &= wp([[S1,S2],S3],\ R) \end{aligned}$$

□

Since the weakest precondition for the sequences $[S1, [S2, S3]]$ and $[[S1, S2], S3]$ are equal, the sequence $[S1, S2, S3]$ is often used in this document to represent these sequences. In addition, singleton sequences and empty sequences are used when convenient.

3.5 Imperative Selective Control

In the imperative paradigm, control flow consisting of a selection between two or more statements is called *selective* control flow. The selection between statements $S1$ and $S2$ is based on a boolean expression B. In general, selective control flow takes the following form.

```
if $B$ then
  $S1$
else
  $S2$
```

To be more specific, $S1$ and $S2$ above may consist of sequentially composed statements, so they are modeled in the GIM as sequences of statements. $S1$ must include at least one statement to execute, but $S2$ can be an empty sequence. Selective control flow is typified in imperative programming languages by the `if-then-else` statement. Selective control flow where $S2$ is empty is typified in the imperative paradigm by the `if-then` statement.

Selective control flow is modeled in the GIM by using the `imperative-selection` class. Figure 11 shows the class description for the `imperative-selection` class. The

imperative-selection
imperative-exp : imperative-data-construct imperative-then-part : seq(imperative-program-construct) imperative-else-part : seq(imperative-program-construct)

Figure 11 Imperative Selection Class

`imperative-exp` attribute models the boolean expression that controls the selection. The `imperative-then-part` models the sequence of imperative statements that are executed if the boolean expression is true. The `imperative-else-part` attribute models the sequence of imperative statements that are executed when the boolean expression is false.

For example, the following FORTRAN `if-then-else` statement shows a choice between executing the statement `LEXIST(I) = 1` or the statement `LERROR = .TRUE.`.

```
IF (MFPGRC .EQ. 0) THEN
  LEXIST(I) = 1
ELSE
  LERROR = .TRUE.
END IF
```

The choice is made based on the value of the boolean expression (MFPGRC .EQ. 0). This FORTRAN `if-then-else` statement is represented in the GIM as shown in Figure 12.

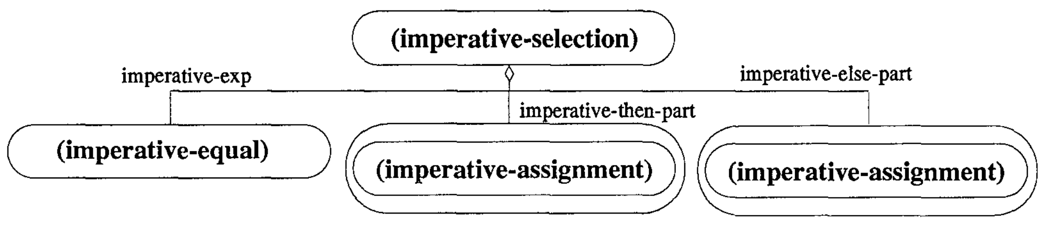


Figure 12 Imperative-Selection Object

The `imperative-exp` attribute holds the `<imperative-equal>` object which models the expression (MFPGRC .EQ. 0). The `imperative-then-part` attribute holds the singleton sequence containing the assignment statement from the `then` part. The `imperative-else-part` holds the singleton sequence containing the assignment statement from the `else` part.

The semantics for the `imperative-selection` statement are defined using weakest precondition notation. Let B represent an imperative boolean expression and let $S1$ and $S2$ represent sequences of imperative statements. The general form of selective control flow presented at the beginning of this section as represented in the GIM is given by the following form.

$$\textbf{if } B \textbf{ then } S1 \textbf{ else } S2$$

The sequence $S1$ is executed if B is true and the sequence $S2$ is executed if B is false. The semantics of this GIM form of selective control flow are defined below.

Definition III.3. $wp(\textbf{if } B \textbf{ then } S1 \textbf{ else } S2, R) = (B \Rightarrow wp(S1, R)) \wedge (\neg B \Rightarrow wp(S2, R))$

To relate these formal semantics to the GIM, note that B is modeled using the `imperative-exp` attribute. The sequence of statements $S1$ is modeled by the `imperative-then-part` attribute and the sequence $S2$ is modeled by the `imperative-else-part` attribute. The formal semantics for selective control flow as modeled in the GIM are now defined.

The formal semantics for selective control flow in the GIM when the sequence $S2$ is empty deserve special attention. If $S2$ is empty, then no change in state occurs if the boolean expression B evaluates to false. Let **skip** represent a statement that has no effect on the state of a program.

Definition III.4. $wp(\textbf{skip},\ R) = R$

Selective control flow in the GIM when $S2$ is empty takes the following form.

$$\textbf{if } B \textbf{ then } S1 \textbf{ else skip}$$

The formal semantics for this form of selective control in the GIM is defined below.

Definition III.5. $wp(\textbf{if } B \textbf{ then } S1 \textbf{ else skip},\ R) \ = \ (B \Rightarrow wp(S1,\ R)) \ \wedge \ (\neg B \Rightarrow R)$

3.6 Imperative Iterative Control

The imperative paradigm also includes a control mechanism for repeating a sequence of statements known as *iterative* control flow. With iterative control flow, the sequence of statements is repeated while some boolean expression is true. Let B represent the boolean expression that controls the iteration and $S1$ represent the sequence of imperative statements that are repeated. The general form of iteration in the imperative paradigm is defined as

$$\begin{array}{c}\text{while } B \\ S1\end{array}$$

Execution of the sequence $S1$ continues while the boolean B is true.

Iteration in the GIM is modeled using ASTs built from the `imperative-iteration` class. Figure 13 shows the class description for the `imperative-iteration` class. The

imperative-iteration
iter-exp : imperative-expression iter-body : seq(imperative-program-construct)

Figure 13 Imperative-Iteration Class

`iter-exp` attribute models the boolean expression B and the `iter-body` attribute models the sequence of statements $S1$.

For example, the following `while` statement from Pascal provides an example of imperative iteration.

```
WHILE Destination > Distance2 DO
  BEGIN
    Time := Time + Interval;
    Distance2 := Distance2 + Increment
  END
```

This `WHILE` statement is modeled using the `imperative-iteration` object shown in Figure 14. Other imperative programming languages include statements that implement

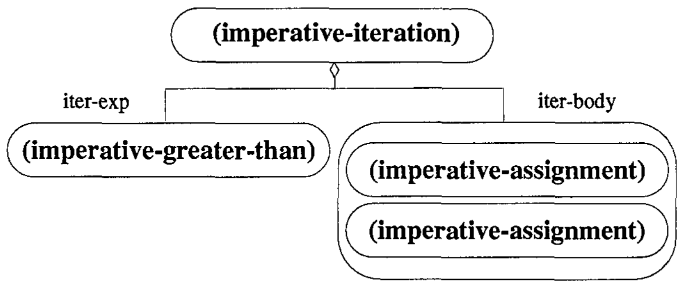


Figure 14 Imperative-Iteration Object

imperative iteration, for example, the `while` statement in C and the `loop` statement in Ada. These statements and statements in other programming languages provide several options for implementing imperative iteration. The GIM `imperative-iteration` class provides one canonical form to represent these looping mechanisms. Chapter VIII demonstrates the conversion of the FORTRAN `DO` loops, `FOR` loops, and loops formed through the structured use of the `GOTO` statement.

The formal semantics of iteration in the GIM are defined using a weakest precondition notation based on the guarded loop mechanism. Given a precondition P and a postcondition R, let $H_k(R)$ represent the set of all states in which

$$\text{while } B \text{ do } S1$$

will establish R in at most k iterations [19].

The formal semantics for iterative control flow in the GIM are defined below.

Definition III.6. $wp(\textit{ while } B \textit{ do } S1,\ R) = (\exists k : 0 \leq k\ \wedge\ H_k(R))$

To tie this general definition to the specific model of iterative control flow in the GIM, recall that B is modeled by the `iter-exp` attribute and $S1$ is modeled by the `iter-body` attribute of the `imperative-iteration` class.

Admittedly, the task still remains to define $H_k(R)$. There is no attempt in my research to define *loop invariants* [19] for iteration in the GIM. What is provided is a general definition of the formal semantics for iterative control flow.

3.7 Imperative Subprograms

The imperative paradigm allows a programmer to group a sequence of statements together into a suitable abstraction unit that may be referenced by name. This mechanism is known as the *subprogram* [23], which gives a name to a sequence of statements. A subprogram call invokes the abstraction unit, i.e. forces control to transfer to the called unit, which upon completion returns control to the calling point [23].

Parameter passing conventions allow explicit communication between a subprogram and a call to the subprogram [23]. A *formal* parameter appears in the definition of a subprogram, and an *actual* parameter appears in the call of a subprogram. Note that the names of the identifiers used as actual parameters need *not* match the names of the identifiers of the formal parameters of a subprogram. In fact, an actual parameter need not be an identifier at all , but can be an expression as well[1].

The following definitions are needed to define subprograms more fully.

Definition III.7. *A data item has a* definition *[2] in an imperative subprogram when the subprogram includes an assignment statement or an input statement that assigns a value to the data item.*

Definition III.8. *A subprogram parameter is an* output *parameter if the subprogram includes a* definition *of the parameter. A parameter is also an* output *parameter if the subprogram invokes another subprogram and the parameter is an* output *parameter of the called subprogram.*

Definition III.9. *A data item has a* use *[2] in an imperative subprogram when any of part of a statement in the subprogram references the data item and obtains its value.*

Definition III.10. *A subprogram parameter is an* input *parameter if a* use *of the parameter occurs before a* definition *of the parameter.*

Specific exemplars of the subprogram in the GIM include *procedures* and *functions.* A procedure in the GIM is a subprogram that does not explicitly return a value at the end of its processing. Instead, all values returned from procedures are returned via output parameters. For this research, the following restriction applies to procedures in the GIM.

Restriction III.1. *A formal parameter of a procedure must not be both an* input *and an* output *parameter.*

Appendix D provides a process for converting procedures with a parameter that is both an input and output parameter into a procedure that has no such parameters. A *function* in the GIM is a subprogram abstraction that *does* return a value at the end of its processing. For this research, the following restriction applies to functions in the GIM.

Restriction III.2. *All functions in the GIM return a* single *value at the end of their execution and have no output parameters.*

[1]This aspect of parameter passing will be restricted by the re-engineering methodology, as described in Section 3.11.

This research provides no process for converting a function with output parameters into a function with no output parameters.

3.8 Imperative Designs

In order to model the collection of subprograms as a whole, the subprograms are stored in an AST built from the GIM class `imperative-design`. Figure 15 shows the class

imperative-design
imperative-programs : seq(imperative-subprogram) imperative-files : seq(imperative-file)

Figure 15 GIM Imperative Design Class

description of the `imperative-design` class. The `imperative-programs` attribute holds the collection of subprograms that comprise the design. The `imperative-files` attribute holds the collection of input/output files being modeled for this design. See Section 3.21 and Section 3.22 for discussions of how input and output statements and files are modeled in the GIM.

3.9 Imperative Procedures

Many imperative languages include a construct for declaring procedures. In FORTRAN, the `SUBROUTINE` statement is used to define the name, formal parameters, and statements of a procedure.

For example, the FORTRAN code shown in Figure 16 defines the procedure `VADD`. The sequence of statements included in the body of this procedure will be executed anytime the `VADD` procedure is called. A call to this procedure must include actual parameters that match the types of `IOP`, `R1`, `R2`, and `R3`. Other examples of programming language constructs that implement procedures are the `procedure` constructs in Pascal and Ada.

The declaration of a procedure is modeled in the GIM using ASTs built from the `imperative-procedure` class. Figure 17 shows the class description for the `imperative-procedure` class. The `imp-proc-identifier` attribute is used to model the name of the procedure. The `imp-proc-formals` attribute models the sequence of formal parameters

```
      SUBROUTINE VADD(IOP,R1,R2,R3)
       INCLUDE 'bdincl.f'
C
C      THIS ROUTINE PERFORMS VECTOR ADDITION FOR THREE DIMENSIONAL
C       VECTORS
C
      DIMENSION R1(3),R2(3),R3(3)
      F = FLOAT(IOP)
      R3(1) = R1(1) + F*R2(1)
      R3(2) = R1(2) + F*R2(2)
      R3(3) = R1(3) + F*R2(3)
      RETURN
      END
```

Figure 16 FORTRAN Subroutine VADD

imperative-procedure
imp-proc-identifier : symbol imp-proc-formals : seq(imperative-variable) imp-proc-statements : seq(imperative-program-construct)

Figure 17 GIM Procedure Class

defined for the procedure. The **`imp-proc-statements`** attribute holds the sequence of statements from the procedure. Figure 18 shows the AST object that models the FORTRAN procedure `VADD` shown in Figure 16.

The declaration of a subprogram does not change the state of a program. It is the *invocation* of the subprogram that may affect the state. For this reason, the formal semantics for subprogram declarations defined in this section serve only as a foundation for the definition of the formal semantics of subprogram *invocations*.

Let $S1$ represent the sequence of statements declared in a subprogram. Let P represent the precondition of $S1$ and let R represent the postcondition of $S1$ such that the following holds.

$$\{P\}$$
$$S1$$
$$\{R\}$$

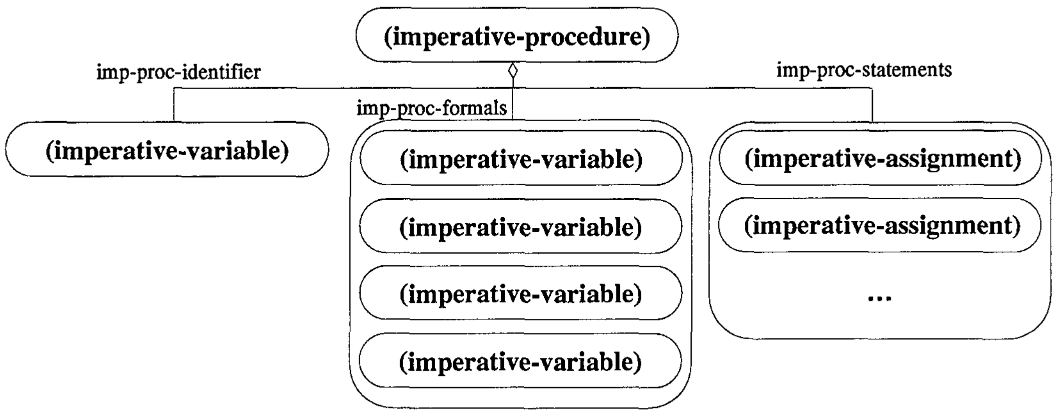


Figure 18 Modeling a Procedure

For a subprogram p, let $\bar{x}$ be a vector representing the input parameters of p. Let $\bar{z}$ be a vector representing the output parameters of p. Each procedure declaration in the GIM takes the following form.

$$\begin{array}{l} \text{procedure } p(\bar{x}, \bar{z}) \\ \quad \{P\} \\ \quad S1 \\ \quad \{R\} \end{array}$$

3.10 Imperative Functions

Many imperative languages also provide a mechanism for declaring functions. For example, the FORTRAN `FUNCTION` statement declares a function that returns a specific type of value from the execution of the statements in the function.

As an example, consider the FORTRAN function declaration found in Figure 19. This function returns a value of type `DOUBLE PRECISION` after the statements in the body of the function are executed. A call to this function must include actual parameters that match the data type of `BETA`, `XLAMDA`, and `DIAM`.

Functions in the imperative paradigm are modeled in the GIM using the **`imperative-function`** class. Figure 20 shows the class description for the **`imperative-function`** class. The **`imp-func-identifier`** attribute holds the name of the function being modeled. The **`imp-func-formals`** attribute models the sequence of formal parameters defined for the

```
      DOUBLE PRECISION function PRDIV(BETA,XLAMDA,DIAM)

      INCLUDE 'bdincl.f'
C
C      CALCULATES THE PROJECTOR BEAM DIVERGENCE FOR A GIVEN OPTICS
C        SIZE THIS IS BASED ON THE RAYLEIGH (AIRY DISK) CRITERION.
C
      DATA FAC/1.220D0/

      QUAL = MAX(BETA,1.0D0)
      PDIAM = MAX(DIAM,0.1D0)
      WAVELN = XLAMDA*1.D-06
      PRDIV = QUAL*WAVELN*FAC/PDIAM
      RETURN
      END
```

Figure 19 FORTRAN Function PRDIV

imperative-function
imp-func-identifier : symbol imp-func-formals : seq(imperative-variable) imp-func-statements : seq(imperative-program-construct) imp-function-return-type : imperative-data-type

Figure 20 GIM Function Class

function. The `imp-func-statements` attribute holds the sequence of statements from the function. The `imp-function-return-type` attribute models the data type of the value returned from the function. The FORTRAN function `PRDIV` shown in Figure 19 is modeled in the GIM using the `imperative-function` object shown in Figure 21.

Because of Restriction III.2, the formal semantics of a function declaration are defined more restrictively than the formal semantics of a procedure declaration. For a function f, let $\bar{x}$ be a vector representing the input parameters in f. Each function declaration in the GIM takes the following form.

$$\begin{array}{l} \text{function } f(\bar{x}) \\ \quad \{P\} \\ \quad S1 \\ \quad \{R\} \end{array}$$

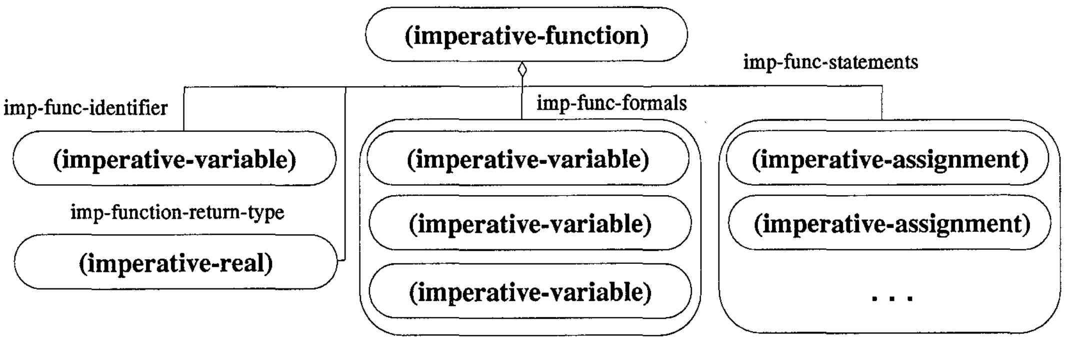


Figure 21 Modeling a Function

In order to define the semantics for the value that is returned from the function f, a modified representation of imperative functions is used. Let y represent the value that is returned from f. Let f' be a procedure that takes the same input parameters as f, and includes y as the single output parameter. The statements in f' are the statements from f, viz. $S1$. The procedure f' takes the following form.

```
procedure $f'(\bar{x}, y)$
    $\{P\}$
    $S1$
    $\{R\}$
```

This modified representation of imperative functions is used to define the semantics of a function invocation, as described in Section 3.12.

3.11 Imperative Procedure Invocation

If an imperative language provides a way to define a subprogram, it will also provide a way to invoke the subprogram. This ***subprogram call*** mechanism refers to the called subprogram by name and passes any actual parameters required by the subprogram. This section defines one of the specific implementations of the subprogram call, viz. procedure calls.

An imperative procedure is invoked using an imperative procedure call, which is a kind of statement in the GIM. The call to a procedure is modeled in the GIM using ASTs built from the `imp-procedure-call` class. Figure 22 shows the class description for the

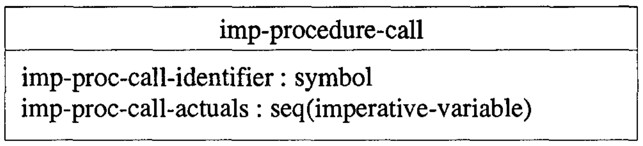

imp-procedure-call
imp-proc-call-identifier : symbol
imp-proc-call-actuals : seq(imperative-variable)

Figure 22 Imperative Procedure Call Class

`imp-procedure-call` class. The `imp-call-identifier` attribute holds the name of the procedure being invoked. The `imp-call-actuals` attribute holds the actual parameters used in the procedure call.

For example, in FORTRAN, the invocation of a defined `SUBROUTINE` is implemented using the `CALL` statement. The FORTRAN procedure `VADD` shown in Figure 16 is invoked with the following `CALL` statement.

```
CALL VADD(1, R1, R2, R3)
```

This procedure call invokes the procedure `VADD` and passes in the parameters `1`, `R1`, `R2`, and `R3`.

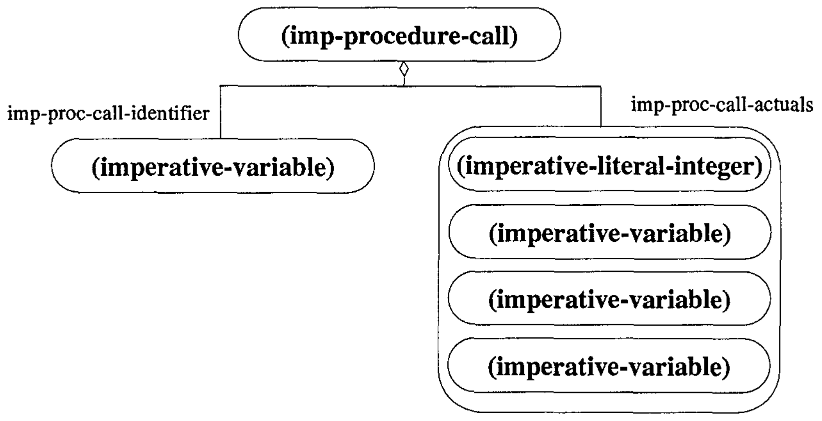


Figure 23 Imperative Procedure Invocation Object

Figure 23 shows how this invocation of `VADD` is modeled in the GIM using an instance of the `imp-procedure-call` class. Because of the nature of the object extraction methodology, actual parameters used in subprogram calls must be variables. This leads to the following restriction on subprogram calls.

Restriction III.3. *All actual parameters in subprograms calls must be variables.*

For this reason, actual parameters such as the integer `1` in Figure 23 are replaced by a temporary variable and an assignment statement is added to assign this variable the value of the parameter. For example, the original GIM call to `VADD` is shown below.

```
VADD ( 1, R1, R2, R3 )
```

The updated call to `VADD` with the integer actual parameter replaced by a variable is shown below.

```
TEMP-29 := 1;
VADD ( TEMP-29, R1, R2, R3);
```

The formal semantics of a procedure call are defined as follows. Let p be a procedure and let $S1$ be the sequence of statements declared in p. Let $\bar{a}$ be a vector representing the actual parameters that correspond to input parameters in p. Let $\bar{c}$ be a vector representing the actual parameters that correspond to output parameters in p. An invocation of the procedure p takes the following form.[2]

$$p(\bar{a}, \bar{c})$$

To invoke p, the actual parameters are copied to the formal parameters and control flow transfers to p. Executing the procedure call is equivalent to executing the following sequence.

$$[\bar{x} := \bar{a}, S1, \bar{c} := \bar{z}]$$

Using weakest precondition notation, the semantics for a procedure call are defined below.

Definition III.11. $wp(p(\bar{a}, \bar{c}), R) \;=\; wp([\bar{x} := \bar{a}, S1, \bar{c} := \bar{z}],\; R)$

3.12 Imperative Function Invocation

This section defines the other specific implementation of the subprogram call, viz. the function call. Most imperative languages invoke declared functions by using the name of the function and passing in the required parameters. A function call is a kind of expression.

[2]Here, p represents both the procedure entity and the identifier that names the procedure.

Because of Restriction III.3, expressions do not appear in procedure calls but do appear in assignment, selective, and iterative statements. This implies function calls do not appear in procedure calls in the GIM, but can appear in the other statements. The call to a function is modeled in the GIM using the `imp-function-call` class. Figure 24 shows

imp-function-call
imp-fun-call-identifier : symbol imp-fun-call-actuals : seq(imperative-variable)

Figure 24 Imperative Function Call Class

the class description for the `imp-function-call` class. The `imp-fun-call-identifier` attribute holds the name of the procedure being invoked. The `imp-fun-call-actuals` attribute holds the actual parameters used in the function call.

An important difference between the invocation of procedures and functions in the imperative paradigm is that procedure calls can only appear as whole statements and not part of another statement. Function calls can only appear as part of an expression that is part of another statement and not as whole statements themselves.

For example, the FORTRAN function `PRDIV` shown in Figure 19 is invoked as part of the following statement.

```
SIGPR = PRDIV(BETA, XLAMDA, DIAM)
```

This function call invokes the function `PRDIV` passing in the parameters `BETA`, `XLAMDA`, and `DIAM`. The result of this function call is used as the expression assigned to the variable `SIGPR`. Figure 25 shows how the invocation of `PRDIV` is modeled in the GIM using an

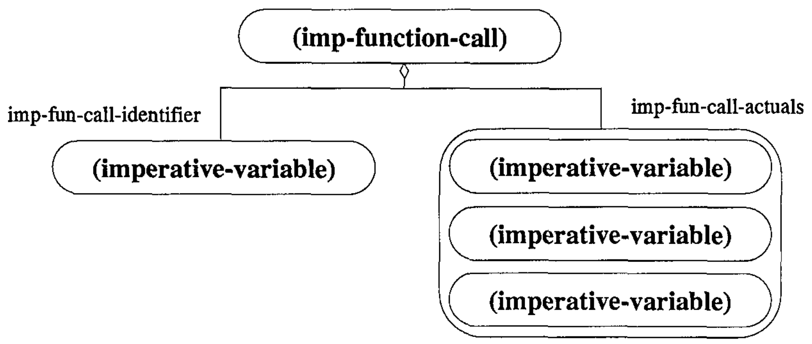


Figure 25 Imperative Function Invocation Object

instance of the `imp-function-call` class.

The formal semantics for function calls in the GIM are defined as follows. Let f be a function and let $S1$ be the sequence of statements declared in f. Let $\bar{a}$ be a vector representing the actual parameters that correspond to the input parameters in f. Since every function appears as part of an expression in some statement, let S represent the statement in which f appears. Recall the modified representation of a function presented in Section 3.10 includes one output parameter, y, which represents the value returned from the function f. Let f' be the procedure that represents f with the input parameters from f and the additional parameter y. Let b represent the actual parameter that corresponds to y.

Using these definitions, an invocation of the function f takes the following form.

$$f(\bar{a})$$

An invocation of the procedure, f', takes the following form.

$$f'(\bar{a}, b)$$

The difference between the invocations of f and f' is that the invocation of f is part of S and the invocation of f' is a single statement. Invoking f is equivalent to invoking f' and then substituting b in S for any invocations of f. This provides a formal representation of the value returned from f. The substitution of b for the call to f in S is represented by the following notation.

$$S_b^{f(\bar{a})}$$

In this way, the call to f is equivalent to the following sequence of statements.

$$[f'(\bar{a}, b), S_b^{f(\bar{a})}]$$

As defined in Section 3.11, when the procedure f' is invoked, the actual parameters in $\bar{a}$ are copied to the formal parameters in $\bar{x}$ and control flow transfers to f'. After execution of the function, the value of y is copied to b. Hence, executing the call to f' is equivalent to executing the following sequence.

$$[\bar{x} := \bar{a}, S1, b := y]$$

Using weakest precondition notation, the formal semantics of a call to function f, where the statement S contains $f(\bar{a})$, are defined below.

Definition III.12. $wp(S,\ R)\ =\ wp([\bar{x} := \bar{a}, S1, b := y, S_b^{f(\bar{a})}],\ R)$

3.13 Recursion

In the imperative paradigm, it is possible for a subprogram to invoke itself in a process known as *recursion.* Knuth [37] claims that any recursive algorithm can be transformed into an iterative algorithm. For this reason, the GIM does not model recursion. This leads to the following restriction.

Restriction III.4. *Subprograms to be modeled in the GIM are not allowed to make calls to themselves.*

Even more restrictively, the *call tree* consisting of the graph of subprogram calls made in a collection of imperative subprograms is assumed to be a *directed acyclic graph* or DAG.

Restriction III.5. *The* call tree *of a collection of imperative subprograms must be a* directed acyclic graph.

3.14 Imperative Variables

Each variable that is set or used in an imperative program is modeled in the GIM AST by building an `imperative-variable` object. Figure 26 shows the class description for the `imperative-variable` object. The `imp-data-scope` attribute models the name of the procedural abstraction in which this variable is defined. The `imp-data-identifier` attribute models the name of the imperative variable. The `imp-data-type` attribute is an

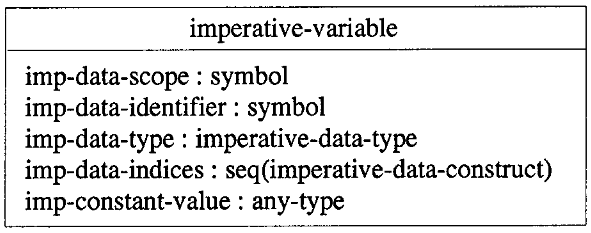

imperative-variable
imp-data-scope : symbol
imp-data-identifier : symbol
imp-data-type : imperative-data-type
imp-data-indices : seq(imperative-data-construct)
imp-constant-value : any-type

Figure 26 Imperative-Variable Class

object that describes the data type of the imperative variable. The **imp-data-indices** attribute stores the GIM representations of the indices for this variable, if any. The **imp-constant-value** attribute stores the value assigned to the variable if the variable is a constant.

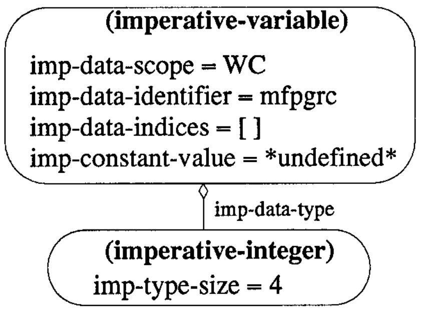


Figure 27 Imperative-Variable Object

For example, Figure 27 shows how the C integer variable **mfpgrc**, defined in a procedure **wc**, is represented in the GIM. Here, the variable **mfpgrc** is modeled by building an instance of the **imperative-variable** class representing an integer variable with no indices and no constant value. The **imp-data-scope** attribute is set to the symbol **WC** since this is the name of the procedure in which this variable is defined. Scoping issues are discussed in more detail in Section 3.16. The data type of **mfpgrc** is modeled using the **imperative-integer** object in the GIM. Certain data types are modeled in the GIM as discussed in Section 3.15.

3.15 Imperative Data Types

In order to model variables in the GIM, certain data types are also modeled. The data types being modeled are:

Integer zero and positive and negative whole numbers

Real rationals and irrationals

Boolean true and false logical values

Character single alpha characters

String sequences of characters

Array homogeneous collections of elements accessed with an index.

Each of the classes representing these data types has an attribute `imp-type-size` that indicates the number of bytes associated with the data type. Some imperative languages allow the programmer to specify the size of the data type at the declaration of a variable. Other languages do not allow this, but include types of different sizes to meet data storage requirements. Programmer-defined data type sizes are modeled in the GIM by storing the specified size in the `imp-type-size` attribute. For languages that do not allow the programmer to define the size of the data type, the size of the built-in type must be determined and stored into the `imp-type-size` attribute.

For example, in the FORTRAN programming language, the programmer is allowed to declare the size of the data type when declaring variables. The `MFPGRC` variable from Section 3.14 can be declared as shown below.

```
        INTEGER*4 MFPGRC
```

The integer data type is represented using the `imperative-integer` object. Figure 28

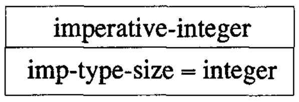


Figure 28 Imperative-Data-Type Class

shows the class description for `imperative-integer`. The `imp-type-size` attribute indicates the size of the data type. The four byte FORTRAN integer data type used to declare `MFPGRC` in the example above is represented using the `imperative-integer` object shown in Figure 29.

In the C programming language, the programmer must use intrinsic data types to specify the size of data storage locations. The `int` data type in C is normally the natural

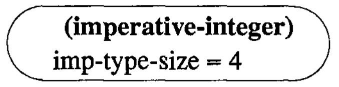


Figure 29 Four Byte Integer Object

size for a particular host machine: either 16 or 32 bits [35]. The `short` or `long` qualifier can be applied to the basic C types to change the number of bytes used for the variable. A `short` integer is often 2 bytes and a `long` integer is often 4 bytes [35]. In order to model these data types in the GIM, the size of the intrinsic data types and qualifiers must be determined for a specific compiler in order to store the correct value for the `imp-type-size` attribute of the imperative data type objects.

In C, the `MFPGRC` variable can be declared as follows:

```
short MFPGRC;
```

Here, an `imperative-integer` object with the `imp-type-size` attribute set to 2 correctly models the data type of the `MFPGRC` variable, as shown in Figure 30.

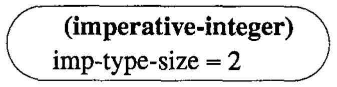


Figure 30 Two Byte Integer Object

3.16 Imperative Scoping Issues

In the imperative paradigm, the "visibility" a variable has to subprograms is known as the *scope* [72] of the variable. A subprogram that defines a variable has visibility to it, but different imperative programming languages provide different rules for the visibility of the variable to other subprograms.

Each subprogram in the GIM has a *local* scope in which the variables declared are not visible to any other subprograms. The GIM subprogram that declared the variable can share its value by passing the variable to other subprograms as a parameter. Passing a local variable to another subprogram as an actual parameter has the effect of making the variable visible to the called subprogram. In the imperative paradigm, this is termed *pass by reference* parameter passing [16]. All parameters in the GIM are passed by reference.

Variables declared in the main program are considered to be in the *global* scope. Some imperative programming languages allow subprograms to directly access variables declared in the global scope, i.e. without passing the variable to the subprogram via a parameter. This is not modeled in the GIM and leads to the following restriction on variable scoping in the GIM.

Restriction III.6. *All variables in a subprogram are either declared locally or are formal parameters of the subprogram.*

In the imperative paradigm, some programming languages allow subprograms to be declared inside another subprogram. This leads to a nesting of variable scopes where variables in the "outer" subprogram are visible to the "inner" subprogram. Declaring a subprogram inside another subprogram is not modeled in the GIM which leads to the following restriction on declaring subprograms in the GIM.

Restriction III.7. *Subprograms can not be declared inside of another subprogram. They are all declared in the main program's global scope.*

3.17 Imperative Homogeneous Data Structures

In the imperative paradigm, collections of homogeneous elements can be built using constructs such as *lists* and *arrays*. The only such collection modeled in the GIM is the array. An *array* is an indexable collection of homogeneous elements. Variables in the GIM can be of the data type array. The indices used to access an array element are modeled using the `imp-indices` attribute of the `imperative-variable` class.

For example, an access to the first element of a FORTRAN array `R` used in a subprogram `VADD` is modeled in the GIM as shown in Figure 31 The `imp-data-indices` attribute models the expression or variable being used to access the element of the array. In the figure, the access to the first element of the array `R` is modeled by storing the literal value `1` as the index into the array.

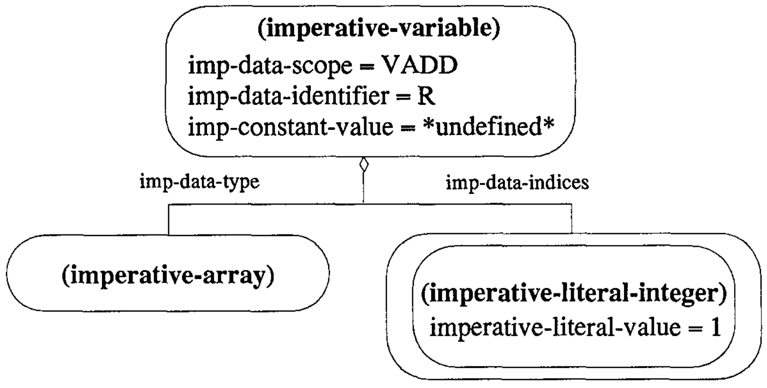


Figure 31 Imperative Array Access Object

3.18 Imperative Heterogeneous Data Structures

Some programming languages in the imperative paradigm, allow the programmer to build collections of heterogeneous data items, e.g. *records* in Pascal and *structs* in C. These heterogeneous data structures are not modeled in the GIM, which leads to the following restriction.

Restriction III.8. *The GIM does not model heterogeneous data structures.*

3.19 Imperative Pointers

Some imperative programming languages allow the programmer to store the address of a variable and then access the variable via the address. This mechanism is typically referred to as using *pointers* to variables. Pointers are not modeled in the GIM, which leads to the following restriction.

Restriction III.9. *The GIM does not model pointers.*

3.20 Imperative Expressions

In the imperative paradigm, expressions can be literal values, unary or binary expressions with operators, function calls, or even variables. This section describes how expressions are modeled in the GIM.

Literal values in the GIM are modeled by the `imperative-literal-constant` class. Figure 32 shows the class description for the `imperative-literal-constant` class. The

imperative-literal-constant
imperative-literal-value : any-type

Figure 32 Modeling Imperative Literal Constants

imperative-literal-value attribute models the value of the literal. Figure 33 shows all

imperative-literal-boolean
imperative-literal-integer
imperative-literal-real
imperative-literal-string

Figure 33 List of Imperative Literal Constants

the literal constants currently modeled in the GIM. Each literal shown is modeled as an AST defined as a subclass of the **imperative-literal-constant** class. Literal constants other than those listed in Figure 33 are not modeled in the GIM, but could easily be modeled by defining a new subclass under the **imperative-literal-constant** class that describes the new literal constant.

Unary expressions in the GIM are modeled using the **imperative-unary-expression** class. Figure 34 shows the class description for the **imperative-unary-expression** class.

imperative-unary-expression
imp-unary-operand : imperative-expression

Figure 34 Modeling Imperative Unary Expressions

The **imp-unary-operand** attribute models the single operand of the unary expression. Figure 35 shows all the unary expressions currently modeled in the GIM. Each unary expression shown is modeled as an AST defined as a subclass of the **imperative-unary-expression** class. As with literal constants, any unary expressions other than those listed in Figure 35 are not modeled in the GIM but can be by building a new subclass that defines the new unary expression.

Binary expressions in the GIM are modeled by the **imperative-binary-expression** class. Figure 36 shows the class description for **imperative-binary-expression** class. Binary expressions in the imperative paradigm either have two operands or multiple operands.

imperative-negate
imperative-not

Figure 35 List of Imperative Unary Expressions

imperative-binary-expression
imp-bin-exp-operand-1 : imperative-name
imp-bin-exp-operand-2 : imperative-data-construct
imp-bin-exp-seq : seq(imperative-expression)

Figure 36 Modeling Imperative Binary Expression

For this reason, the `imperative-binary-expression` class has three attributes for modeling the operands of the expression. The `imp-bin-exp-operand-1` and `imp-bin-exp-operand-2` attributes hold the two operands if the binary expression has only two operands. When the same binary expression is repeated for multiple operands, such as in the expression `A + B + C`, the multiple operands are modeled in the `imp-bin-exp-seq` attribute. Certain binary expressions can be repeated in this way, e.g. addition and subtraction. Others can not be repeated in this way, e.g. <, <=, and >. Figure 37 lists all the binary expressions that are modeled in the GIM. The binary expressions that can not be repeated are shown in Figure 37(a). These expressions are modeled using the `imp-bin-exp-operand-1` and `imp-bin-exp-operand-2` attributes. The binary expressions that can be repeated are shown in Figure 37(b). These expressions are modeled using the `imp-bin-exp-seq` attribute.

imperative-division
imperative-equal
imperative-exponent
imperative-greater-than-or-equal
imperative-greater-than
imperative-less-than-or-equal
imperative-less-than
imperative-not-equal
imperative-subtraction

(a) Two operand.

imperative-addition
imperative-and
imperative-concat
imperative-multiplication
imperative-or

(b) Sequence.

Figure 37 List of Binary Expressions Modeled in the GIM

As with literal constants and unary expressions, any binary expressions not shown in Figure 37 are not modeled by the GIM. This new binary expression could be modeled by adding a subclass to the `imperative-binary-expression` class that defines the new binary expression.

3.21 Imperative Input

Programming languages in the imperative paradigm allow the programmer to interact with the user via input and output statements. These statements, if directed towards a file, allow the programmer to implement persistent storage of data. This section describes how input statements in imperative languages are modeled in the GIM.

In order to receive data values from outside an imperative program, imperative programming languages provide the programmer with an input statement. This statement can be used to communicate with the user or used to input data from a file. The Cobol programming language actually has two statements: one for inputting low-volume data and one for inputting data from a file. For example, the following Cobol `ACCEPT` statement is used to input a program name from the user using the console typewriter.

```
ACCEPT PROGRAM-NAME FROM CONSOLE.
```

Input statements are modeled in the GIM using the `imperative-input` object. Figure 38 shows the class description for this object. The `imp-in-logical-file` attribute

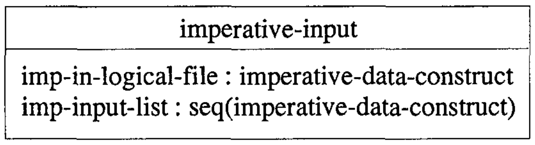

imperative-input
imp-in-logical-file : imperative-data-construct
imp-input-list : seq(imperative-data-construct)

Figure 38 Imperative Input Class

is used to model from where the input is coming. When the input comes from the user, the attribute is set to an accepted value that models that fact. The `imp-input-list` attribute holds the sequence of data items being input. Figure 39 shows how the Cobol `ACCEPT` statement is modeled in the GIM, assuming the `PROGRAM-NAME` variable is defined in a procedure called `GET-INFO`. The `imp-in-logical-file` attribute, in this example, has been arbitrarily set to the symbol `console` in order to represent input from the user

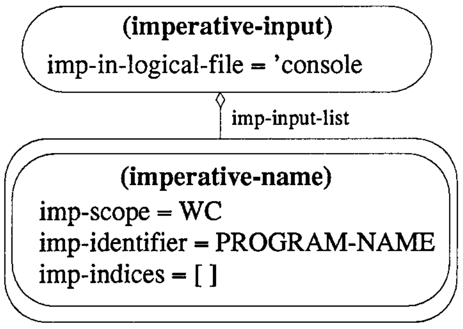


Figure 39 Imperative Input Object

console. As long as this attribute is set consistently, receiving input from the user can be modeled in this way.

In imperative languages, when the programmer wants to input from a file, the file must first be "opened". The act of opening a file establishes the link between the physical file on disk and the logical file in the program. The status of the file, e.g. opened for input or output, and the access mode of the file, e.g. direct or sequential, are also established when the file is opened. This information about the file is modeled in the GIM using the `imperative-file` object. Figure 40 shows the class description for this object. The

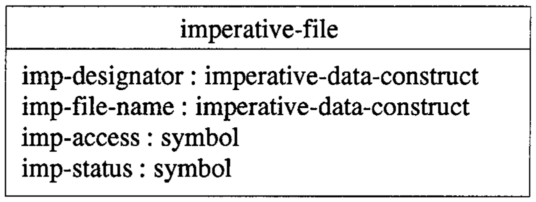

imperative-file
imp-designator : imperative-data-construct
imp-file-name : imperative-data-construct
imp-access : symbol
imp-status : symbol

Figure 40 Imperative File Class

`imp-designator` attribute holds name of the logical file as referenced in the program. The `imp-access` attribute holds the access type, direct or sequential, for the file. The `imp-status` attribute holds the status of the file, opened either for input or output. In Cobol, the `OPEN` statement is used to open a file. The following Cobol `OPEN` statement opens a file `MASTER-FILE` for input. The access mode is assumed to be sequential.

```
OPEN INPUT MASTER-FILE
```

Figure 41 shows how this file is modeled in the GIM. The name of the file being opened is used as the file designator.

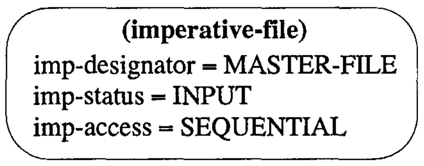


Figure 41 Modeling MASTER-FILE

Once files are opened, input statements are used to input data from the files. In Cobol, the READ statement is used to input data from a file. For example, the following READ statement reads the next record from the file MASTER-FILE and stores it into the variable MASTER-WORK.

```
READ MASTER-FILE INTO MASTER-WORK
   AT END PERFORM END-DATA-MASTER.
```

Figure 42 shows how this READ statement is modeled in the GIM. The imp-in-logical-

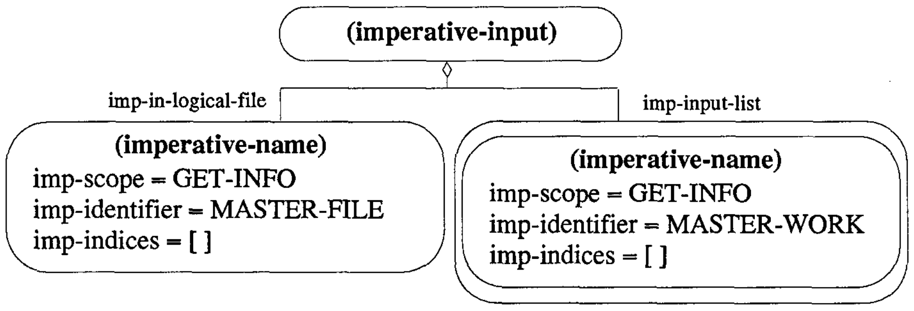


Figure 42 Modeling the COBOL Read

file attribute, in this example, has been set to an imperative name object referring to the file MASTER-FILE in order to indicate the input is coming from this specific file.

The semantics of the GIM input statement are similar to the semantics of the GIM assignment statement. Both statements define values for variables. Abstractly, an input statement in the GIM is equivalent to calling a subprogram where all the parameters are output from the subprogram.

For example, let I represent an input statement and let $\bar{a}$ represent a vector holding the variables that appear in I. Let $\bar{x}$ represent a vector of variables returned from the input statement after the values are read from the indicated file (or the console). Then, execution of the GIM input statement is equivalent to the following sequence:

$$[I,\ \bar{a} := \bar{x}]$$

The formal semantics for the input statement I are defined below.

Definition III.13. $wp(I(\bar{a}),\ R) \ = \ wp([I(\bar{a}),\ \bar{a} := \bar{x}],\ R)$

3.22 Imperative Output

This section describes how output statements in imperative languages are modeled in the GIM. In order to communicate the values of variables outside a program, imperative programming languages provide the programmer with an output statement. This statement is used to communicate with the user or store values in data files. The output statement is modeled in the GIM using the `imperative-output` object. Figure 43 shows

imperative-output
imp-out-logical-file : imperative-data-construct imp-output-list : seq(imperative-data-construct)

Figure 43 Imperative-Output Class

the class description for this object. The `imp-out-logical-file` attribute is used to store the logical link to a file opened for output. The `imp-output-list` attribute holds the list of data items to be output. The data items are modeled using `imperative-output-item` objects. Figure 44 shows the class description for the `imperative-output-item` object.

imperative-output-item
imp-out-item : imperative-expression imp-format : imperative-format

Figure 44 Imperative Output Item

The `imp-out-item` attribute holds the expression to be output. The `imp-format` attribute holds formatting information for the item to be output. This attribute models the output formatting implemented in some imperative languages, e.g. total number of spaces in the output or number of digits to output after the decimal point. If the output item is not being formatted in any way, the imp-format attribute is empty.

In COBOL, there are two output statements: one for low-volume output and the other for output to a data file. The DISPLAY statement is used in COBOL to output data to the user. For example, the following COBOL DISPLAY statement is used to prompt the user for the program name outputting to the console screen.

```
DISPLAY 'ENTER PROGRAM NAME'
    UPON CONSOLE.
```

Figure 45 shows how the COBOL DISPLAY statement is modeled in the GIM. In this

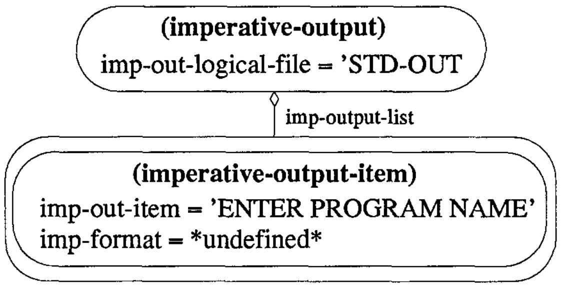


Figure 45 Modeling the COBOL Display

example, the imp-out-logical-file attribute has been set to the symbol STD-OUT to indicate output to the standard output port. As with input, as long as the standard I/O ports are modeled consistently, output to the user can be modeled in this way. The imp-format attribute is empty indicating this output is not formatted.

In COBOL, the WRITE statement is used when the programmer wants to output data values to a file. After the file has been opened for output, a data record is associated with the file and this data record can be output to the file. While this mechanism is more complicated than the mechanism in COBOL for reading from a file, writing to a file can still be modeled in the GIM by determining with which file a record is associated and storing this information in the imperative-output object. For example, the following COBOL program, ADDRESS-LISTING, declares a file ADDRESS-LIST, declares a record OUTPUT-RECORD associated with the file, opens the file for output, and writes a record to the file.

```
PROGRAM-ID.
     PROGRAM-ADDRESSES.
FILE-CONTROL.
     SELECT ADDRESS-LIST     ASSIGN TO UR-1403-S-OUTFILE.
FILE SECTION.
FD   ADDRESS-LIST
01   OUTPUT-RECORD.
     05 LAST-NAME       PICTURE X(12).
PROCEDURE DIVISION.
MAIN-ROUTINE.
     OPEN   OUTPUT   ADDRESS-LIST
     ACCEPT OUTPUT-RECORD FROM CONSOLE.
     WRITE OUTPUT-RECORD.
```

Figure 46 shows how the COBOL **WRITE** statement is modeled in the GIM. In this example,

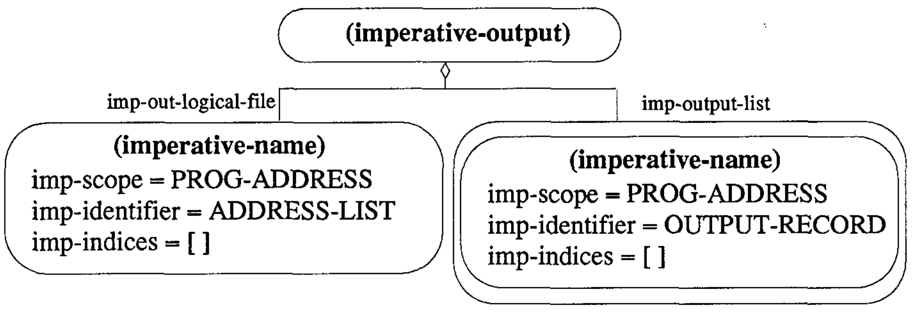


Figure 46   Modeling the COBOL Write

the connection between **OUTPUT-RECORD** and the file **ADDRESS-LIST** is used to determine the value of the **imp-out-logical-file** attribute. Even though the logical file is not explicitly referenced in the **WRITE** statement, the indirect reference can be used to model output to the file.

The semantics of the GIM output statement are defined using the weakest precondition notation. Semantically, the output statement is not allowed to change the state of the program. Let O represent an output statement in the GIM. The formal semantics of the output statement are defined as

Definition III.14. $wp(O,\ R) = R$

This means the GIM output statement can have no side-effects that would change the state. In this way, the input and output statements provided by imperative programming languages are modeled in the GIM.

The following sections provide formal definitions for imperative subprograms and subprogram calls. These definitions are used throughout the rest of this document to build formal transformations.

3.23 Formalizing GIM Statements and Expressions

This section presents formal representation for GIM statements and expressions. These formal representations are used in the formal transformations presented in Chapter VI. Let x represent a GIM variable and let e represent a GIM expression. A GIM assignment statement is represented formally by the following tuple.

$$< x,\ :=, \mathrm{e} >$$

This tuple indicates that x is the variable to be assigned the value of the expression e. The $:=$ symbol is a syntactic token that distinguishes this tuple as an assignment statement.

Let e represent a GIM expression and let S_1 and S_2 represent sequences of GIM statements. A GIM selection statement is represented by the following tuple.

$$< \mathrm{if}, e,\ \mathrm{then}, S_1,\ \mathrm{else}, S_2 >$$

This tuple indicates that the sequence of statements S_1 will be executed if the expression e evaluates to true otherwise the S_2 sequence will be executed. The `if`, `then`, and `else` symbols are syntactic tokens used to identify this tuple.

Similarly, let e represent a GIM expression and let S_3 represent a sequence of GIM statements. The GIM iteration statement is represented by the following tuple.

$$< \mathrm{while}, e, S_3 >$$

This indicates that the sequence of statements S_3 will be executed while the expression e is true. The `while` symbol is a syntactic token used to identify this tuple.

Let E represent a sequence of expressions. The GIM input and output statements are represented formally by the following tuples.

$$\begin{gathered} <\text{ input, iport}, E > \\ <\text{ output, oport}, E > \end{gathered}$$

These tuples indicate the expressions in the sequence E are either input from the input port `iport` or output to the output port `oport`.

Expressions in the GIM are represented formally by the following tuples. Let e_1 and e_2 be GIM expressions.

$$\begin{aligned} < e_1, +, e_2 > \\ < e_1, \text{ and}, e_2 > \\ < e_1, \text{ concat}, e_2 > \\ < e_1, \ /, e_2 > \\ < e_1, =, e_2 > \\ < e_1, \ {}^{**}, e_2 > \\ < e_1, >=, e_2 > \\ < e_1, >, e_2 > \\ < e_1, <=, e_2 > \\ < e_1, <, e_2 > \\ < e_1, \ {}^{*}, e_2 > \\ < e_1, <>, e_2 > \\ < e_1, \text{ or}, e_2 > \\ < e_1, -, e_2 > \\ < -, e_1 > \\ < \text{ not}, e_1 > \end{aligned}$$

3.24 Formalizing Imperative Subprograms

This section explains how subprograms modeled by the GIM `imperative-procedure` and `imperative-function` ASTs can be represented more mathematically.

The following definitions are used to formally define subprograms.

id_S - A symbol storing the subprogram's name.

P_{in} - The sequence of input parameters.

P_{out} - The sequence of output parameters.

P_{ret} - The sequence of returned values.

P_{loc} - The sequence of local variables.

Σ - The sequence of statements from the subprogram.

Thus, a subprogram is defined by the following tuple.

$$< id_S, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma >$$

Several of the attributes of imperative subprograms are sequences of imperative variables, viz. P_{in}, P_{out}, P_{ret}, and P_{loc}. In order to manipulate these sequences formally, the four special sequence operators, $\oplus$, $\ominus$, $\sqsubseteq$, and *pos* are defined below.

$S_1 \oplus S_2 \;=\; S_1 \; concat \; [x \mid x \in S_2 \;\wedge\; x \notin S_1]$

$S_1 \ominus S_2 \;=\; [x \mid x \in S_1 \;\wedge\; x \notin S_2]$

$S_1 \sqsubseteq S_2 \;=\; \forall\, x \in S_1 \;\Rightarrow\; x \in S_2$

$pos(x, \Sigma)$ Let the *pos* operation indicate the ordinal position of the element x in the sequence Σ.

For example, Figure 47 shows the imperative subprogram `UPLREQ`. This subprogram is formally defined by the following tuple.

```
      DOUBLE PRECISION function UPLREQ(ZCOS,XLAMDA,SIGTRB)
       INCLUDE 'bdincl.f'
C
      DATA SIGT0/3.75D-06/
C
      TMP1 = XLAMDA*ZCOS*ZCOS*ZCOS
      TMP2 = TMP1**0.200
C
C      SIGUP IS THE UPLINK BEAM DIVERGENCE DUE TO TURBULENCE FOR
C       A BEAM WITH NO PHASE CORRECTION.
C
      SIGUP = SIGT0/TMP2
C
      IF(SIGTRB.GT.0.0) THEN
      UPLREQ = SIGUP/SIGTRB
      ELSE
      UPLREQ = 1000000.0D0
      END IF
      RETURN
      END
```

Figure 47 FORTRAN Function UPLREQ

$$
\begin{aligned}
&< UPLREQ, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma > \text{ where} \\
&\quad P_{in} = [\text{ZCOS, XLAMDA, SIGTRB}] \ \mathit{and} \\
&\quad P_{out} = [\,] \ \mathit{and} \\
&\quad P_{ret} = [\text{UPLREQ}] \ \mathit{and} \\
&\quad P_{loc} = [\text{SIGT0, TMP1, TMP2, SIGUP}] \ \mathit{and} \\
&\quad \Sigma = \text{ statements between DATA and END}
\end{aligned}
$$

The input parameters of UPLREQ are the data items ZCOS, XLAMDA, and SIGTRB. UPLREQ has no output parameters and the data item returned from the execution of the UPLREQ function is represented by including the identifier UPLREQ in the P_{ret} sequence. The local variables of UPLREQ are the data items SIGT0, TMP1, TMP2, and SIGUP. The sequence of imperative statements between the BEGIN and the END of UPLREQ comprise Σ.

3.25 Formalizing Imperative Subprogram Calls

This section provides formal definitions of the GIM `imp-procedure-call` entity, a kind of statement, and the GIM `imperative-function-call` entity, a kind of expression.

The calling relationship between any two subprograms can be defined formally as follows. Let S_1 be the calling subprogram and let S_2 be the called subprogram. Let id_{S_2} be a symbol that holds the name of the called subprogram. Let $P_{forms_{S_1}}$ be a sequence of variables that are formal parameters of the calling subprogram, i.e. $(P_{ins_{S_1}} \oplus P_{outs_{S_1}})$. Let $P_{acts_{S_1}}$ be a sequence of variables that are actual parameters in the calling subprogram, where $P_{acts_{S_1}} \sqsubseteq (P_{forms_{S_1}} \oplus P_{locs_{S_1}})$. Thus, a call to the subprogram S_2 by subprogram S_1 is represented by the following tuple.

$$< id_{S_2}, P_{acts_{S_1}} >$$

In the imperative paradigm, the number, order, and type of parameters in $P_{acts_{S_1}}$ must match the number, order, and type of the parameters in $P_{forms_{S_2}}$ of the called subprogram.

This link between an actual parameter in a call to a subprogram and the formal parameter in the called subprogram is important in re-engineering. Even though one subprogram can have multiple calls to another subprogram or can call many other subprograms, a static mapping between actuals and formals *of each call* can be built. For each call to a subprogram, the order of the actual parameters determines to which formal parameter in the called subprogram the actual parameter is mapped.

Definition III.15. *The μ map links an actual parameter from the calling program to a single formal parameter in the called subprogram.*

Let l be a call from subprogram S_1 to subprogram S_2. Let p_l be an actual parameter in l. Let p_2 be a formal parameter in the subprogram S_2. Let $pos(p_l, P_{act_l})$ indicate the position the parameter p_l takes in the sequence of actual parameters P_{act_l}. Let $pos(p_2, P_{form_2})$ indicate the position of p_2 in the sequence of formal parameters P_{form_2}.

The μ relation is defined formally as follows.

$$\begin{aligned}
\mu(p_l) \;=\; & p_2 \text{ where} \\
& l \;= <id_{S_2}, P_{act_l}> \;\; and \\
& S_2 = <id_{S_2}, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma> \;\; and \\
& p_{form} = \; P_{in} \oplus P_{out} \; and \\
& p_l \in P_{act_l} \; and \\
& p_2 \in P_{form} \; and \\
& pos(p_l,\; P_{act_l}) \;=\; pos(p_2,\; P_{form})
\end{aligned}$$

```
      DOUBLE PRECISION function PRDIV(BETA,XLAMDA,DIAM)

      INCLUDE 'bdincl.f'
C
C      CALCULATES THE PROJECTOR BEAM DIVERGENCE FOR A GIVEN OPTICS
C        SIZE THIS IS BASED ON THE RAYLEIGH (AIRY DISK) CRITERION.
C
      DATA FAC/1.220D0/

      QUAL = MAX(BETA,1.0D0)
      PDIAM = MAX(DIAM,0.1D0)
      WAVELN = XLAMDA*1.D-06
      PRDIV = QUAL*WAVELN*FAC/PDIAM
      RETURN
      END
```

Figure 48   FORTRAN Function PRDIV

For example, Figure 48 shows the declaration of the imperative subprogram `PRDIV` that has three formal parameters, `BETA`, `XLAMDA`, and `DIAM`.

Figure 49 shows the partial declaration of the subprogram `RELAY` and the call to the subprogram `PRDIV`. In this example, let

$$l \;= <\text{PRDIV},\;\; [\text{B, X, DIAM}]>$$

Thus, the actuals in l and the formals from `relay` are defined as follows.

$$\begin{aligned}
P_{act_l} &= [\text{B, X, DIAM}] \; and \\
P_{form_{\text{PRDIV}}} &= [\text{BETA, XLAMDA, DIAM}]
\end{aligned}$$

```
      SUBROUTINE RELAY(B,X,DIAM)
       INCLUDE 'bdincl.f'
C
C      DESE RESEARCH AND ENGINEERING, INC. (21 MAY 1985)
C
C
C      FIRST THE PROJECTOR DIVERGENCE
C
      SIGPR = PRDIV(B,X,DIAM)
      SIGPSQ = SIGPR*SIGPR
C
      RETURN
      END
```

Figure 49 FORTRAN Subroutine RELAY

Let $\xrightarrow{\mu}$ represent a μ mapping from an actual parameter to a formal parameter. Hence, the following mappings are included in the μ relation.

$$\begin{array}{rcl} \text{RELAY::B} & \xrightarrow{\mu} & \text{PRDIV::BETA} \\ \text{RELAY::X} & \xrightarrow{\mu} & \text{PRDIV::XLAMDA} \\ \text{RELAY::DIAM} & \xrightarrow{\mu} & \text{PRDIV::DIAM} \end{array}$$

Note that the identifiers used to name the parameters may or may not be the same. In the example, the scope of the parameter is used to differentiate between actuals and formals using the same identifier. For example, the `PRDIV::` prefix indicates the scope of a variable is `PRDIV`. In this way, the `DIAM` variable from `PRDIV` is distinguished from the `DIAM` variable in `RELAY`.

3.26 Formalizing Imperative Designs

This section provides a formal representation for imperative designs. Formal transformations that add subprograms to a design and remove subprograms from a design are also provided.

The formal definition of an imperative design is provided below. Let S^* represent the set of subprograms included in the design. An imperative design, SD, is represented formally by the following tuple.

$$SD = S^*$$

An imperative design consists of the set of imperative subprograms to be included in the design.

Let T_s^+ be a transformation of an imperative design that adds a new subprogram to an existing design and returns a new imperative design. Let SD represent an imperative design and let S represent the subprogram to be added to the design. The transformation is defined below.

$$\begin{aligned} T_s^+(SD, S) &= SD' \text{ where} \\ SD' &= SD \cup \{S\} \end{aligned}$$

Let T_s^- be a transformation of an imperative design that removes a subprogram from an existing design and returns a new imperative design. Let SD represent an imperative design and let S represent the subprogram to be removed from the design. The transformation is defined below.

$$\begin{aligned} T_s^-(SD, S) &= SD' \text{ where} \\ SD' &= SD - \{S\} \end{aligned}$$

3.27 Summary

This chapter has defined the Generic Imperative Model (GIM) that is used to model assignment, variables, expressions, and control flow constructs in imperative programming languages.

Several restrictions have been placed on the GIM. Some of these restrictions are meant to canonicalize the representations in the GIM. Others place limits on which imperative entities can possibly be modeled using the GIM. Specifically, Restriction III.1 states that a parameter of a procedure can not be both an input and an output parameter. This restriction is meant to simplify the way parameters are modeled. This restriction is not unreasonable since any procedures that do not meet the restriction can be converted using the process presented in Appendix D. Similarly, Restriction III.2 does not allow functions to include output parameters or return more than a single data item. It is hypothetically possible to convert such functions into procedures and then represent the procedures in the GIM. Restriction III.3 requires all parameters in subprogram calls to be variables. Parameters that are not variables are easily converted to a temporary variable, so this is a reasonable restriction of the GIM. Restrictions III.4 and III.5 limit subprogram calls to non-recursive calls. This restriction is reasonable because Knuth [37] argues any recursive algorithm can be represented using an iterative algorithm. Restriction III.6 does not allow *global* variables to appear in subprograms. This means subprograms with variables other than formal parameters or local variables must first be converted into the proper form. It is hypothetically possible to convert global variables into formal parameters throughout the call tree of a legacy system, so this restriction is not unreasonable. Restriction III.7 does not allow one subprogram to be declared inside another subprogram. The scoping issues involved in this nesting of subprograms is more complicated than that of imperative variables, but it is hypothetically possible to convert the nested subprograms into subprograms declared at the global level.

None of the restrictions discussed above preclude the representation of imperative entities in the GIM. However, Restriction III.8 states that heterogeneous data types are not modeled in the GIM. Restriction III.9 states that pointers are not modeled in the GIM. These two restrictions limit the entities that can be modeled by the GIM. If the legacy system includes either heterogeneous data types or pointers, the system can not be represented using the GIM. This is not unreasonable for the FORTRAN language, but is a limitation for languages such as C or Pascal. The GIM is easily extended by adding new ASTs to represent the new constructs and by defining formal semantics for them using weakest preconditions.

Overall, the GIM provides a canonical form for representing imperative programs. The GIM is used throughout the following chapters to represent subprograms that are being converted from the imperative paradigm to the object-oriented paradigm.

IV. The Generic Object Model

4.1 Introduction

This chapter defines the Generic Object-Oriented Design Model (GOM) which has been developed to model the objects, classes, methods, and messages typically built as part of an object-oriented programming language. A general discussion of the object-oriented paradigm is presented first, followed by detailed descriptions of how the entities in this paradigm are modeled using abstract syntax trees (ASTs). The classes that define the GOM ASTs are presented using Rumbaugh's OMT notation [62]. The formal semantics of the program constructs are defined using the ***weakest precondition*** notation.

In order to scope this research, the GOM was built to model OO languages that use sequences of imperative statements in the methods. Usually these languages started in the imperative paradigm and OO extensions were added in new versions of the languages, e.g. Ada 95 and C++. For this reason, imperative assignment, sequential control, selective control, and iterative control constructs are included in the GOM. These constructs must be adapted for use in the GOM because GOM expressions now include accesses to object attributes (as explained in Section 4.6) and possibly messages to other objects (as explained in Section 4.8).

Since the GOM provides a canonical form for representing object-oriented languages, OO languages such as Ada 95, C++, and Java can be modeled using the GOM. Since the focus of this research was to produce an OO design, no attempt was made to recover GOM ASTs from OO code. Instead, rudimentary transformations from the GOM to Ada 95 were developed as a proof-of-concept prototype. This prototype research is discussed with the feasibility demonstration in Chapter VIII. For these reasons, throughout the rest of this chapter, examples are given that have more of a forward engineering flavor (from the GOM to a specific language) than a reverse engineering flavor (from an OO language to the GOM).

In order to visualize the ASTs built in the GOM, a surface syntax has been developed for each entity in the GOM. The surface syntax is termed the Generic Object-Oriented

Language (GOL) and is shown in Appendix B. The GOL was built only as a tool for viewing the GOM ASTs and is not intended as a new object-oriented programming language.

4.2 The Object-Oriented Paradigm

As discussed in Chapter III, the object-oriented paradigm is one of the four programming language paradigms [16]. Korson and McGregor [38] characterize the object-oriented paradigm using the following concepts:

Classes A class is a template that defines the attributes and operations for each instance of the class.

Objects An object is an instance of a class. Objects model real-world entities that have *state*, *behavior*, and *identity.*

Methods A method is a sequence of object-oriented statements that implement a specific behavior.

Messages A message invokes a specific method in an object. Messages are sent to a *target* object that must be able to execute the method being invoked.

Inheritance The classes in an object-oriented design are organized in a *class hierarchy* where certain classes *inherit* the attributes and operations from other classes in the hierarchy.

Polymorphism In an object-oriented design, it is possible to have methods (from different classes) with the same name. Polymorphism means the appropriate method will be executed based on the class of an object instance.

The GOM provides a canonical form for modeling each of these aspects of object-oriented languages. The object-oriented statements in each method in the GOM are assumed to be based on *imperative* control flow constructs. It is also assumed that one special method is given the flow of control as the object-oriented system begins execution. This special method is termed the *main method.* The following sections define the ASTs that store the knowledge about these entities. Where applicable, the formal semantics of these entities is also defined.

Because of the intertwining of the aspects of the OO paradigm, it is hard to present them in an order that does not include any forward references to other aspects. Because of this, it is assumed that the reader has a rudimentary understanding of the Object-Oriented paradigm as aspects of the GOM are discussed. There are several good references that explain the OO paradigm in great detail, and the reader is referred to them [9,10,38,62].

4.3 The GOM Domain Model

This section presents a brief overview of the ASTs that are included in the GOM domain model. Figure 50 shows a partial representation of the GOM domain model. The

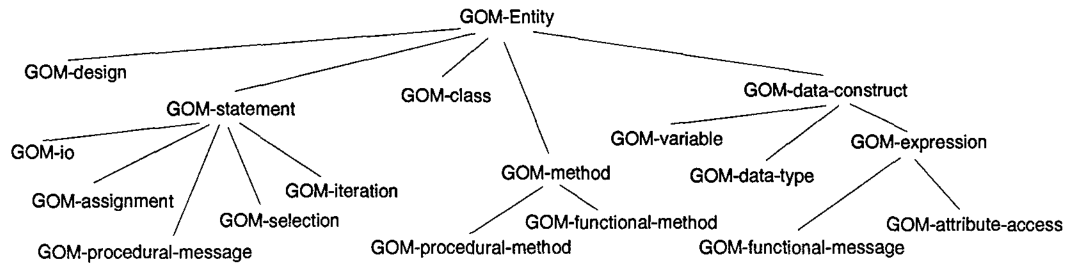


Figure 50 GOM Domain Model

overall superclass of the domain is the **GOM-entity** AST. The **GOM-design** class models collections of classes as defined in Section 4.5. The abstract class **GOM-statement** is the superclass for all object-oriented programming statements modeled in the GOM. The **GOM-data-construct** class is the superclass of object-oriented expressions, data types, attribute accesses, functional messages, and variables modeled in the GOM. Each of the lower-level classes in the GOM domain model are described in the rest of this chapter.

4.4 Classes

In the OO paradigm, *classes* define templates for objects. They define the attributes each instance of the class will have and store the behavior each instance is capable of executing. Classes are modeled in the GOM using the **gom-class** class. Figure 51 shows

gom-class
gom-name : symbol gom-attrs : set(gom-attribute) gom-opers : set(gom-method) gom-super : symbol

Figure 51 GOM Class Class

the class description for the gom-class class. The gom-name attribute holds a symbol that represents the name of the class. The gom-attrs attribute of this class holds the attributes of classes being modeled in the GOM. The gom-opers attribute of this class holds the methods for the class being modeled. The gom-super attribute holds the symbol representing the superclass from which this class inherits. The symbol USER-OBJECT is provided to represent the overall superclass at the root of the class hierarchy. As an example,

```
class CLASS-5 attributes RHOSTD, LAMBDA, ANGFAC
  method RHO ( C-5 ): real
    begin
      RHO := GET-RHOSTD ( C-5)
    end
  superclass USER-OBJECT
```

Figure 52 Class CLASS-5

Figure 52 shows the class CLASS-5 (using GOL syntax). This class has three attributes named RHOSTD, LAMBDA, and ANGFAC and one method named RHO. The class inherits from the super class USER-OBJECT. Figure 53 shows how the class CLASS-5 is modeled in the

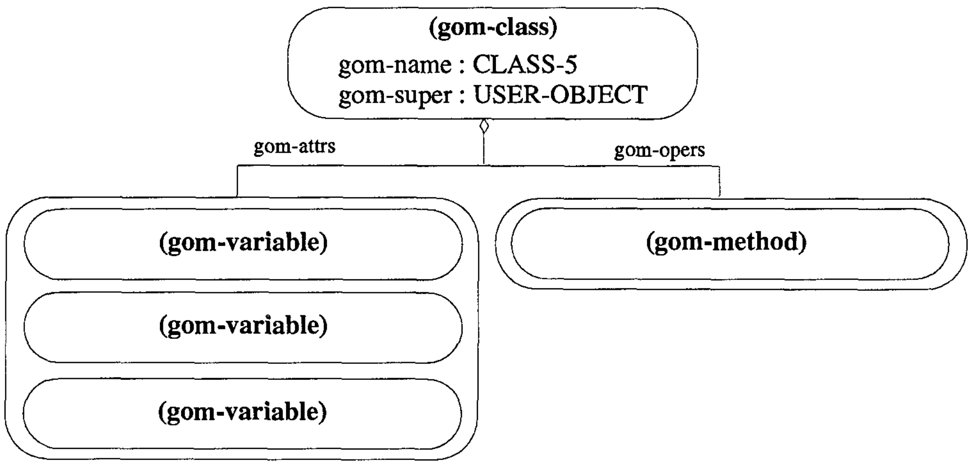


Figure 53 GOM Class Object

GOM. The attributes RHOSTD, LAMBDA, and ANGFAC are stored in the gom-attrs attribute. The method RHO is stored in the gom-opers attribute. The USER-OBJECT symbol is stored in the gom-super attribute to represent the fact that this class inherits from the class named USER-OBJECT.

Each class in the OO paradigm must be able to instantiate instances of the class. The **gom-instantiate** object is used to model this ability of a class. Since almost every OO language has a unique way to instantiate an instance of a class, the **gom-instantiate** object provides a canonical form for modeling object instantiation. Figure 54 shows the

gom-instantiate
gom-inst-class : symbol

Figure 54 GOM Instantiate Class

class description for the **gom-instantiate** object. The **gom-inst-class** attribute holds a symbol representing the name of the class being requested to instantiate a new instance. The GOL surface syntax for this class is the keyword **new** followed by the value of the **gom-inst-class** attribute. The **gom-instantiate** class models the intrinsic behavior of a class to create new instances. As with OO objects being modeled, there is no need to define formal semantics for OO classes being modeled. Classes do not directly affect the state of the OO system.

4.5 Modeling Object-Oriented Designs

In order to model the collection of classes in an OO design as a whole, classes are stored in an AST built from the GOM class **gom-design**. An instance of the **gom-design** class stores all the classes in the class hierarchy defined for a specific object-oriented design. Figure 55 shows the class description of the **gom-design** class. The **gom-classes** attribute

gom-design
gom-classes : seq(gom-class) gom-files : seq(gom-file)

Figure 55 GOM Design Class

holds the collection of classes that comprise the design. The **gom-files** attribute holds the collection of input/output files being modeled for this design. See Section 4.26 and Section 4.27 for discussions of how input and output statements and files are modeled in the GOM.

4.6 Objects

The basic building block for the object-oriented paradigm is the object. An *object* has state, behavior, and identity [9]. Objects are modeled in the GOM by using variables whose data type refers to a *class*. The variable is the handle into the object and is used to access the attributes of the object and to receive messages from other objects. Variables are discussed in Section 4.18, but an example of a variable used to model an object is presented here for clarity. Figure 56 shows an example of a variable that models an object

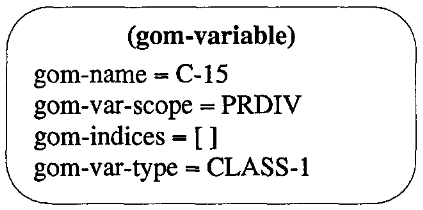

(gom-variable)
gom-name = C-15
gom-var-scope = PRDIV
gom-indices = [ ]
gom-var-type = CLASS-1

Figure 56   An Object Instance Variable

instance built from class `CLASS-1`. This variable is modeled by building an AST from the class `gom-variable`. In the figure, the `gom-name` attribute holds the symbol `C-15` which models the identifier for the variable. The variable is referenced by this identifier in GOM statements to provide access to the object and its attributes. The `gom-var-scope` in this example holds the symbol `PRDIV` to represent the fact that this variable is declared in the scope of the `PRDIV` method. Scoping issues for GOM variables are discussed in Section 4.21. The `gom-var-type` attribute is set to the data type `gom-instance` which indicates the variable is an instance of `CLASS-1`. For simplicity, the fact that `C-15` is an instance of `CLASS-1` is represented in the figure by using a symbol.

Each object from an OO program is an instance of a class, which defines the attributes of the object. Each object has a copy of these attributes defining a separate state space for each object instance. Each OO programming language provides a way to access these attributes of the object. The access of an object attribute is modeled in the GOM by the `gom-attr-access` class. Figure 57 shows the class definition for the `gom-attr-access` class. The `gom-tar-object` attribute holds the variable representing the target object whose attribute is being accessed. The `gom-attrib` attribute holds the variable representing the name of the attribute to be accessed. For example, Figure 58 shows an access

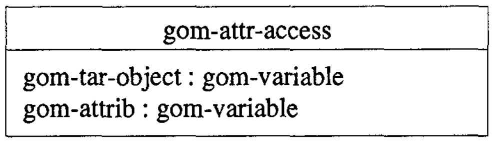


Figure 57 Attribute Access Class

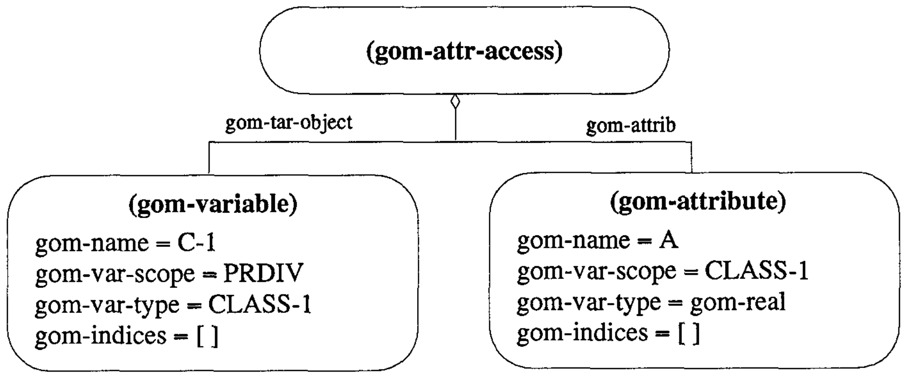


Figure 58 Attribute Access Object.

of the attribute `A` of the object `C-1`. The GOL surface syntax for this attribute access is `C-1.A`, thus `C-1` is the GOM variable holding the target object and `A` is the name of the attribute being accessed.

There is no need to define formal semantics for OO objects being modeled in the GOM. Objects hold pieces of the state of an OO system, but they do not change the state directly. The programming language statements in the methods of objects do change the state and formal semantics for these statements are defined in the sections that describe how these statements are modeled.

4.7 Methods

A method in the OO paradigm is a named sequence of object-oriented programming language statements that is stored in a class. The statements that are modeled in the GOM include assignment, sequential control flow, selective control flow, iterative control flow, and subprogram invocation of non-user-defined subprograms. A method has formal parameters and may or may not return a value. Methods are modeled in the GOM using ASTs built from the `gom-method` class. Figure 59 shows the class description for the `gom-method` class. The `gom-name` attribute stores a symbol representing the name of the method. The

gom-method
gom-name : symbol gom-params : seq(gom-parameter) gom-stmts : seq(gom-program-construct) gom-return-type : gom-data-type

Figure 59 GOM Method Class

gom-params attribute stores the formal parameters of the method. Each formal parameter is a GOM variable. The number, order, and type of these parameters is important for consistency and is always maintained. The **gom-stmts** attribute models the sequence of programming language constructs that comprise the method. The **gom-return-type** attribute models the return type (if any) of the method. If the method does not return a value, then this attribute is undefined.

```
class CLASS-5 attributes RHOSTD, LAMBDA, ANGFAC
  method RHO ( C-5 ): real
    begin
      RHO := GET-RHOSTD ( C-5)
    end
  superclass USER-OBJECT
```

Figure 60 Class CLASS-5

For example, Figure 60 shows the class CLASS-5 (using GOL syntax). CLASS-5 has a single method named RHO. This method has one formal parameter, C-5, which is the target object of the method and an instance of CLASS-5. The RHO method returns a value of type **gom-real** and has one statement in the body of the method. Figure 61 shows how the RHO method is represented using a **gom-method** AST. The name of the method, RHO, is stored in the **gom-name** attribute, the formal parameter C-5 is stored as the only element of the **gom-params** attribute[1]. The single statement is stored in the **gom-stmts** attribute.

Methods in the OO paradigm are analogous to imperative subprograms. Methods that return a value after execution are similar to imperative functions. This kind of method

[1]For simplicity, the fact that C-5 is an instance of CLASS-5 is represented in this diagram by using a symbol.

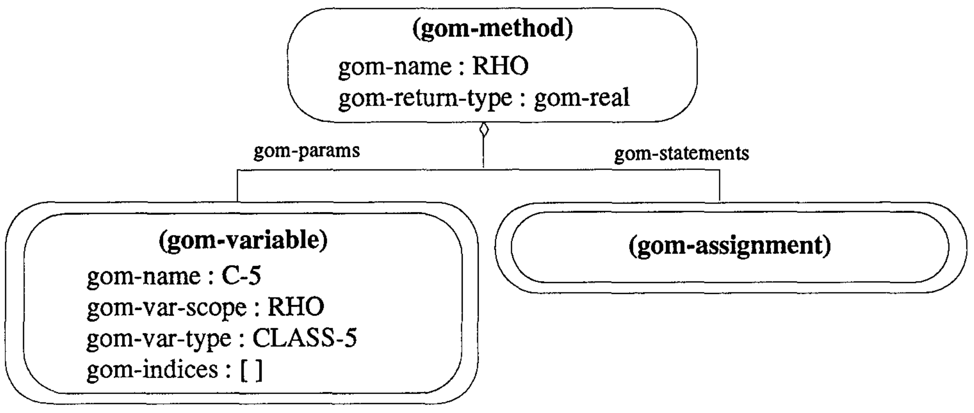


Figure 61 GOM Method Object

is referred to in the rest of this document as a *functional* method. Methods that do not return a value after execution, but, instead, communicate through input and output parameters are similar to imperative procedures. This kind of method is referred to in the rest of this document as a *procedural* method. Both methods and subprograms define formal parameters where the number, order, and type of the parameters must be matched by the actual parameters in a message or a subprogram call, respectively.

Because methods in the GOM are built from subprograms in the GIM, certain properties of GIM subprograms are carried forward in the transformation and now apply to GOM methods. Specifically, for both functional and procedural GOM methods, the following restriction applies.

Restriction IV.1. *A formal parameter of a method must not be both an* input *and an* output *parameter.*

For functional methods, the following restriction applies.

Restriction IV.2. *All functional methods must return a* single *value at the end of their execution and have no output parameters.*

As with GIM subprograms, the declaration of a GOM method does not change the state of a program. A *message* that invokes the method is allowed to change the state, thus the formal semantics for method declarations provided in this section serve as a foundation for the definition of the formal semantics of messages.

Given a precondition P, a postcondition R, and a sequence of OO statements $S1$, it is assumed that P and R exist such that the following holds.

$$\begin{array}{c} \{P\} \\ S1 \\ \{R\} \end{array}$$

For a method m, let $\bar{x}$ be a vector representing the input parameters of m. Let $\bar{z}$ be a vector representing the output parameters of m. A procedural GOM method takes the following general form.

$$\begin{array}{l} \text{method } m(\bar{x}, \bar{z}) \\ \quad \{P\} \\ \quad S1 \\ \quad \{R\} \end{array}$$

A functional GOM method takes the following general form.

$$\begin{array}{l} \text{method } m(\bar{x}) \\ \quad \{P\} \\ \quad S1 \\ \quad \{R\} \end{array}$$

In order to define the semantics for the value that is returned from the method m, a modified representation is used. Let y represent the value that is returned from m. Let m' be a method that takes the same input parameters as m, and includes y as the single output parameter. The statements in m' are the statements from m, viz. $S1$. The method m' takes the following form.

$$\begin{array}{l} \text{method } m'(\bar{x}, y) \\ \quad \{P\} \\ \quad S1 \\ \quad \{R\} \end{array}$$

These general forms define the formal semantics for method declarations in the GOM. To tie this general form to the specific representation in the GOM, recall that the sequence of OO statements, $S1$, in the method is modeled by the `gom-stmts` attribute in the GOM.

4.8 Messages

In the OO paradigm, objects communicate by sending messages. Messages are requests from one object (the sending object) to another object (the target object). If the target object has a method that corresponds to the message received, then the target object executes this method. Messages have actual parameters that must match in number, order, and type to the formal parameters of the requested method.

Rumbaugh [62] describes the abstract notion of *flow of control* in the OO paradigm as a collection of concurrently-active objects sending messages to each other. Rumbaugh uses state diagrams in his *dynamic* model to represent this interaction of objects. Since the GOM was built to model OO languages that use sequences of imperative statements in their methods, flow of control in the GOM is modeled using a *procedure-driven* approach versus using *concurrent tasks*. Specifically, flow of control is passed from the sending object to the target object when the message is sent, and it is returned to the sending object when the method completes execution.

Messages are modeled in the GOM by using instances of the **`gom-message`** class. Figure 62 shows the class description for the **`gom-message`** class. The **`gom-call`** attribute

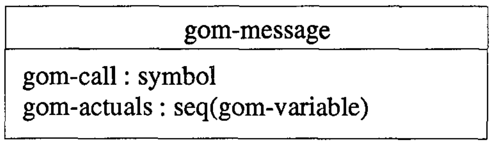


Figure 62 GOM Message Class

stores a symbol representing the name of the method to be executed in the target object. The **`gom-actuals`** attribute stores the sequence of GOM variables that represent the actual parameters of the method. The first parameter in the sequence of actual parameters is always the target object. For example, consider the following assignment statement (shown in GOL syntax).

```
SIGTRB := RHO ( C-7);
```

This assignment statement includes a message that invokes the `RHO` method shown in Figure 60. Figure 63 shows how the message is represented using an AST built from the

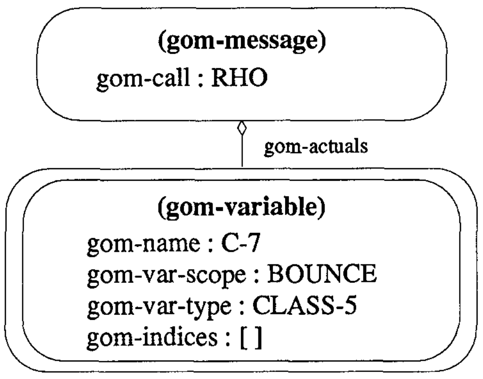


Figure 63 GOM Message Object

gom-message class. The **gom-call** attribute is set to the symbol RHO to indicate the name of the method to be invoked. The **gom-actuals** attribute holds the sequence with the single actual parameter C-7. This parameter is represented as a GOM variable which is named C-7, is in the scope of PRDIV, and is an instance of CLASS-5[2]. Since C-7 is the first parameter of the message, it is the target object.

Messages in the OO paradigm are analogous to subprogram calls in the imperative paradigm. A message that invokes a functional method is similar to a function call. This kind of message is referred to in the rest of this document as a *functional* message. Functional messages are a kind of OO expression, i.e. they do not comprise a statement on their own. A message that invokes a procedural method is similar to a procedure call. This kind of message is referred to in the rest of this document as a *procedural* message. Procedural messages are a kind of OO statement.

The semantics of messages in the GOM are defined using the weakest precondition notation as well as the general form of method declarations defined in Section 4.7. Let m be a procedural method, and let g be a procedural message that invokes m. Let $\bar{a}$ be a vector representing the actual parameters that correspond to input parameters in m. Let $\bar{c}$ be a vector representing the actual parameters that correspond to output parameters in m. The message g takes the form

$$m(\bar{a}, \bar{c})$$

[2]For simplicity, this instance of CLASS-5 is represented in this figure as the symbol CLASS-5

As defined in Section 4.7, the `gom-stmts` attribute of a GOM method declaration models a sequence of GOM statements. Let $S1_m$ be this sequence of statements from the method m. Recall from Section 4.7 that $\bar{x}$ is a vector representing the input parameters of m, and $\bar{z}$ is a vector representing the output parameters of m. When a message is sent to an object, the formal parameters of the method being requested are set to the values of the actual parameters from the message. The method corresponding to the message sent is then executed. Formally, this is equivalent to executing the following sequence.

$$[\bar{x} := \bar{a}, S1_m, \bar{c} := \bar{z}]$$

Using weakest precondition notation, the semantics for the message g is defined as follows.

Definition IV.1. $wp(m(\bar{a}, \bar{c}), R) \;=\; wp([\bar{x} := \bar{a}, S1_m, \bar{c} := \bar{z}], R)$

The formal semantics for functional messages are defined more restrictively as follows. Let m be a functional method, let $S1$ be the sequence of statements declared in m, and let g be a functional message that invokes m. Let $\bar{a}$ be a vector representing the actual parameters that correspond to input parameters in m. Since every such method appears as part of an expression in some statement, let S represent the statement in which m appears. Recall the modified representation of a method presented in Section 4.7 includes one output parameter, y, which represents the value returned from m. Let m' be the method that represents m with the formal parameters from m and the additional parameter y. Let b represent the actual parameter that corresponds to y.

Using these definitions, the invocation of m takes the following form.

$$m(\bar{a})$$

The invocation of the method m' takes the following form.

$$m'(\bar{a}, y)$$

The difference between the invocations of m and m' is that the invocation of m is part of S and the invocation of m' is a single statement. Invoking m is equivalent to invoking m' and then substituting b in S for any invocations of m. This provides a formal representation of the value returned from m. The substitution of b for the call to m in S is represented by the following notation.

$$S_b^{m(\bar{a})}$$

In this way, the call to m is equivalent to the following sequence of statements.

$$[m'(\bar{a}, b), S_b^{m(\bar{a})}]$$

When the method m' is invoked, the formal parameters of m' are set to $\bar{a}$ and the statements in $S1$ are executed. After execution, the parameter b is set to the value y. Formally, the invocation of m' is equivalent to executing the following sequence.

$$[\bar{x} := \bar{a}, S1, b := y]$$

Using weakest precondition notation, the formal semantics for the message g are defined as follows.

Definition IV.2. $wp(m(\bar{a}),\ R) \ = \ wp([\bar{x} := \bar{a}, S1, b := y, S_b^{m(\bar{a})}], R)$

4.9 "GET-" and "SET-" Messages

In order to provide a better encapsulation of extracted objects, the following restriction applies.

Restriction IV.3. *Object attributes are accessed and assigned values only through "GET-" and "SET-" messages sent to the object.*

Figure 64 shows the `GET-ZCOS` and `SET-ZCOS` methods used for accessing and assigning a value to the `ZCOS` attribute.

The general form of a "SET-" message is

$$\text{SET-}a(\tau, e)$$

```
method GET-ZCOS ( C-10 ): real
  begin
    GET-ZCOS := C-10.ZCOS
  end
method SET-ZCOS ( C-11, VAL-10 )
  begin
    C-11.ZCOS := VAL-10
  end
```

Figure 64 "SET-" and "GET-" Methods for ZCOS

where a is the name of the attribute being assigned a value, τ is the target object, and e is the expression to which the attribute a is set. The formal semantics for a "SET-" message are defined using the weakest precondition notation. Given a postcondition R, let $R_e^{\tau.a}$ represent R with all free occurrences of the attribute a from τ simultaneously replaced with e. The semantics for the general form of the "SET-" message are defined below.

Definition IV.3. $wp(SET\text{-}a(\tau, e),\ R) = R_e^{\tau.a}$

The general form of a "GET-" message is shown below.

$$\text{GET-}a(\tau)$$

Here, a is the name of an attribute being accessed and τ is the target object. The "GET-" message for an attribute returns the value of the attribute a from the target object τ. Since the "GET-" message is returning a value, it is similar to a function invocation from the imperative paradigm. Because of this similarity, the formal semantics for a "GET-" message are defined using a modified version of the "GET-" method. The modified version includes a single output parameter, y, which represents the value returned from the "GET-" method. Let b represent the actual parameter that corresponds to the formal parameter y. The invocation of the modified "GET-" method takes the following form.

$$\text{GET-}a(\tau, b)$$

The execution of this modified "GET-" message is equivalent to executing the following sequence.

$$[y := \tau.a,\ \ b := y]$$

As with function invocation, the value of b replaces the original "GET-" message in the statement that includes it. Let S be the statement that includes the "GET-" message. Let $S_b^{\text{GET-}a(\tau)}$ represent the textual substitution of b for each occurrence of the "GET-a" message in the statement S. The formal semantics of a "GET-" message are defined as follows.

Definition IV.4. $wp(GET\text{-}a(\tau),\ R)\ =\ wp([y := \tau.a,\ b := y,\ S_b^{GET\text{-}a(\tau)}],\ R)$

The "GET-" and "SET-" methods for each attribute of each class in the object-oriented design must be provided before the design can be implemented. This research provides a transformation, κ, in Section 6.11.2 which formalizes the automatic generation of these methods and their addition to the set of operations for each class in the design.

4.10 Inheritance

In the GOM, the collection of classes in an object-oriented design is modeled as a class hierarchy where certain classes *inherit* the attributes and operations from one or more other classes. This concept of *inheritance* is represented in the GOM by the `gom-super` attribute of ASTs built from the `gom-class` class. The `gom-super` attribute indicates from which class a specific class inherits.

```
class CLASS-5 attributes RHOSTD, LAMBDA, ANGFAC
  method RHO ( C-5 ): real
    begin
      RHO := GET-RHOSTD ( C-5)
    end
  superclass USER-OBJECT
```

Figure 65   CLASS-5 inherits from USER-OBJECT

For example, Figure 65 shows the class `CLASS-5` (using GOL syntax). `CLASS-5` inherits from the class named `USER-OBJECT`. To represent inheritance, the `gom-super` attribute holds the symbol that represents the name of the class from which this class inherits.

4.11 Polymorphism

There is support for modeling polymorphism in the GOM, i.e. declaring methods with the same name in different classes is allowed. This support is rather rudimentary because the concept of an ***abstract*** class is not supported by the GOM. By including abstract classes in the GOM, it would be possible to model ***class-wide*** operations as implemented by the Ada 95 language.

4.12 Object-Oriented Assignment

Several imperative programming language constructs appear in the GOM including the ***assignment*** construct. As discussed in Section 3.3, assignment takes the general form of $x := e$, where x is a variable and e is an expression. The object-oriented version of the assignment construct now allows x to be an attribute access, which models the way attributes of objects are assigned values. The expression e is also allowed to be an attribute access or a message to another object. OO assignment statements are modeled in the GOM using instances of the `gom-assignment` class. Figure 66 shows the class description for the

gom-assignment
gom-assign-lhs : gom-variable gom-assign-rhs : gom-data-construct

Figure 66 GOM Assignment Class

`gom-assignment` class. The `gom-assign-lhs` attribute models the variable or attribute being assigned a value, i.e. x. The `gom-assign-rhs` attribute models the expression e.

For example, consider the following assignment statement (shown using GOL surface syntax).

```
SIGTRB := RHO ( C-7)
```

This assignment statement assigns the value returned from the RHO message. Figure 67

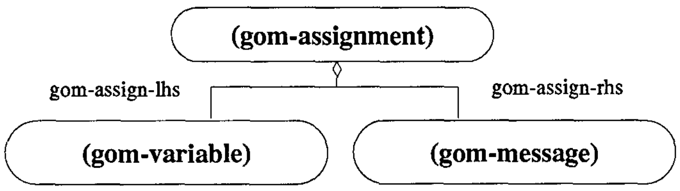


Figure 67 GOM Assignment Object

shows the AST built to model this assignment statement. The `gom-assign-lhs` attribute models the GOM variable `SIGTRB`, which is in the scope of `PRDIV`, and is of type `gom-real`. The `gom-assign-rhs` attribute models the `RHO` message which returns a value of type `gom-real` (see Figure 63).

The formal semantics for an OO assignment statement are defined using the weakest precondition notation. Given a postcondition R, let R_e^x represent R with all free occurrences of x simultaneously replaced with e [19]. The semantics for the general form of the GOM assignment, $x = e$ are defined below.

Definition IV.5. $wp(x := e,\ R) = R_e^x$

To relate these semantics to the GOM, note that the `gom-assign-lhs` attribute models x and the `gom-assign-rhs` models e.

4.13 Object-Oriented Sequential Control

As in the imperative paradigm, the default control mechanism for executing statements in the OO paradigm is *sequential* control. When methods are being executed, the statements are executed one after another in a sequential manner. As in the GIM, sequential control flow is modeled in the GOM by storing the statements to be executed in a sequence. The order of the sequence indicates the order in which the statements are to be executed.

For example, in the collection of statements shown below, `<OO Statement 1>` is executed first followed by `<OO Statement 2>` followed by `<OO Statement 3>`.

```
<OO Statement 1> <OO Statement 2> <OO Statement 3>
```

This collection of statements is modeled in the GOM using the following sequence.

```
[ <OO Statement 1>, <OO Statement 2>, <OO Statement 3> ]
```

The formal semantics for sequential control flow are defined using the weakest precondition notation. Let $S1$ and $S2$ be statements, then the semantics for the sequential composition of these two statements in the GOM is defined below.

Definition IV.6. $wp([S1, S2],\ R) = wp(S1,\ wp(S2, R))$

The formal semantics of sequential control flow in the GOM are identical to the formal semantics of sequential control flow in the GIM. As in the GIM, the representation of sequential control flow in the GOM is based on function composition [25], which is associative, so the GOM representation of sequential control flow is associative.

Theorem IV.1. $wp([S1, [S2, S3]],\ R) = wp([[S1, S2], S3],\ R)$

Proof.

$$\begin{aligned} wp([S1, [S2, S3]],\ R) &= wp(S1, wp([S2, S3],\ R)) \\ &= wp(S1, wp(S2, wp(S3,\ R))) \\ &= wp([S1, S2], wp(S3,\ R)) \text{ (by function composition)} \\ &= wp([[S1, S2], S3],\ R) \end{aligned}$$

□

Since the weakest precondition for the sequences $[S1, [S2, S3]]$ and $[[S1, S2], S3]$ are equal, the sequence $[S1, S2, S3]$ is often used in this document to represent these sequences. In addition, singleton sequences and empty sequences are also used when convenient.

4.14 Object-Oriented Selective Control

As defined in the imperative paradigm (see Section 3.5), *selective* control flow consists of a selection between two or more statements. Since the statements to be executed represent one or more sequentially composed statements, they are modeled as sequences of statements in the GOM. Selective control flow constructs are modeled in the GOM

gom-selection
gom-exp : gom-data-construct gom-then-part : seq(gom-program-construct) gom-else-part : seq(gom-program-construct)

Figure 68 GOM Selection Class

using ASTs built from the **gom-selection** class. Figure 68 shows the class description for the **gom-selection** class. The **gom-exp** attribute models the OO expression that controls which statements will be executed by the selective programming construct. The **gom-then-part** models the sequence of OO statements that are executed when the OO expression in **gom-exp** is true. The **gom-else-part** models the sequence of statements that will be executed if the **gom-exp** is false. As in the GIM, $S1$ must contain at least one statement, but $S2$ may be empty. ASTs where $S1$ and $S2$ have at least one statement model object-oriented **if-then-else** statements. ASTs where $S2$ is empty model object-oriented **if-then** statements.

For example, the **if-then-else** statement shown below (using GOL syntax) provides a good example of selective control flow in the GOM.

```
if GET-SIGTRB ( C-12) > 0.0 then
  UPLREQ := SIGUP / GET-SIGTRB ( C-12)
else
  UPLREQ := 1000000.0d0
endif
```

This statement makes a selection based on the result of the GET-SIGTRB message, which is sent to the target object C-12. The statement selects between two assignment statements that assign values to the variable UPLREQ. Figure 69 shows the object instance that mod-

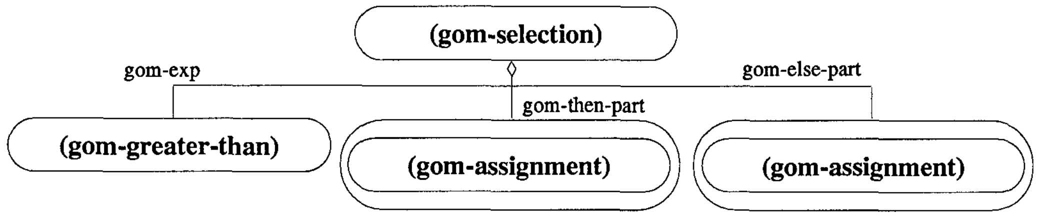


Figure 69 GOM Selective Control Flow Example

els this GOM **if-then-else** statement. The **gom-exp** attribute models the **greater-than** boolean expression that controls the selection and the two assignment statements are modeled by the **gom-then-part** and the **gom-else-part** attributes.

The formal semantics of selective control flow in the GOM are defined using the weakest precondition notation. Let B represent an OO boolean expression and let $S1$ and $S2$ represent sequences of OO statements. The selective control flow constructs in the GOM take the following form

$$\textbf{if } B \textbf{ then } S1 \textbf{ else } S2$$

Recall that $S1$ is executed if B is true and $S2$ is executed if B is false. The semantics of this GOM form of selective control flow are defined below.

Definition IV.7. $wp(\textbf{if } B \textbf{ then } S1 \textbf{ else } S2,\ R)\ =$

$$(B \Rightarrow wp(S1, R))\ \wedge\ (\neg B \Rightarrow wp(S2, R))$$

To relate these formal semantics to the GOM, note that B is modeled using the **gom-exp** attribute. The sequence of statements $S1$ is modeled by the **gom-then-part** attribute and the sequence $S2$ is modeled by the **gom-else-part** attribute. The formal semantics for GOM selection is now defined.

As in the imperative paradigm, the formal semantics for selective control flow when the sequence $S2$ is empty are defined using the **skip** command. Recall from Section 3.5 that the formal semantics of the **skip** command are $wp(\textbf{skip},\ R)\ =\ R$.

Selective control flow in the GOM when $S2$ is empty takes the form

$$\textbf{if } B \textbf{ then } S1 \textbf{ else skip}$$

The formal semantics for this form of selective control flow in the GOM is defined below.

Definition IV.8. $wp(\textbf{if } B \textbf{ then } S1 \textbf{ else skip},\ R)\ =\ (B \Rightarrow wp(S1,\ R))\ \wedge\ (\neg B \Rightarrow R)$

4.15 Object-Oriented Iterative Control

Iterative control flow, as defined in the imperative paradigm (see Section 3.6) is a mechanism for repeating sequences of statements. The execution of a sequence of statements continues while some boolean expression is true. Iterative control flow in the GOM is modeled using ASTs built from the `gom-iteration` class. Figure 70 shows the class de-

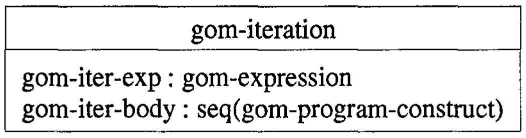

gom-iteration
gom-iter-exp : gom-expression
gom-iter-body : seq(gom-program-construct)

Figure 70 GOM Iteration Class

scription for the `gom-iteration` class. The `gom-iter-expr` attribute models the boolean expression that controls the iteration. The `gom-iter-body` holds the sequence of statements that are executed each time through the iteration.

For example, the `while` loop shown below (using GOL syntax).

```
IL := 1;
while IL <= GET-NLASER ( C-14) do
  begin
  LLAS ( IL) := 0;
  TLAS ( IL) := 0;
  IL := IL + 1
  end
```

This `while` loop is an example of iterative control flow in the object paradigm. Figure 71

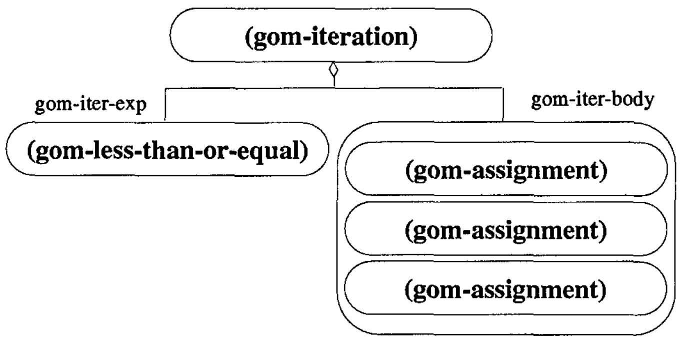


Figure 71 GOM Iteration Object

shows the AST that models this `while` loop in the GOM. The `gom-ter-exp` attribute holds the less-than-or-equal expression that controls the iteration. The `gom-iter-body` holds the three assignment statements in the body of the loop.

The GOM model of iteration provides a canonical form for representing imperative-style iteration in the object paradigm. Several OO programming languages provide programming constructs that implement iteration including the `while` statement in C++ and the `loop` statement in Ada 95. The feasibility demonstration (see Chapter VIII) provides rudimentary examples of converting certain GOM entities into Ada 95 code.

As with the other control flow constructs, the formal semantics for iterative control flow are defined using the weakest precondition notation. Given a precondition P and a postcondition R, let $H_k(R)$ represent the set of all states in which

$$\text{while } B \text{ do } S1$$

will establish R in at most k iterations [19]. The formal semantics for iterative control flow in the GOM are defined below.

Definition IV.9. $wp(\textit{ while } B \textit{ do } S1,\ R) = (\exists k : 0 \leq k \ \wedge \ H_k(R))$

To tie this general definition to the specific model of iterative control flow in the GOM, recall that B is modeled by the **`gom-iter-exp`** attribute and $S1$ is modeled by the **`gom-iter-body`** attribute of the **`gom-iteration`** class.

As in the GIM, the task still remains to define $H_k(R)$. There is no attempt in this research to define *loop invariants* [19] for iteration in the GOM. The set of states $H_k(R)$ defines the semantics of iteration in the GOM to a level of detail sufficient for this research.

4.16 Object-Oriented Subprogram Invocation

Even though the primary means of communication in the OO paradigm consists of objects sending messages to each other, there is still a need to model subprogram invocations in the GOM. Specifically, there will still be calls to utility subprograms such as `SIN` and `COS` that must be modeled in the GOM. Since this research involves re-engineering GIM subprograms to the GOM, all user-defined subprograms must be converted to methods and all calls to user-defined subprograms must be converted to messages. This leads to the following restriction of the GOM.

Restriction IV.4. *Only calls to non-user-defined subprograms are modeled by subprogram invocations in the GOM.*

As in the imperative paradigm, there are two types of subprogram invocation, viz. *procedure* calls and *function* calls. If the non-user-defined subprogram being called is a *procedure*, then the **gom-procedure-call** class is used to model the procedure call. Figure 72 shows the class definition for the **gom-procedure-call** class. The attribute **gom-proc-call-identifier** holds a symbol that represents the name of the procedure being called. The **gom-proc-call-actuals** attribute holds the sequence of actual parameters sent to the procedure.

gom-procedure-call
gom-proc-call-identifier : symbol
gom-proc-call-actuals : seq(gom-data-construct)

Figure 72 GOM Procedure Call Class

If the non-user-defined subprogram being called is a *function*, then the **gom-function-call** class is used to model the function call. Figure 73 shows the class description for the **gom-function-call** class. The attribute **gom-fun-call-identifier** models the name of the function being called. The **gom-fun-call-actuals** attribute models the actual parameters that are passed to the called function.

gom-function-call
gom-fun-call-identifier : symbol
gom-fun-call-actuals : seq(gom-data-construct)

Figure 73 GOM Function Call Class

The formal semantics for calls to non-user-defined procedures in the GOM are similar to the semantics for procedure calls in the GIM (see Section 3.11). As in the GIM, procedures modeled in the GOM are not allowed to have parameters that are both input and output parameters, thus calls to non-user-defined procedures must have no such parameters.

Let p be a non-user-defined procedure and let $S1$ be the sequence of statements declared in p. Let $\bar{a}$ be a vector representing the actual parameters that correspond to input parameters in p. Let $\bar{c}$ be a vector representing the actual parameters that correspond to output parameters in p. An invocation of p takes the form:

$$p(\bar{a}, \bar{c})$$

To invoke p, the actual parameters are copied to the formal parameters and control flow transfers to p. Assume the vectors $\bar{x}$ and $\bar{z}$ are the vectors representing the input and output parameters of the non-user-defined procedure, respectively. Executing the procedure call p is equivalent to executing the following sequence

$$[\bar{x} := \bar{a}, S1, \bar{c} := \bar{z}]$$

Using weakest precondition notation, the semantics for a call to a non-user-defined procedure are defined below.

Definition IV.10. $wp(p(\bar{a}, \bar{c}), R) \;=\; wp([\bar{x} := \bar{a}, S1, \bar{c} := \bar{z}],\ R)$

The formal semantics for calls to non-user-defined functions in the GOM are similar to the semantics for function calls in the GIM (see Section 3.11). As in the GIM, a restriction is placed on functions as defined below.

Restriction IV.5. *Non-user-defined functions must have no output parameters.*

Let f be a non-user-defined function and let $S1$ be the sequence of statements declared in f. Let $\bar{a}$ be a vector representing the actual parameters that correspond to the input parameters in f. Since every function appears as part of an expression in some statement, let S represent the statement in which f appears. Recall the modified representation of a function presented in Section 3.10 includes one output parameter, y, which represents the value returned from the function f. Let f' be the procedure that represents f with the formal parameters from f and the additional parameter y. Let b represent the actual parameter that corresponds to y. Using these definitions, an invocation of the function f takes the following form.

$$f(\bar{a})$$

An invocation of the procedure, f', takes the following form.

$$f'(\bar{a}, b)$$

The difference between the invocations of f and f' is that the invocation of f is part of S and the invocation of f' is a single statement. Invoking f is equivalent to invoking f' and then substituting b in S for any invocations of f. This provides a formal representation of the value returned from f. The substitution of b for the call to f in S is represented by the following notation.

$$S_b^{f(\bar{a})}$$

In this way, the call to f is equivalent to the following sequence of statements.

$$[f'(\bar{a}, b), S_b^{f(\bar{a})}]$$

As defined in Section 3.11, when the procedure f' is invoked, the actual parameters in $\bar{a}$ are copied to the formal parameters in $\bar{x}$ and control flow transfers to f'. After execution of the function, the value of y is copied to b. Hence, executing the call to f' is equivalent to executing the following sequence.

$$[\bar{x} := \bar{a}, S1, b := y]$$

Using weakest precondition notation, the formal semantics of a call to the non-user-defined function f are defined below.

Definition IV.11. $wp(f(\bar{a}),\ R)\ =\ wp([\bar{x} := \bar{a}, S1, b := y, S_b^{f(\bar{a})}],\ R)$

4.17 Recursion

In the object paradigm, it is possible for a method to send a message that invokes itself, which is termed *recursion.* The GOM does not model recursion, which leads to the following restriction.

Restriction IV.6. *Methods modeled in the GOM are not allowed to send messages invoking the method recursively.*

Even more restrictively, the *call tree* consisting of the graph of messages sent between objects is assumed to be a *directed acyclic graph.*

Restriction IV.7. *The* call tree *of messages sent between objects must be a* directed acyclic graph.

4.18 Variables

As in the imperative paradigm, *variables* are used to store pieces of information in the object-oriented paradigm. Variables appear in OO programs as *local* variables or formal parameters of methods. Local variables are declared in the local scope of a method and the value of the variable is lost when execution of the method ends. Formal parameters of methods are variables that are declared in the method and used to pass data items into the method. This section defines the AST class used to model variables in the GOM and presents an example of modeling a variable.

Both local variables and formal parameters are modeled in the GOM by using the `gom-variable` class. Figure 74 shows the class description for the `gom-variable` class.

gom-variable
gom-name : symbol gom-var-scope : symbol gom-var-type : gom-data-type gom-indices : seq(gom-data-construct)

Figure 74 GOM Variable Class

The `gom-name` attribute holds a symbol that represents the name of the variable. The `gom-var-scope` holds a symbol that represents the scope of this variable. See Section 4.21 for a discussion of scoping issues in the GOM. The `gom-var-type` attribute models the data type of the variable. Section 4.20 describes and lists the data types modeled in the GOM. If a variable is an array with indices, the `gom-indices` attribute is used to model the access into the array.

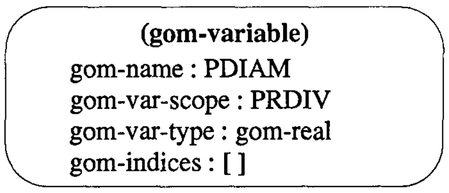

(gom-variable)
gom-name : PDIAM
gom-var-scope : PRDIV
gom-var-type : gom-real
gom-indices : [ ]

Figure 75 GOM Variable Object Example

For example, Figure 75 shows how the local variable `PDIAM`, defined in a method `PRDIV`, is represented in the GOM using an instance of the `gom-variable` class. The `gom-var-scope` attribute is set to the symbol `PRDIV` to indicate this variable is in the scope of the method `PRDIV`. The data type of `PRDIV` is modeled using the `gom-real` object. The `gom-indices` attribute holds an empty sequence to indicate this variable has no indices.

4.19 Attributes

In the object paradigm, data items are also stored as *attributes* of objects. As described in Section 4.4, classes define a template for objects by declaring attributes that are built into each instance of the class. This forms a collection of variables associated with the object that are accessed as attributes of the object. This section defines the AST class that models object attributes in the GOM and presents an example of how an attribute is modeled.

Attributes are modeled in the GOM using the `gom-attribute` class. Figure 76 shows

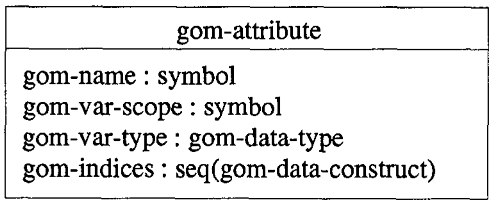

gom-attribute
gom-name : symbol
gom-var-scope : symbol
gom-var-type : gom-data-type
gom-indices : seq(gom-data-construct)

Figure 76 GOM Attribute Class

the class definition for the `gom-attribute` class. The `gom-name` attribute holds a symbol that represents the name of the attribute. The `gom-var-scope` attribute holds a symbol that represents the scope of this attribute. The `gom-var-type` attribute models the data type of the attribute. If an attribute is an array, the `gom-indices` attribute is used to model the indices used when accessing the array. Note that the classes that model variables and attributes in the GOM are identical. This is because of the similarity between attributes and variables in the OO paradigm.

```
class CLASS-2 attributes ZCOS
  method GET-ZCOS ( C-10 ): real
    begin
      GET-ZCOS := C-10.ZCOS
    end
  method SET-ZCOS ( C-11, VAL-10 )
    begin
      C-11.ZCOS := VAL-10
    end
superclass USER-OBJECT
```

Figure 77 Example of ZCOS Attribute

For example, consider the class shown in Figure 77. The **CLASS-2** class has one attribute, **ZCOS**. The "SET-" and "GET-" methods are also shown in the figure. As described in Section 4.4, the **gom-attrs** attribute of the class **gom-class** models the attributes of a class. These attributes are modeled using instances of the **gom-attribute** class. For

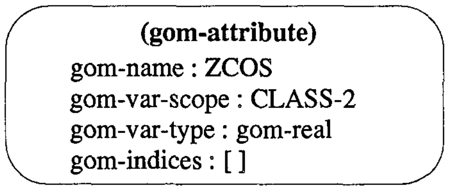


Figure 78 GOM Attribute Object Example

example, Figure 78 shows the **gom-attribute** object that models the **ZCOS** attribute of the **CLASS-2** class. The **gom-name** attribute holds the symbol **ZCOS**. The **gom-var-scope** attribute holds the symbol **CLASS-2** to represent the fact that **ZCOS** is declared in the scope of **CLASS-2**. The **gom-var-type** attribute models the data type of the attribute, which in this example is *real*.

4.20 *Data types*

This section describes the data types that are used when modeling variables in the GOM. The data types being modeled are as follows.

Instance object instances of a class

Integer zero and positive and negative whole numbers

Real rationals and irrationals

Boolean true and false logical values

Character single alpha characters

String sequences of characters

Array homogeneous collections of elements accessed with an index.

The **Instance** data type is used as the data type of variables in the GOM that are instances of a class. Each of the data types presented here is modeled using a separate class in the GOM. In order to model a specific data type in the GOM, the data type attribute of GOM variables and GOM attributes holds an instance of an AST built from the class that models the data type of the variable or attribute. Examples of this are shown in Figure 75 and Figure 78.

As discussed in Section 3.15, the number of bytes associated with an imperative data type can be specified by the user or derived from pre-defined data types. This is true in the object paradigm as well, so each class that models an object-oriented data type has an attribute `gom-type-size` that models the number of bytes associated with the data type.

4.21 *Scoping Issues*

As in the imperative paradigm, each variable in the object-oriented paradigm has a *scope* associated with it that refers to its "visibility". A method that defines a variable has visibility to the variable. A variable that is an instance of a class provides visibility to the attributes of the instance. Attributes in the object-oriented paradigm also have scope. This section describes how the scope of variables and attributes in the GOM is modeled.

In the GOM, each method has a *local* scope much as subprograms in the GIM have a local scope. If a variable is declared in a method, this local variable is only visible to the declaring method. The formal parameters of a method are also only visible to the declaring method. In order for statements in a method to access the attributes of an instance, the instance must be a local variable or a formal parameter of the method.

Some object-oriented programming languages allow variables declared outside the local scope to be accessed by a method. This type of access is not allowed in the GOM, which leads to the following restriction.

Restriction IV.8. *All variables in a GOM method are either declared locally or are formal parameters of the method.*

In the object paradigm, a method passes information to other methods by including variables as parameters of messages. As in the imperative paradigm, parameters are either input or output, i.e. the value of the parameter is either read in the method or written to in the method. If one method passes a variable to another method as an output parameter, the method being called changes the value of the variable in the calling method. As in the imperative paradigm, this is termed *pass by reference* parameter passing [16]. All parameters are passed by reference in the GOM.

In some object-oriented programming languages, one method can be declared inside another method. This means the method would be visible only to the method in which it was declared. Limiting the visibility of a method in this way is not allowed in the GOM, which leads to the following restriction.

Restriction IV.9. *Methods cannot be declared inside of another method.*

Each of the methods declared in the GOM must be part of a class, i.e. in the set of operations for a class. This means a particular class has visibility to each of the methods declared within it. When a message is sent to a target object, the class of the object is used to determine with method to invoke. This mechanism is represented and canonicalized in the GOM by requiring each class to be declared in one overall *global* scope. This leads to the following restriction on the GOM.

Restriction IV.10. *Classes cannot be declared inside of another class.*

4.22 Homogeneous Data Structures

As in the imperative paradigm, collections of homogeneous elements can be built in the object paradigm using constructs such as *lists* and *arrays*. The only such collection modeled in the GOM is the array. Arrays are represented in the GOM by building variables and attributes of the data type gom-array (see Section 4.20).

For example, the following statement includes an access to the array variable GEOM.

```
RANGE := RANGE + GEOM ( K)
```

The GOM AST shown in Figure 79 models this access to the Kth element of the GEOM array variable.

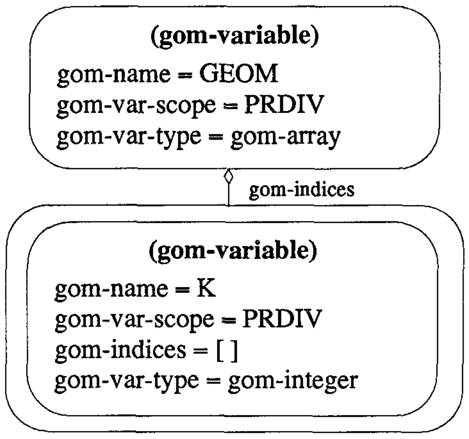


Figure 79 GEOM Array Variable Access Object

If an attribute of an object is an array, then the indices used to access the array are passed as parameters to the "GET-" or "SET-" message that accesses the attribute. For example, the following statement accesses the Kth element of the GEOM attribute of the instance variable C-10.

```
RANGE := RANGE + GET-GEOM ( C-10, K)
```

The GEOM attribute is an array and the GET-GEOM message is used to access the elements of the array. In the statement, the message GET-GEOM (C-10, K) is sent to the target object C-10 requesting element K from the GEOM array attribute. The GOM AST shown in Figure 80 models the message used to access the Kth element of the GEOM array attribute.

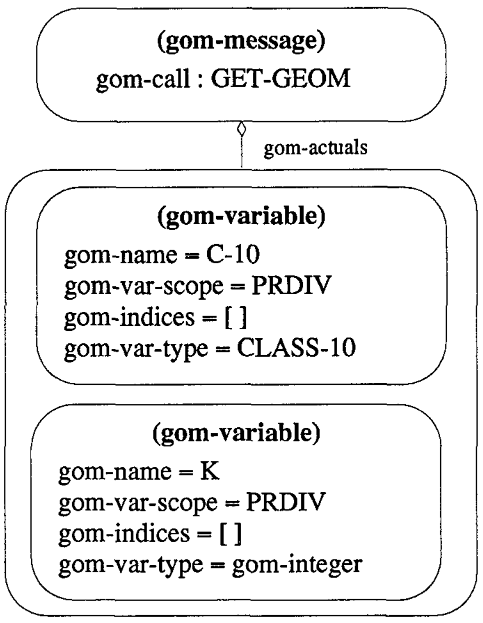


Figure 80 GEOM Attribute Array Access Object

The first parameter of the message is the target object `C-10` and the second parameter is the variable `K`. This variable holds the index of the element to be retrieved from the array.

To set the value of an array attribute, the appropriate "SET-" message is used. For example, the following statement sets the `K`th element of the `GEOM` attribute to the value of the `RANGE` variable.

```
SET-GEOM ( C-10, K, RANGE)
```

In this statement, the message `SET-GEOM ( C-10, K, RANGE)` is sent to the target object `C-10` in order to set the `K`th element to the value of the `RANGE` variable. The GOM AST shown in Figure 81 models the "SET-" message. The first parameter is the target object `C-10`, the second parameter is the index variable `K`, and the final parameter is the variable to which the array element will be set, i.e. the `RANGE` variable.

Figure 82 shows the `GET-GEOM` and `SET-GEOM` methods used for accessing and assigning a value to the `GEOM` array attribute.

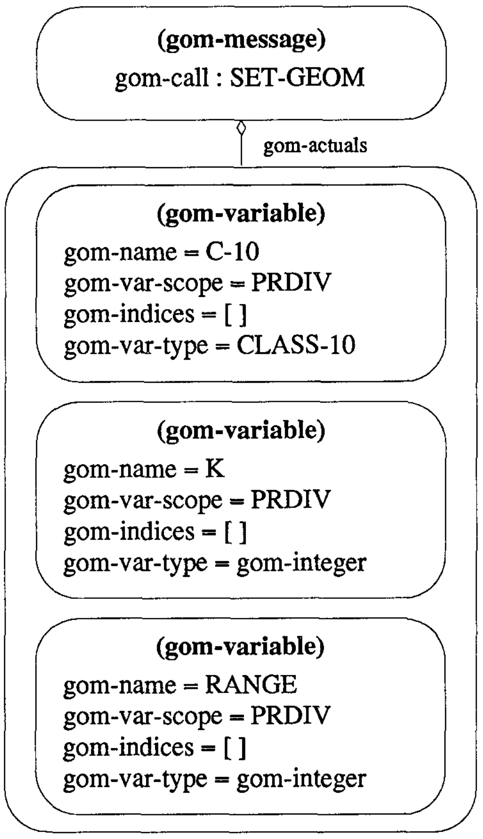


Figure 81 GEOM Attribute Array Access Object

4.23 Heterogeneous Data Structures

Some object-oriented programming languages allow the programmer to build collections of heterogeneous data items, for example, *records* in Ada 95 and *structs* in C++. These heterogeneous data structures are not modeled in the GOM.

Restriction IV.11. *The GOM does not model heterogeneous data structures.*

4.24 Pointers

As in the imperative paradigm, some object-oriented programming languages allow the programmer to store the address of a variable and then access the variable via the address. This mechanism is referred to as using *pointers* to variables. Pointers are not modeled in the GOM.

Restriction IV.12. *The GOM does not model pointers.*

```
method GET-GEOM ( C-10, IDX-1 ): real
  begin
    GET-GEOM := C-10.GEOM ( IDX-1)
  end
method SET-GEOM ( C-11, IDX-2, VAL-10 )
  begin
    C-11.GEOM ( IDX-2) := VAL-10
  end
```

Figure 82 "SET-" and "GET-" Methods for GEOM

4.25 Object-Oriented Expressions

In the object-oriented paradigm, expressions can be literal values, unary or binary expressions with operators, non-user-defined function calls, functional messages, or variables. This section describes how expressions are modeled in the GOM.

Literal values in the GOM are modeled by the **`gom-literal-constant`** class. The

gom-literal-constant
gom-literal-value : any-type

Figure 83 Modeling Object-Oriented Literal Constants

class description for the **`gom-literal-constant`** class is shown in Figure 83. The **`gom-literal-value`** attribute models the value of the literal. Figure 84 shows the literal con-

gom-literal-integer
gom-literal-real
gom-literal-boolean
gom-literal-string

Figure 84 Object-Oriented Literal Constants

stants currently modeled in the GOM. Each literal shown is modeled as an AST defined as a subclass of the **`gom-literal-constant`** class. As in the GIM, literal constants other than those listed in Figure 84 are not modeled in the GOM, but can be modeled by defining a new subclass under the **`gom-literal-constant`** class that describes the new literal constant.

Unary expressions in the GOM are modeled using ASTs built from the `gom-unary-expression` class. Figure 85 shows the class description for the `gom-unary-expression`

gom-unary-expression
gom-unary-operand : gom-expression

Figure 85 Modeling Object-Oriented Unary Expressions

class. The `gom-unary-operand` attribute models the single operand of the unary expression. Figure 86 shows all the unary expressions modeled in the GOM. Each unary ex-

gom-negate
gom-not

Figure 86 List of Object-Oriented Unary Expressions

pression shown is modeled as an AST defined as a subclass of the `gom-unary-expression` class. As with literal constants, any unary expressions other than those listed in Figure 86 are not modeled in the GOM but can be built as a new subclass that defines the new unary expression.

Binary expressions in the GOM are modeled by the `gom-binary-expression` class. Figure 87 shows the class description for `gom-binary-expression` class. As in the impera-

gom-binary-expression
gom-bin-exp-operand-1 : gom-expression gom-bin-exp-operand-2 : gom-expression gom-bin-exp-seq : seq(gom-expression)

Figure 87 GOM Binary Expressions Class

tive paradigm, binary expressions in the object-oriented paradigm have either two operands or multiple operands. For this reason, the `gom-binary-expression` class has three attributes for modeling the operands of the expression. The `gom-bin-exp-operand-1` and `gom-bin-exp-operand-2` attributes hold the two operands if the binary expression has only two operands. When the same binary expression is repeated for multiple operands, such as in the expression `ZCOS * ZCOS * ZCOS`, the multiple operands are modeled using the `gom-bin-exp-seq` attribute. Certain binary expressions can be repeated in this way, e.g. addition and multiplication. Others cannot be repeated in this way, e.g. <, <=, and >.

gom-division	
gom-equal	
gom-exponent	
gom-greater-than-or-equal	
gom-greater-than	gom-addition
gom-less-than-or-equal	gom-and
gom-less-than	gom-concat
gom-not-equal	gom-multiplication
gom-subtraction	gom-or
(a) Two operand.	(b) Sequence.

Figure 88 Object-Oriented Binary Expressions Modeled in the GOM

Figure 88 lists all the binary expressions that are modeled in the GOM. The binary expressions that cannot be repeated are shown in Figure 4.88(a). These expressions are modeled using the **`gom-bin-exp-operand-1`** and **`gom-bin-exp-operand-2`** attributes. The binary expressions that can be repeated are shown in Figure 4.88(b). These expressions are modeled using the **`gom-bin-exp-seq`** attribute.

As with literal constants and unary expressions, any binary expressions not shown in Figure 88 are not modeled in the GOM. New binary expressions are modeled by adding a subclass to the **`gom-binary-expression`** class to define the new binary expression.

4.26 Object-Oriented Input

This section defines the ASTs that are used to model input statement from object-oriented programming languages. Most object-oriented programming languages allow the programmer to input data items from a file or the console. The GOM models this aspect of object-oriented programming languages using ASTs as defined below.

Input statements are modeled in the GOM by using ASTs built from the **`gom-input`** class. Figure 89 shows the class description for the **`gom-input`** class. The **`gom-in-logical-file`** attribute is used to model the source of the input. In the GOM, input coming from

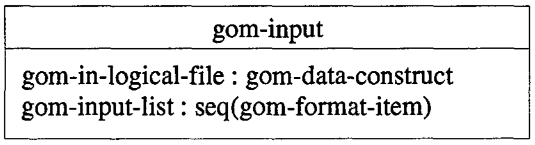

gom-input
gom-in-logical-file : gom-data-construct
gom-input-list : seq(gom-format-item)

Figure 89 GOM Input Class

the user is modeled by setting this attribute to the symbol `CONSOLE`. The **`gom-input-list`** attribute models the sequence of items being input.

For example, the following input statement (shown in GOL syntax)

```
read ( CONSOLE, NLASER)
```

inputs the value of the variable `NLASER` from the console, which is indicated by the symbol `CONSOLE` as the first parameter of the statement. Figure 90 shows how this input statement

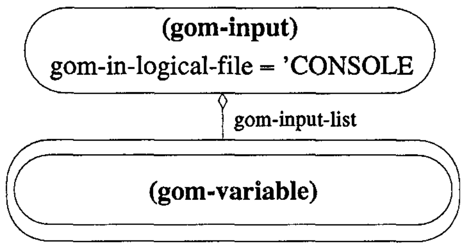


Figure 90 GOM Input Object

is modeled using an AST in the GOM. The **`gom-in-logical-file`** attribute holds the symbol `CONSOLE` to indicate the input is coming from the user. The **`gom-input-list`** attribute is a singleton list holding the `NLASER` variable.

As in the imperative paradigm, object-oriented programming languages also allow the programmer to input data from a file. Before inputting data from the file, the file must be "opened" to establish the logical link to the physical file on disk. These *files* are modeled in the GOM using ASTs built from the **`gom-file`** class. Figure 91 shows the class description for the **`gom-file`** class. The **`gom-designator`** attribute holds the name of the logical file as referenced in the program. The **`gom-file-name`** attribute holds the name of the physical file. The **`gom-access`** attribute holds the access type for the file, e.g. direct or sequential. The **`gom-status`** attribute holds the status of the file, opened either for input or output. ASTs built to model files are stored as attributes of a design in the GOM.

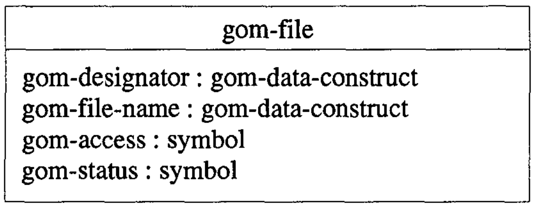

gom-file
gom-designator : gom-data-construct
gom-file-name : gom-data-construct
gom-access : symbol
gom-status : symbol

Figure 91 GOM File Class

The semantics of the GOM input statement are similar to the semantics of the GOM assignment statement. Both statements define values for variables. In general, an input statement in the GOM is equivalent to a subprogram call where all the parameters are output from the subprogram.

For example, let I represent an input statement and let $\bar{a}$ represent a vector of variables in the **gom-input-list** attribute of I (as modeled in the GOM). Let $\bar{x}$ represent a vector of variables returned from the execution of the input statement after the values are read from the indicated file (or the console). Execution of the GOM input statement is equivalent to the following sequence:

$$[I(\bar{a}),\ \bar{a} := \bar{x}]$$

The formal semantics for the GOM input statement $I(\bar{a})$ are defined as follows.

Definition IV.12. $wp(I(\bar{a}),\ R) \ = \ wp([I(\bar{a}),\ \bar{a} := \bar{x}],\ R)$

This indicates each of the variables in the vector $\bar{a}$ is set to the values returned by the input statement $I(\bar{a})$.

4.27 Object-Oriented Output

This section defines the ASTs that are used to model output statements from object-oriented programming languages. Most object-oriented programming languages allow the programmer to output data items to a file or to the console. The GOM models this aspect of object-oriented languages as defined below.

Output statements in the GOM using ASTs built from the **gom-output** class. The class description for the **gom-output** class is shown in Figure 92. The **gom-out-logical-**

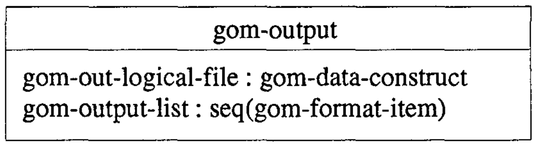

gom-output
gom-out-logical-file : gom-data-construct
gom-output-list : seq(gom-format-item)

Figure 92 GOM Output Class

`file` attribute represents whether the output is being sent to the console or to an output file. In the GOM, the symbol `STD-OUT` is stored in this attribute if the output is going to the console. The `gom-output-list` models the sequence of data items that are being output.

For example, the following output statement (shown in GOL syntax)

```
write ( STD-OUT, " NLASER =", NLASER);
```

outputs two data items to the console. The string literal `" NLASER ="` is output followed by the value of the variable `NLASER`. Figure 93 shows the AST used to model this output

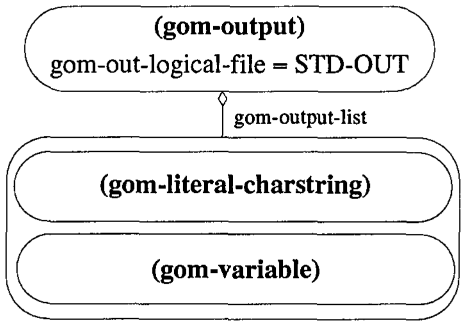


Figure 93 GOM Output Object

statement in the GOM. The `gom-out-logical-file` attribute holds the symbol `STD-OUT` to indicate the output is being sent to the standard output port. The `gom-output-list` attribute holds the sequence of two data items representing the string literal `"NLASER ="` and the variable `NLASER`.

In the GOM, an output statement is not allowed to change the state of the program. Let O represent a GOM output statement and let $\bar{a}$ represent a vector of data items appearing in the output statement O. The semantics of GOM output statements is defined as follows.

Definition IV.13. $wp(O(\bar{a}),\ R)\ =\ R$

This means the GOM output statements are restricted from changing the state of the program.

The following sections provide formal definitions for object-oriented classes, methods, messages, and designs. Formal transformations of an object-oriented design are provided at the end of this section.

4.28 Formalizing GOM Statements and Expressions

This section presents formal representations of GOM statements and expressions. Let x represent a GOM variable and let e represent a GOM expression. A GOM assignment statement is represented formally by the following tuple.

$$< x,\ :=, e >$$

This tuple indicates that x is the variable to be assigned the value of the expression e. The $:=$ symbol is a syntactic token that distinguishes this tuple as an assignment statement.

Let e represent a GOM expression and let S_1 and S_2 represent sequences of GOM statements. A GOM selection statement is represented formally by the following tuple.

$$< \text{if}, e,\ \text{then}, S_1,\ \text{else}, S_2 >$$

This tuple indicates that the sequence of statements S_1 will be executed if the expression e evaluates to true otherwise the S_2 sequence will be executed. The `if`, `then`, and `else` symbols are syntactic tokens used to identify this tuple.

Similarly, let e represent a GOM expression and let S_3 represent a sequence of GOM statements. The GOM iteration statement is represented by the following tuple

$$< \text{while}, e, S_3 >$$

This indicates that the sequence of statements S_3 will be executed while the expression e is true. The `while` symbol is a syntactic token used to identify this tuple.

Let E represent a sequence of expressions. The GOM input and output statements are represented formally by the following tuples.

$$< \text{input},\ \text{iport}, E >$$
$$< \text{output},\ \text{oport}, E >$$

These tuples indicate that the expressions in the sequence E are either input from the input port `iport` or output to the output port `oport`.

Expressions in the GOM are represented formally by the following tuples. Let e_1 and e_2 be GOM expressions.

$$\begin{aligned}
< e_1, +, e_2 >\\
< e_1,\ \text{and}, e_2 >\\
< e_1,\ \text{concat}, e_2 >\\
< e_1,\ /, e_2 >\\
< e_1, =, e_2 >\\
< e_1,\ **, e_2 >\\
< e_1, >=, e_2 >\\
< e_1, >, e_2 >\\
< e_1, <=, e_2 >\\
< e_1, <, e_2 >\\
< e_1, *, e_2 >\\
< e_1, <>, e_2 >\\
< e_1,\ \text{or}, e_2 >\\
< e_1, -, e_2 >\\
< -, e_1 >\\
< \ \text{not}, e_1 >
\end{aligned}$$

These formal representations are used in the formal transformations presented in Chapter VI.

4.29 Formalizing Object-Oriented Classes

This section describes how classes modeled by the `gom-class` AST can be defined more mathematically. The formal definition of an object-oriented class is presented below.

id_C - A symbol storing the name of the class.

Φ - The set of instance attributes.

Ω - The set of instance methods.

λ - The symbol storing the name of the superclass.

Thus, an object-oriented class, C, is defined formally as

$$C \; = < id_C, \Phi_C, \Omega_C, \lambda >$$

```
class CLASS-2 attributes ZCOS, XLAMDA, SIGTRB
  method GET-ZCOS ( C-10 ): real begin GET-ZCOS := C-10.ZCOS
    end
  method SET-ZCOS ( C-11, VAL-10 ) begin C-11.ZCOS := VAL-10
    end
  method GET-XLAMDA ( C-10 ): real begin
    GET-XLAMDA := C-10.XLAMDA end
  method SET-XLAMDA ( C-11, VAL-11 ) begin
    C-11.XLAMDA := VAL-11 end
  method GET-SIGTRB ( C-10 ): real begin
    GET-SIGTRB := C-10.SIGTRB end
  method SET-SIGTRB ( C-11, VAL-12 ) begin
    C-11.SIGTRB := VAL-12 end
  superclass USER-OBJECT
```

Figure 94 Class $CLASS - 2$

For example, Figure 94 shows a class (using GOL syntax) that has three attributes `ZCOS`, `XLAMDA`, and `SIGTRB`. The class inherits from the class `USER-OBJECT` and includes the appropriate methods to set and access its attributes. This class is formally defined by the following tuple.

$$\begin{aligned} C &= < \text{CLASS-2}, \Phi_C, \Omega_C, \text{ USER-OBJECT} > \text{ where} \\ \Phi_C &= \{\text{ZCOS, XLAMDA, SIGTRB}\} \textit{ and} \\ \Omega_C &= \{\text{SET-ZCOS, SET-XLAMDA, SET-SIGTRB,} \\ &\quad \text{GET-ZCOS, GET-XLAMDA, GET-SIGTRB }\} \end{aligned}$$

The name of the class is the symbol `CLASS-2`. The attributes of the class are represented by the set of data items `ZCOS`, `XLAMDA`, and `SIGTRB`. The operations of the class are represented by the set of methods including `SET-ZCOS`, `SET-XLAMDA`, `SET-SIGTRB`, `GET-ZCOS`, `GET-XLAMDA`, `GET-SIGTRB`. Each operation is represented by a formal definition as presented in the following section.

4.30 Formalizing Object-Oriented Methods

This section explains how methods represented using `gom-method` ASTs can be defined more mathematically. Methods are defined as follows.

id_M - A symbol storing the name of the method.
τ - The target object.
Q_{obj} - The sequence of objects passed to the method (other than τ).
Q_{form} - The sequence of formal parameters.
Q_{ret} - The sequence of returned values.
Q_{loc} - The sequence of local variables.
Ψ - The sequence of object-oriented statements.

Thus, an object-oriented method is defined by the tuple:

$$< id_M, \tau, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi >$$

All object-oriented methods are elements of Ω of some object-oriented class.

For example, Figure 95 shows the method `UPLREQ`. This method expects an instance of the class `CLASS-2` to be passed in as the target object. The method returns a value of type `real`. This method is defined formally using the following tuple.

```
class CLASS-2 attributes ZCOS, XLAMDA, SIGTRB
  method UPLREQ ( C-12 ): real
    begin
    SIGTO := 3.75d-6;
    TMP1 :=
      GET-XLAMDA ( C-12) * GET-ZCOS ( C-12) * GET-ZCOS ( C-12)
        * GET-ZCOS ( C-12);
    TMP2 := TMP1^0.2;
    SIGUP := SIGTO / TMP2;
    if GET-SIGTRB ( C-12) > 0.0 then
      UPLREQ := SIGUP / GET-SIGTRB ( C-12)
    else
      UPLREQ := 1000000.0d0
    endif
    end
  superclass USER-OBJECT
```

Figure 95 Method *UPLREQ*

$$< \text{UPLREQ}, \tau, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi > \text{ where}$$
$$\begin{aligned} \tau &= \text{C-12 } \textit{and} \\ Q_{obj} &= [\,] \textit{ and} \\ Q_{form} &= [\,] \textit{ and} \\ Q_{ret} &= [\text{UPLREQ}\,] \textit{ and} \\ Q_{loc} &= [\text{SIGT0, TMP1, TMP2, SIGUP}\,] \textit{ and} \\ \Psi &= \text{the statements between the BEGIN and the END} \end{aligned}$$

The name of the method is the symbol UPLREQ. The data item C-12 holds the target object passed to the method. UPLREQ expects no other objects or parameters to be passed since the Q_{obj} and Q_{form} sequences are empty. The data item returned from this method is represented by including the name of the method in the Q_{ret} sequence. The local variables of UPLREQ are the data items SIGTO, TMP1, TMP2, and SIGUP. The sequence of object-oriented statements between the BEGIN and the END of UPLREQ comprise Ψ.

4.31 Formalizing Object-Oriented Messages

This section explains how messages represented using the `gom-message` AST can be defined more mathematically. Messages are defined formally below.

id_o - a symbol that holds the name of the method to be invoked.

Q_{act} - a sequence of variables that are the actual parameters passed to the method.

Q_{form} - a sequence of variables that are formal parameters of the method to be invoked.

Let C_1 be a class that has a method o_1. Let C_2 be a class that has a method o_2 that is invoked in o_1. Then, the message sent in method o_1 is represented by the tuple

$$< id_{o_2}, Q_{act} >$$

In the object-oriented paradigm, the number, order, and type of parameters in Q_{act} must match the number, order, and type of the parameters in Q_{form}.

```
class CLASS-5 attributes PHASE, ZENITH, UPLFAC
  method RELAY-PHASE ( C-14, C-22, PHASE )
    begin
    PHASE := UPLREQ ( C-22 )
    end
  superclass USER-OBJECT
```

Figure 96 Class with method RELAY-PHASE

For example, as shown in Figure 96, the message `UPLREQ( C-22 )` is being sent by the `RELAY-PHASE` method. The target object of this message is `C-22` and this instance is being asked to execute the method `UPLREQ`. The object `C-22` is an instance of the class `CLASS-2`. Figure 97 shows the class `CLASS-2` and the (partially declared) method `UPLREQ`. The message `UPLREQ( C-22 )` is defined formally by the tuple

$$< UPLREQ, \ [\ \text{C-22}\ ] >$$

Since this message is invoking the `UPLREQ` method, the `UPLREQ` identifier is the first item in the tuple. The singleton sequence containing `C-22` represents that this call is passing in a single parameter, which is the target object `C-22`.

```
class CLASS-2 attributes ZCOS, XLAMDA, SIGTRB
  method UPLREQ ( C-12 ): real
    begin
      ...
    end
  superclass USER-OBJECT
```

Figure 97 Class with method UPLREQ

4.32 Formalizing the Object Model

Rumbaugh [62] defines an *object model* in his Object Modeling Technique (OMT). The object model includes definitions for the classes, inheritance, and aggregation associations in an object-oriented design. This section provides a formal definition of an object-oriented design, a discussion of the inheritance and aggregation associations in the design, and formal transformations for the object model.

The formal definition of an object-oriented design is given below. Let C^* be a set of object-oriented classes to be included in an object-oriented design. An object-oriented design is represented formally using the following one-tuple.

$$OOD = C^*$$

An object-oriented design consists of the set of classes to be included in the design. One of these classes in the design is the *system class* that includes the main method, which is given the flow of control when execution begins.

Inheritance associations are represented in a design by references to the superclass in a class description, i.e. the λ item in the tuple representing a class. An aggregation association exists between two classes when object instances of one class are stored as instance attributes of the other class.

Blaha [7] discusses several *primitive* transformations for object models. The transformations defined to formalize the PBOI method include removing or adding a class and removing or adding an attribute. A mathematical description for these transformations is provided below.

Let T_c^+ be a transformation of the object model that adds a new class to an existing design and returns a new object-oriented design. Thus,

$$\begin{aligned} T_c^+(OOD, C) &= OOD' \text{ where} \\ OOD' &= OOD \cup \{C\} \end{aligned}$$

Let T_c^- be a transformation of the object model that removes a class and returns a new object-oriented design. Thus,

$$\begin{aligned} T_c^-(OOD, C) &= OOD' \text{ where} \\ OOD' &= OOD - \{C\} \end{aligned}$$

Let T_a^+ be a transformation of the object model that adds a new attribute to a specific class and returns a new object-oriented design. Thus,

$$\begin{aligned} T_a^+(OOD, C, a) &= OOD' \text{ where} \\ C &= < id_C, \Phi_C, \Omega_C, \lambda > \ and \\ C' &= < id_C, \Phi_C \cup \{a\}, \Omega_C, \lambda > \ and \\ OOD' &= T_c^+(T_c^-(OOD, C), C') \end{aligned}$$

Let T_a^- be a transformation of the object model that removes an attribute of a specific class and returns a new object-oriented design. Thus,

$$\begin{aligned} T_a^-(OOD, C, a) &= OOD' \text{ where} \\ C &= < id_C, \Phi_C, \Omega_C, \lambda > \ and \\ C' &= < id_C, \Phi_C - \{a\}, \Omega_C, \lambda > \ and \\ OOD' &= T_c^+(T_c^-(OOD, C), C') \end{aligned}$$

Blaha [7] does not address the addition or removal of a method from a class. These additional transformations are introduced below for adding and removing a method, o, of a class, C.

Let T_m^+ be a transformation that adds a new method to a class and returns a new object-oriented design. Thus,

$$\begin{aligned}
T_m^+(OOD, C, o) &= OOD' \text{ where} \\
C &= < id_C, \Phi_C, \Omega_C, \lambda > \ \textit{and} \\
C' &= < id_C, \Phi_C, \Omega_C \cup \{o\}, \lambda > \ \textit{and} \\
OOD' &= T_c^+(T_c^-(OOD, C), C')
\end{aligned}$$

Let T_m^- be a transformation that removes a method from a class and returns a new object-oriented design. Thus,

$$\begin{aligned}
T_m^-(OOD, C, o) &= OOD' \text{ where} \\
C &= < id_C, \Phi_C, \Omega_C, \lambda > \ \textit{and} \\
C' &= < id_C, \Phi_C, \Omega_C - \{o\}, \lambda > \ \textit{and} \\
OOD' &= T_c^+(T_c^-(OOD, C), C')
\end{aligned}$$

4.33 Summary

This chapter has defined the Generic Object-Oriented Design Model (GOM) which models objects, classes, methods, messages, assignment, variables, expressions, and control flow in object-oriented programming languages. Several restrictions have been placed on the GOM. Some restrictions are intended to canonicalize the representations of object-oriented entities. Other restrictions place limits on which OO entities can possibly be modeled using the GOM.

Specifically, Restriction IV.1 states that neither functional nor procedural methods can have a parameter that is both and input and an output parameter. This restriction is not unreasonable since any method that does not meet this restriction can be converted using a process similar to the one presented in Appendix D.

Similarly, Restriction IV.2 does not allow functional methods to return more than one data item. Hypothetically, functional methods of this type can be converted to procedural methods that return multiple values.

Restriction IV.3 is intended to canonicalize the accessing and assigning of object attributes and does not limit which entities can be model using the GOM.

Restrictions IV.6 and IV.7 limit messages to non-recursive invocations. This restriction is reasonable because Knuth [37] argues any recursive algorithm can be represented using an iterative algorithm.

Restriction IV.8 does not allow *global* variables to appear in methods. This means methods with variables other than formal parameters or local variables must first be converted into the proper form. It is hypothetically possible to convert global variables into formal parameters throughout the call tree of an object-oriented system, so this restriction is not unreasonable.

Restriction IV.9 does not allow one method to be declared inside another method. The scoping issues involved in this nesting of methods is more complicated than that of variables, but it is hypothetically possible to convert the nested methods into methods declared at the global level. Similarly, Restriction IV.10 does not allow one class to be declared inside another class. This is a minor restriction since declaring classes inside of other classes is not a common practice in object-oriented programming.

Restriction IV.4 requires that only calls to non-user-defined subprograms are modeled by subprogram invocations in the GOM. This means that the GOM represents a "pure" object-oriented design where no mixing of the imperative paradigm and the object-oriented paradigm is allowed. This restriction presents a limitation if the design to be modeled does include such a mixture.

None of the restrictions discussed above preclude the representation of object-oriented entities in the GOM. However, Restriction IV.11 states that heterogeneous data types are not modeled in the GOM. Restriction IV.12 states that pointers are not modeled in the GOM. These two restrictions limit the entities that can be modeled by the GOM. If the object-oriented system being modeled includes either heterogeneous data types or pointers, the system cannot be represented using the GOM. This is not unreasonable when re-engineering from the GIM, but is a more serious limitation for representing languages such as C++ or Ada 95.

As presented in this chapter, the GOM provides a canonical form for modeling entities in the object paradigm. Most of the restrictions placed on the GOM are meant to canonicalize and simplify the representation of these entities. The GOM is used throughout the following chapters to represent object-oriented entities.

V. Identifying Objects

5.1 Introduction

This chapter introduces a novel method, Parameter-Based Object Identification (PBOI), for recovering objects from legacy imperative programs. The method is based on fundamental axioms that relate programs written in imperative languages such as C or Cobol to objects and classes written in object-oriented languages such as Ada 95 or C++. This chapter provides an informal introduction to the object identification methodology. Transformations that formalize the methodology are presented in Chapter VI.

In order to focus the task of transforming imperative subprograms to the object-oriented paradigm, a taxonomy of imperative subprograms has been developed. This chapter describes the taxonomy, which classifies all imperative subprograms into one of six categories and explains how the object identification methodology is used to convert subprograms in each of the categories. Finally, the chapter describes how the process of *program slicing* [76] is used to focus the transformation task even further.

5.2 Taxonomy of Imperative Subprograms

This section describes the taxonomy of imperative subprograms built to focus the task of re-engineering imperative legacy code to the object paradigm. By defining this taxonomy, an imperative subprogram can be converted to the object paradigm by determining the classification of the subprogram and then applying the formal transformation appropriate to that category of subprogram. The taxonomy reduces the re-engineering task to defining the transformations for the six categories of subprograms identified. The following definitions are used to build the taxonomy.

Definition V.1. *One subprogram* calls *another subprogram by using the name of the subprogram to invoke the other subprogram.*

Definition V.2. *C_{sub} is the collection of calls a subprogram makes to other subprograms.*

Definition V.3. *An imperative procedure* produces *a data item if the data item is an output parameter.*

Definition V.4. *An imperative function* produces *the data item returned at the end of its execution.*

Definition V.5. *P_{pro} is the collection of data items* produced *by a subprogram.*

The taxonomy of imperative subprograms considers whether or not a subprogram calls another subprogram and whether the subprogram produces zero, one, or multiple data items. The distinction between producing zero or one data item is important for the classification of the *main program*, i.e. the subprogram that is given the flow of control as the system begins execution. Figure 98 shows the taxonomy of imperative subprograms based on the sizes of C_{sub} and P_{pro}.

	$\mid P_{pro} \mid = 0$	$\mid P_{pro} \mid = 1$	$\mid P_{pro} \mid > 1$
$\mid C_{sub} \mid = 0$	0	2	4
$\mid C_{sub} \mid > 0$	1	3	5

Figure 98 Subprogram Taxonomy

Category 0 subprograms produce no data items and call no other subprograms. These subprograms could consist of *output* statements or busy/wait loops. Category 1 subprograms produce no data items but call other subprograms. These subprograms could be driver programs or the main program. Category 2 subprograms produce a single data item and call no other subprograms. Category 3 subprograms produce a single data item and call other subprograms. Category 4 subprograms produce multiple data items and call no other subprograms. Category 5 subprograms produce multiple data items and call other subprograms.

Since any imperative subprogram can be classified into one of these six categories, the process of transforming the subprogram from the GIM to the GOM is reduced to building transformations for each category.

5.3 Parameter-Based Object Identification

Parameter-Based Object Identification (PBOI) is a novel method for identifying objects in imperative legacy code based on the data items passed as parameters in imperative subprogram calls. The PBOI method provides a rationale for converting imperative subprograms into classes and methods that implement the subprograms. The overall rationale for the PBOI method is based on the following thesis.

Thesis V.1. Object attributes manifest themselves as data items passed from subprogram to subprogram in the imperative paradigm.

Object attributes are extracted from the parameters of imperative subprograms using specific transformations defined by the PBOI methodology. Once the attributes of an object are built into a class, the behavior associated with the attributes is built into the class.

In this methodology, it is assumed that all data items in an imperative subprogram are either passed as parameters to the subprogram or declared locally in the subprogram. In this way, the *main program* is the origin of all data items being passed to the imperative subprograms.

The PBOI methodology starts the transformation process at the main program. Before the main program is transformed each of the subprograms called by the main program is transformed to the object-oriented paradigm. Similarly, each of the subprograms called by a Category 1, 3, or 5 subprogram is converted before the subprogram itself is converted. This *depth-first* style of transformation results in incremental designs being merged together at different steps in the transformation.

The following sections define the PBOI method in detail as it applies to the six categories of imperative subprograms. The discussions of the different categories are not presented in numerical order, but in an order that more closely fits the incremental nature of the PBOI method.

5.4 Converting Category 2 Subprograms

As a starting point, consider the conversion of Category 2 imperative subprograms. These subprograms produce one data item without calling any other subprograms, so an analogous entity is needed from the object paradigm. In his Object Modeling Technique (OMT), Rumbaugh [62] distinguishes between object-oriented operations that have side effects and those that merely compute a functional value. He labels the latter form of operation *queries.* Queries with no arguments except the target object are considered *derived attributes* [62] of the object. Imperative functions fit this description, so the PBOI method uses the following thesis to convert Category 2 functions to the object-oriented paradigm.

Thesis V.2. Category 2 imperative functions implement *derived attribute* queries of objects.

Category 2 imperative procedures have a single output parameter, so the PBOI method uses the following thesis to convert Category 2 procedures to the object-oriented paradigm.

Thesis V.3. Category 2 imperative procedures implement *side-effecting* operations of objects.

Based on these two theses, the PBOI methodology converts each formal parameter of a Category 2 subprogram into an attribute of a class and transforms the subprogram into a method of the class.

```
real function CAPTURE
    ( BEAMRA, AXMAJ, AXMIN, ANGFAC )
  begin
  TILTFA := DSIN ( ANGFAC);
  RMAREA := AXMAJ * AXMIN;
  EFAREA := RMAREA * TILTFA;
  BMAREA := BEAMRA * BEAMRA;
  if EFAREA <= BMAREA * 6.0d0
    then USED := EFAREA / BMAREA;
      CAPTURE := 1.0d0 - DEXP ( -USED)
    else CAPTURE := 1.0d0 endif
  end
```

Figure 99   Imperative function CAPTURE

For example, consider the imperative function `CAPTURE` shown in Figure 99. This function is shown using the GIL surface syntax. The `CAPTURE` function is a Category 2 subprogram that returns a single value of type *real.* The parameters `BEAMRA`, `AXMAJ`, `AXMIN`, and ANGFAC are passed into the function and used by the statements of the function to calculate the value returned. The CAPTURE function is converted to the object-oriented paradigm using the PBOI method. Figure 100 shows the class and method built for CAPTURE.

```
class CLASS-2 attributes
    AXMIN, AXMAJ, BEAMRA, ANGFAC
  method CAPTURE ( C-2 ): real
    begin
    TILTFA := DSIN ( GET-ANGFAC ( C-2));
    RMAREA := GET-AXMAJ ( C-2)
      * GET-AXMIN ( C-2);
    EFAREA := RMAREA * TILTFA;
    BMAREA := GET-BEAMRA ( C-2)
      * GET-BEAMRA ( C-2);
    if EFAREA <= BMAREA * 6.0d0
      then USED := EFAREA / BMAREA;
        CAPTURE := 1.0d0 - DEXP ( -USED)
      else CAPTURE := 1.0d0 endif
    end
  superclass USER-OBJECT
```

Figure 100 Class and method built for CAPTURE

This class and method are represented in the figure using the GOL surface syntax. Note that each of the parameters BEAMRA, AXMAJ, AXMIN, and ANGFAC are attributes of the class CLASS-2. The method that implements the CAPTURE subprogram has been built as the single method of CLASS-2. This method expects one parameter, C-2, which is an instance of CLASS-2, and the method returns a value of type *real.* The statements of the method now access the BEAMRA, AXMAJ, AXMIN, and ANGFAC data items from the object C-2 in order to calculate the value returned from the method.

Because of Restriction IV.3, the BEAMRA, AXMAJ, AXMIN, and ANGFAC attributes are accessed using the "GET-" and "SET-" methods. Each "GET-" and "SET-" method is generated for each attribute and added to the set of operations for the class. Similarly, a method to instantiate instances of the class is needed, so a "CREATE-" method is generated for each class. These methods are not generated until the main program is transformed, which explains why they do not appear in Figure 100. The transformation that formalizes the generation of these methods is presented in Section 6.11.2.

The conversion of Category 2 procedures is identical to the conversion of Category 2 functions except that the single output parameter is converted to an attribute of the new class built for the subprogram. By using this part of the PBOI method, all Category 2 subprograms can be converted to the object-oriented paradigm.

5.5 Converting Category 0 Subprograms

A Category 0 subprogram produces no data items and calls no other subprograms. These subprograms may be routines that include `imperative-output` statements, or they may be needed in the system for timing considerations. Category 0 subprograms are treated as a special case of Category 2 subprograms. All Category 0 subprograms are transformed to the object-oriented paradigm by using the transformation defined for Category 2 subprograms.

5.6 Converting Category 3 Subprograms

Category 3 subprograms produce a single data item and call other subprograms. Since Category 3 subprograms call other subprograms, the attributes of objects that have already been built can be analyzed using the PBOI methodology. This section describes the depth-first analysis that is done to the object-oriented design while transforming Category 3 subprograms to the object-oriented paradigm.

The first step in converting a Category 3 subprogram is to transform all of the subprograms that it calls. Then a new class with a method that implements the Category 3 subprogram is built. Initially this class has an attribute built from each of the parameters of the subprogram, as was done for Category 2 subprograms. As Category 3 subprograms are converted to the object-oriented paradigm, the attributes of this class, as well as the attributes of the other classes in the design, are *filtered* to determine which should remain attributes and which should be converted from attributes to parameters of the methods.

Specifically, consider the case where a Category 3 subprogram S_1 makes one call to another subprogram S_2 (S_2 may be from any category in the taxonomy). Each of the *actual* parameters from S_1 in the call to S_2 is linked to a corresponding *formal* parameter in the declaration of S_2. Let C_1 represent the class built for S_1 and let C_2 represent the class built for S_2.

An actual parameter from S_1 may or may not also be a formal parameter of S_1, and the corresponding formal parameter in S_2 may be an attribute of C_2 or a parameter of the method in C_2. Figure 101 shows the four cases considered for each actual parameter when converting a Category 3 subprogram. In each PBOI case, a specific change is made to the

	Actual *is* Formal	Actual is *not* Formal
Formal is Attribute	PBOI Case 1	PBOI Case 3
Formal is Parameter	PBOI Case 2	PBOI Case 4

Figure 101 PBOI Cases

design developed so far as explained below.

PBOI Case 1 The data item remains an attribute of C_2 and is removed as an attribute of C_1.

PBOI Case 2 The data item is converted from an attribute of C_1 to a parameter of C_1's method.

PBOI Case 3 The data item is converted from an attribute of C_2 to a parameter of C_2's method.

PBOI Case 4 No change is required for the design.

In Case 1, the data item has already been made an attribute of C_2, so it should not be built as an attribute of another class, i.e. C_1. In Case 2, it has been determined through some earlier filtering that the data item should not be an attribute, so it is not built as an attribute of C_1. In Case 3, the data item is not being passed down from the main program, so it is not part of an object. Thus, it is filtered out of C_2 and moved to a parameter. In Case 4, the data item is not being passed down and it is already a parameter instead of an attribute. These four cases are used to evaluate the entire design built so far when each

Category 3 subprogram is converted to the object-oriented paradigm. For clarification, an example is given in the following sections for each of the PBOI cases.

5.6.1 PBOI Case 1. As an example of PBOI Case 1, consider the Category 3 subprogram BOUNCE shown in Figure 102 (using the GIL surface syntax). The BOUNCE

```
real function BOUNCE ( ANGFAC, RNGFAC, PROJRA,
    SIGB, RANGE, AXMAJ, AXMIN, XLAMDA)
  begin
  RHOSTD := 0.95 * PROJRA * ANGFAC;
  BEAMRA :=
    RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
  BOUNCE :=
    CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
      * RHO ( RHOSTD, XLAMDA, ANGFAC)
  end
```

Figure 102 Subprogram BOUNCE

subprogram calls the Category 2 subprogram RADIUS shown in Figure 103, as well as the CAPTURE and RHO subprograms (not shown in the figures). Figure 104 shows the class built for the RADIUS subprogram (using the GOL surface syntax) by using the rule for Category 2 subprograms. Note that the formal parameters of RADIUS have been built as attributes of the class CLASS-1.

```
real function RADIUS
    ( PROJRA, SIGB, RNGFAC, RANGE )
  begin
  if RNGFAC > 0.0 and RANGE > 0.0
    then SIGABS := DABS ( SIGB);
      SLOPE := SIGABS - PROJRA / RANGE;
      SPOT := SIGABS * RANGE;
      RAD := DABS ( PROJRA + SLOPE * RNGFAC);
      RADIUS := DMAX1 ( SPOT, RAD)
    else
      RADIUS := PROJRA endif
  end
```

Figure 103 Subprogram RADIUS

In the call to RADIUS, the actual parameters PROJRA, SIGB, RNGFAC and RANGE are also formal parameters of the BOUNCE subprogram. These four data items provide an example

```
class CLASS-1 attributes
    RANGE, RNGFAC, SIGB, PROJRA
  method RADIUS ( C-1 ): real
    begin
    if GET-RNGFAC ( C-1) > 0.0 and
         GET-RANGE ( C-1) > 0.0
      then SIGABS := DABS ( GET-SIGB ( C-1));
        SLOPE := SIGABS - GET-PROJRA ( C-1)
          / GET-RANGE ( C-1);
        SPOT := SIGABS * GET-RANGE ( C-1);
        RAD := DABS ( GET-PROJRA ( C-1) +
          SLOPE * GET-RNGFAC ( C-1));
        RADIUS := DMAX1 ( SPOT, RAD)
      else RADIUS := GET-PROJRA ( C-1) endif
    end
  superclass USER-OBJECT
```

Figure 104 Class and method built for RADIUS

of PBOI Case 1 and remain attributes of CLASS-1.

Figure 105 shows the initial class and method built for the Category 3 subprogram BOUNCE. Note that the subprogram calls to RADIUS, CAPTURE, and RHO have not yet been transformed to messages, i.e. the transformation is incomplete. Also note that each of the parameters from the subprogram BOUNCE has been built as an attribute of the class CLASS-4 including the formal parameters PROJRA, SIGB, RNGFAC and RANGE. Since these data items are passed to the RADIUS subprogram and have been built as attributes of the class built for RADIUS, they are examples of PBOI Case 1 and are converted from attributes of CLASS-4 to attributes of CLASS-1.

Figure 106 shows CLASS-4 after the attributes PROJRA, SIGB, RNGFAC and RANGE have been converted from CLASS-4 to CLASS-1. As shown in the figure, these data items are no longer attributes of CLASS-4. The message GET-PROJRA is now sent to the object C-5, an instance of CLASS-1, instead of being sent to the object C-4, an instance of CLASS-4.

```
class CLASS-4 attributes ANGFAC, RNGFAC, PROJRA,
      SIGB, RANGE, AXMAJ, AXMIN, XLAMDA
  method BOUNCE ( C-4 ): real
    begin
    RHOSTD := 0.95 * GET-PROJRA ( C-4)
      * GET-ANGFAC ( C-4);
    BEAMRA :=
      RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
    BOUNCE :=
      CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
        * RHO ( RHOSTD, XLAMDA, ANGFAC)
    end
  superclass USER-OBJECT
```

Figure 105   Initial class built for BOUNCE

The call to the subprogram RADIUS has been transformed into a message sent to C-5 that invokes the RADIUS method of CLASS-1.

5.6.2 PBOI Case 2. As an example of PBOI Case 2, consider another version of CLASS-1 shown in Figure 107 that might have been built for the RADIUS subprogram. The class shown in the figure includes the PROJRA data item as a parameter to the RADIUS method instead of as an attribute of the class CLASS-1. This data item provides an example of PBOI Case 2. Figure 105 shows the initial class built for the BOUNCE subprogram. Each of the formal parameters of the subprogram BOUNCE has been built as an attribute of CLASS-4, including the PROJRA data item. In this example, the PROJRA data item is passed to the RADIUS subprogram but was *not* built as a parameter of the class built for RADIUS. This is an example of PBOI Case 2, so the data item PROJRA is transformed from an attribute of CLASS-4 into a parameter of the methods of CLASS-4.

Figure 108 shows the updated class after converting the data item PROJRA from an attribute of CLASS-4 into a parameter of the BOUNCE method. The PROJRA data item is passed as an actual parameter to the RADIUS method. The message GET-PROJRA (C-4) that was used to access PROJRA as an attribute of the class has been converted to an access of the parameter PROJRA. The class in the figure has also been updated to convert the data items SIGB, RNGFAC and RANGE from attributes of CLASS-4 to attributes of CLASS-1 since

```
class CLASS-4 attributes
    ANGFAC, AXMAJ, AXMIN, XLAMDA
  method BOUNCE ( C-4, C-5 ): real
    begin
    RHOSTD := 0.95 * GET-PROJRA ( C-5)
      * GET-ANGFAC ( C-4);
    BEAMRA := RADIUS ( C-5);
    BOUNCE :=
      CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
        * RHO ( RHOSTD, XLAMDA, ANGFAC)
    end
  superclass USER-OBJECT
```

Figure 106 Updated class built for BOUNCE

these data items are examples of PBOI Case 1. The message RADIUS (C-5, PROJRA) now matches the signature of the method RADIUS and is passed an instance of CLASS-1 and the parameter PROJRA.

5.6.3 PBOI Case 3. As an example of PBOI Case 3, Figure 109 shows the Category 3 imperative subprogram BOUNCE and the Category 2 subprogram CAPTURE, which is called by BOUNCE. The class originally built for the CAPTURE subprogram is shown in Figure 110. Since CAPTURE is a Category 2 subprogram, each of the parameters, including the data item BEAMRA, is converted into an attribute of the class CLASS-2. The BEAMRA data item is not a formal parameter of the subprogram BOUNCE, so it does not remain an attribute of CLASS-2. This data item is an example of PBOI Case 3, so it must be converted from an attribute of CLASS-2 into a parameter of the CAPTURE method. Figure 111 shows CLASS-2 after this change has been made. Figure 105 shows the initial class built for the BOUNCE subprogram. The data item BEAMRA is not a formal parameter of the BOUNCE subprogram, so it has not been built as a parameter of the class CLASS-4. There is no change required by PBOI Case 3 to the attributes of CLASS-4 or the parameters of the BOUNCE method. The only change made to the statements of the method is to ensure the BEAMRA data item is passed as a parameter of the CAPTURE message. To illustrate, note

```
class CLASS-1 attributes RANGE, RNGFAC, SIGB
  method RADIUS ( C-1, PROJRA ): real
    begin
    if GET-RNGFAC ( C-1) > 0.0 and
         GET-RANGE ( C-1) > 0.0
      then SIGABS := DABS ( GET-SIGB ( C-1));
        SLOPE := SIGABS - PROJRA
          / GET-RANGE ( C-1);
        SPOT := SIGABS * GET-RANGE ( C-1);
        RAD := DABS ( PROJRA + SLOPE *
          GET-RNGFAC ( C-1));
        RADIUS := DMAX1 ( SPOT, RAD)
      else RADIUS := PROJRA  endif
    end
  superclass USER-OBJECT
```

Figure 107   Class built for RADIUS

that the AXMAJ, AXMIN, and ANGFAC data items are examples of PBOI Case 1, so they are converted from attributes of CLASS-4 to attributes of CLASS-2.

Figure 112 shows CLASS-4 after this transformation has been done and the call to CAPTURE has been converted into a message. As shown in the figure, the message CAPTURE (C-6, BEAMRA) invokes the CAPTURE method in CLASS-2. The instance variable C-6 now includes the AXMAJ, AXMIN, and ANGFAC data items as attributes (because of PBOI Case 1), and the BEAMRA data item is passed as an actual parameter of the message (because of PBOI Case 3).

5.6.4 PBOI Case 4. As an example of PBOI Case 4, consider the version of CLASS-2 shown in Figure 111. This class has been built for the subprogram CAPTURE and, for this example, the data items AXMAJ, AXMIN, and ANGFAC have been built as attributes of the class. For this example, the data item BEAMRA has been built as a parameter of the CAPTURE method. As shown in Figure 109, the BEAMRA data item is a formal parameter of the CAPTURE subprogram.

Figure 113 shows the initial class built for the BOUNCE subprogram, as originally presented in Figure 105. The BEAMRA data item is not a formal parameter of the BOUNCE

```
class CLASS-4 attributes
    ANGFAC, AXMAJ, AXMIN, XLAMDA
  method BOUNCE ( C-4, C-5, PROJRA ): real
    begin
    RHOSTD := 0.95 * PROJRA
      * GET-ANGFAC ( C-4);
    BEAMRA := RADIUS ( C-5, PROJRA);
    BOUNCE :=
      CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
        * RHO ( RHOSTD, XLAMDA, ANGFAC)
    end
  superclass USER-OBJECT
```

Figure 108 Updated class built for BOUNCE

subprogram (as shown in Figure 109). Since BEAMRA is not a formal parameter and it has not been built as an attribute of CLASS-2, it is an example of PBOI Case 4. In this case, there are no changes required to either CLASS-2 or CLASS-4. The only change made to the statements of the method BOUNCE is to ensure that BEAMRA is included as an actual parameter of the CAPTURE message.

Figure 114 shows CLASS-4 after the data items AXMAJ, AXMIN, and ANGFAC are converted from attributes of CLASS-4 to attributes of CLASS-2 (because of PBOI Case 1) and the call to CAPTURE has been converted to a message. As shown in the figure, the message CAPTURE (C-6, BEAMRA) includes C-6, an instance of CLASS-2, and the data item BEAMRA passed as actual parameters.

5.6.5 Eliminating Duplicate Classes. In the call tree of an imperative design, it is possible that one subprogram will have multiple calls to another subprogram. Since the first step in converting a Category 3 subprogram is to transform all of the subprograms that it calls, the subprogram that is called multiple times will be converted multiple times. This introduces *duplicate* classes into the design returned to the Category 3 subprogram. Furthermore, there may be different combinations of PBOI cases for each of the calls to a subprogram. This means the signature of the method built for one call to a subprogram may not match the signature of the method built for another call to the subprogram.

```
real function CAPTURE
    ( BEAMRA, AXMAJ, AXMIN, ANGFAC )
  begin
  TILTFA := DSIN ( ANGFAC);
  RMAREA := AXMAJ * AXMIN;
  EFAREA := RMAREA * TILTFA;
  BMAREA := BEAMRA * BEAMRA;
  if EFAREA <= BMAREA * 6.0d0
    then USED := EFAREA / BMAREA;
      CAPTURE := 1.0d0 - DEXP ( -USED)
    else CAPTURE := 1.0d0 endif
  end

real function BOUNCE
    ( ANGFAC, RNGFAC, PROJRA, SIGB,
      RANGE, AXMAJ, AXMIN, XLAMDA)
  begin
  RHOSTD := 0.95 * PROJRA * ANGFAC;
  BEAMRA :=
    RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
  BOUNCE :=
    CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
      * RHO ( RHOSTD, XLAMDA, ANGFAC)
  end
```

Figure 109 Subprograms CAPTURE and BOUNCE

To resolve this possible mismatch between the signature of methods in duplicate classes, the duplicate classes are identified and eliminated as part of the conversion of Category 3 subprograms. Two duplicate classes are replaced in the design with a new class that is built as follows. The sets of attributes from the two duplicate classes are compared and any attributes not in the intersection are moved from attributes to parameters. The attributes in the intersection become attributes of the new class. Since both duplicate classes include a method that implements a single subprogram, the set of operations for the new class includes the method built for this subprogram. The resulting class replaces the two duplicate classes in the design. If any other methods include calls to the methods from the duplicate classes, these calls are updated to match the signature of the method in the new class.

```
class CLASS-2 attributes
    AXMAJ, AXMIN, ANGFAC, BEAMRA
  method CAPTURE ( C-2 ): real
    begin
    TILTFA := DSIN ( GET-ANGFAC ( C-2));
    RMAREA := GET-AXMAJ ( C-2) *
      GET-AXMIN ( C-2);
    EFAREA := RMAREA * TILTFA;
    BMAREA := GET-BEAMRA ( C-2) *
      GET-BEAMRA ( C-2);
    if EFAREA <= BMAREA * 6.0d0
      then USED := EFAREA / BMAREA;
          CAPTURE := 1.0d0 - DEXP ( -USED)
      else CAPTURE := 1.0d0 endif
    end
  superclass USER-OBJECT
```

Figure 110 Original class built for CAPTURE

For example, Figure 115 shows two classes built for the subprogram CAPTURE. The signatures of the two methods do not match since BEAMRA is passed as a parameter in one version and ANGFAC is passed as a parameter in the other. The classes are easily identified as duplicates because the names of the two classes are the same. These duplicate classes are replaced in the design by the class shown in Figure 116. The new version of CLASS-2 includes attributes from the intersection of the sets of attributes from the duplicate classes. The single operation in the new class is the method from the duplicate classes updated to properly access the attributes and parameters of the new class.

Duplicate classes may also be introduced into the design when different subprograms call the same subprogram. These duplicates will be identified and eliminated when the call subtrees that include the two calling subprograms join. This is guaranteed to happen at least at the main program.

5.7 Converting Category 1 Subprograms

Category 1 subprograms produce no data items but call other subprograms. These subprograms provide another opportunity to extract objects using the PBOI methodology.

```
class CLASS-2 attributes AXMAJ, AXMIN, ANGFAC
  method CAPTURE ( C-2, BEAMRA ): real
    begin
    TILTFA := DSIN ( GET-ANGFAC ( C-2));
    RMAREA := GET-AXMAJ ( C-2) *
      GET-AXMIN ( C-2);
    EFAREA := RMAREA * TILTFA;
    BMAREA := BEAMRA * BEAMRA;
    if EFAREA <= BMAREA * 6.0d0
     then USED := EFAREA / BMAREA;
          CAPTURE := 1.0d0 - DEXP ( -USED)
     else CAPTURE := 1.0d0 endif
    end
  superclass USER-OBJECT
```

Figure 111 Updated class built for CAPTURE

Typically, the main program is classified as a Category 1 subprogram. All Category 1 subprograms *other than the main program* are transformed using the PBOI methodology defined for Category 3 subprograms. Because of the unique handling of the main program, the following assumption applies.

Assumption V.1. *The main program of the system that is being converted to the object paradigm is uniquely identified.*

There is no attempt in this research to identify the main program automatically. Since it is at the top of the *call tree* of imperative subprograms, such automatic identification of the main program should be straightforward to implement. The main program is transformed to the object-oriented paradigm as described in the following section.

5.8 Converting the Main Program

The main program of a system provides a unique opportunity to update the object-oriented design developed so far and is transformed using the rationale defined in this section. Before the main program is transformed, each of the subprograms called by the main program is transformed to the object-oriented paradigm. This *depth-first* transformation results in an object-oriented design that includes classes built for all of the sub-

```
class CLASS-4 attributes  RNGFAC, PROJRA, SIGB,
    RANGE, XLAMDA
  method BOUNCE ( C-4, C-6 ): real
    begin
    RHOSTD := 0.95 * GET-PROJRA ( C-4)
      * GET-ANGFAC ( C-6);
    BEAMRA :=
      RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
    BOUNCE :=
      CAPTURE ( C-6, BEAMRA)
        * RHO ( RHOSTD, XLAMDA, ANGFAC)
    end
  superclass USER-OBJECT
```

Figure 112 Updated class built for BOUNCE

programs in the system (except the main program). A special class is built for the main program to represent the overall *system* class in the object-oriented design. This class has no attributes and only one method which implements the main program. The name of this class is CLASS-SYSTEM. In this way, the entire imperative design is converted into an object-oriented design using depth-first transformations.

In this research, an assumption is made that all data items in the imperative subprograms are either passed as parameters to the subprogram or declared locally in the subprogram; thus the main program is the origin of all data items being passed to the imperative subprograms. For this reason, the data items in the main program are used to create all of the instances required throughout the entire object-oriented design. Creating every object instance from data items in the main program leads to the following thesis.

Thesis V.4. Attributes that can be linked to the same main program data item are part of the same object.

This is used as the rationale for two design updates done at this stage of the main program transformation, as discussed in Section 5.8.1 and Section 5.8.2.

Since each of the subprograms called by the main program has been converted to a method, the subprogram calls in the main program must be transformed into messages that invoke these methods. Each method includes one or more object instances in its

```
class CLASS-4 attributes ANGFAC, RNGFAC, PROJRA,
      SIGB, RANGE, AXMAJ, AXMIN, XLAMDA
  method BOUNCE ( C-4 ): real
    begin
    RHOSTD := 0.95 * GET-PROJRA ( C-4)
      * GET-ANGFAC ( C-4);
    BEAMRA :=
      RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
    BOUNCE :=
      CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
        * RHO ( RHOSTD, XLAMDA, ANGFAC)
    end
  superclass USER-OBJECT
```

Figure 113   Initial class built for BOUNCE

signature, so these instances must be created by the main program. Data items which appear as parameters in the imperative paradigm must be built as attributes of object instances in the object paradigm. The assignment statements that create these instances are inserted in the sequence of main program statements just before the message.

For example, Figure 117 shows the partial declaration of the subprogram LNKCAL-DWELLT. Assume this subprogram is called by the main program BMDSIM1. The formal parameters BETA, XLAMDA, and DIAM are shown in the figure as part of the signature of this subprogram. Figure 118 shows the partial declaration of the class CLASS-20 which has the method LNKCAL-DWELLT that implements the subprogram LNKCAL-DWELLT from Figure 117. The signature of the method is radically different from the signature of the subprogram because the data items BETA, XLAMDA, and DIAM are now attributes of a class and do not appear as parameters in the signature of the method. Instead, an instance of the class that has BETA, XLAMDA, and DIAM as attributes is included in the method LNKCAL-DWELLT, viz. C-22. The instance object C-22 is a formal parameter of the method LNKCAL-DWELLT and a corresponding object must be passed as an actual parameter of any messages that invoke the method LNKCAL-DWELLT. Since, in this example, the LNKCAL-DWELLT method is invoked by the main program BMDSIM1, the main program must build an instance to send in the message that invokes LNKCAL-DWELLT.

```
class CLASS-4 attributes  RNGFAC, PROJRA, SIGB,
    RANGE, XLAMDA
  method BOUNCE ( C-4, C-6 ): real
    begin
    RHOSTD := 0.95 * GET-PROJRA ( C-4)
      * GET-ANGFAC ( C-6);
    BEAMRA :=
      RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
    BOUNCE :=
      CAPTURE ( C-6, BEAMRA)
        * RHO ( RHOSTD, XLAMDA, ANGFAC)
    end
  superclass USER-OBJECT
```

Figure 114 Updated class built for BOUNCE

Figure 119 shows the class CLASS-5 that, in this example, is the class of which C-22 is an instance. Note the method CREATE-CLASS-5 shown in the figure. This method is used to build new instances of CLASS-5 and is generated automatically when using the PBOI methodology. Section 6.11.2 presents the transformation that formalizes the generation of this method. Figure 120 shows the class CLASS-SYSTEM built for the main program BMDSIM1. Some of the statements from the main program are shown in the figure. Specifically, the message that invokes the LNKCAL-DWELLT method is shown, and the message that creates C-22 is shown. Since C-22 must be passed to the LNKCAL-DWELLT method, it must first be created and inserted into the statements of the BMDSIM1 method. The variable C-22 (a computer-generated name) is assigned the object instance returned from the CREATE-CLASS-5 method shown in Figure 119. Note that the signature of the LNKCAL-DWELLT message is incomplete. If there are any other objects in the signature of the LNKCAL-DWELLT method, assignment statements are built and inserted into the statements of BMDSIM1.

5.8.1 Removing Duplicate Object Instances. Following the creation of object instances and the transformation of subprogram calls into messages, the next design update is to replace *duplicate* object instances. Duplicate object instances are defined as follows.

```
class CLASS-2 attributes AXMAJ, AXMIN, ANGFAC
  method CAPTURE ( C-2, BEAMRA ): real
    begin
    TILTFA := DSIN ( GET-ANGFAC ( C-2));
    RMAREA := GET-AXMAJ ( C-2) *
      GET-AXMIN ( C-2);
    EFAREA := RMAREA * TILTFA;
    BMAREA := BEAMRA * BEAMRA;
    if EFAREA <= BMAREA * 6.0d0
     then USED := EFAREA / BMAREA;
          CAPTURE := 1.0d0 - DEXP ( -USED)
     else CAPTURE := 1.0d0 endif
    end
  superclass USER-OBJECT

class CLASS-2 attributes AXMAJ, AXMIN, BEAMRA
  method CAPTURE ( C-2,  ANGFAC ): real
    begin
    TILTFA := DSIN ( ANGFAC);
    RMAREA := GET-AXMAJ ( C-2) *
      GET-AXMIN ( C-2);
    EFAREA := RMAREA * TILTFA;
    BMAREA := GET-BEAMRA ( C-2) * GET-BEAMRA ( C-2);
    if EFAREA <= BMAREA * 6.0d0
     then USED := EFAREA / BMAREA;
          CAPTURE := 1.0d0 - DEXP ( -USED)
     else CAPTURE := 1.0d0 endif
    end
  superclass USER-OBJECT
```

Figure 115 Duplicate classes built for CAPTURE

```
class CLASS-2 attributes AXMAJ, AXMIN
  method CAPTURE ( C-2, BEAMRA ): real
    begin
    TILTFA := DSIN ( ANGFAC);
    RMAREA := GET-AXMAJ ( C-2) *
      GET-AXMIN ( C-2);
    EFAREA := RMAREA * TILTFA;
    BMAREA := BEAMRA * BEAMRA;
    if EFAREA <= BMAREA * 6.0d0
     then USED := EFAREA / BMAREA;
          CAPTURE := 1.0d0 - DEXP ( -USED)
     else CAPTURE := 1.0d0 endif
    end
  superclass USER-OBJECT
```

Figure 116 Class to replace the duplicates

```
procedure LNKCAL-DWELLT ( ..., BETA, XLAMDA, DIAM, ... )
  begin
  ...
  end
```

Figure 117 Partial Signature of Subprogram LNKCAL-DWELLT

Definition V.6. *Separate object instances that are built from the* same *class using the* same *data items are* duplicate *object instances.*

Duplicate object instances which are identified in the main program are replaced with a single object instance built from the class. This is done to avoid duplicate copies of the instance attributes which may be updated by different methods. Having duplicate object instances would result in an inconsistent system state, so they are eliminated.

For example, Figure 121 shows the overall system class CLASS-SYSTEM built for the main program BMDSIM1. As shown in the figure, the three variables C-22, C-32, and C-36 are created before being passed as actual parameters in the LNKCAL-DWELLT message. These three variables are all instances of CLASS-5, which is also shown in the figure. These instances are built by using the class's creation method, CREATE-CLASS-5. All three objects are built from the data items BETA, XLAMDA, and DIAM, thus all three objects are *duplicate*

```
class CLASS-20 attributes MIRR, MIRF, SLACT, IENG, SLANG
  method LNKCAL-DWELLT ( ..., C-22, ..., )
    begin
    ...
    end
```

Figure 118 Partial Signature of Method LNKCAL-DWELLT

```
class CLASS-5 attributes BETA, XLAMDA, DIAM
  method CREATE-CLASS-5 ( A-BETA, A-XLAMDA, A-DIAM ): a CLASS-5
    begin
    INST-CLASS-5 := new ( CLASS-5);
    SET-BETA ( INST-CLASS-5, A-BETA);
    SET-XLAMDA ( INST-CLASS-5, A-XLAMDA);
    SET-DIAM ( INST-CLASS-5, A-DIAM);
    CREATE-CLASS-5 := INST-CLASS-5
    end
  ...
```

Figure 119 Partial Class Containing Attributes BETA, XLAMDA, and DIAM

object instances. Figure 122 shows how method BMDSIM1 is updated by replacing the duplicate object instances C-22, C-32, and C-36 with the single instance C-60. The C-60 instance is created using the CREATE-CLASS-5 and passed to the LNKCAL-DWELLT method in place of C-22, C-32, and C-36. The instance C-60 is passed as three actual parameters in order to properly replace the object instances C-22, C-32, and C-36 in the invocation of LNKCAL-DWELLT. This allows the LNKCAL-DWELLT method to remain unchanged and properly refer to the instance C-60 using its three formal parameters. Future research should explore the changes required to methods such as LNKCAL-DWELLT in order to refer to a single formal parameter when eliminating duplicate objects.

5.8.2 Merging Overlapping Classes. The second update done to the design when transforming the main program is to merge classes that *overlap*.

Definition V.7. *A class* overlaps *another class when an instance of each class is built using at least one common data item.*

```
class CLASS-SYSTEM attributes
  method BMDSIM1 ( )
    begin
    ...
    C-22 := CREATE-CLASS-5 ( BETA, XLAMDA, DIAM);
    ...
    LNKCAL-DWELLT ( C-46, ..., C-22, ... );
    ...
    end
```

Figure 120 Creating an Instance of CLASS-5

In this case, the overlapping classes are *merged* into a new class. The merging of two classes unions the attributes and operations of the classes into a new class. A single new instance that is built from this new class replaces the two separate instances. Throughout the entire design, any instances from the overlapping classes are replaced by an instance of the new class. As with duplicate object instances, overlapping classes are merged in order to avoid duplicating instance attributes which may be changed by different methods. Building object instances from overlapping classes would result in an inconsistent system state, so the classes are merged.

For example, Figure 123 shows overlapping classes CLASS-19 and CLASS-17 as well as the overall system class CLASS-SYSTEM built for the main program BMDSIM1. As shown in the figure, the instance C-56 is built as an instance of CLASS-19 using the data items DWELLT and NMIRL. The instance C-57 is built as an instance of CLASS-17 using the data item DWELLT. By Definition V.7, classes CLASS-19 and CLASS-17 overlap and are merged into one new class.

Figure 124 shows the result of merging CLASS-19 and CLASS-17 into the new class CLASS-20. The attributes of CLASS-20 are the union of the attributes of CLASS-19 and CLASS-17. The names of the attributes in CLASS-20 match the names of the data items in the main program. The attribute names are changed because the names of the attributes in overlapping classes do not always match.[1] As shown in the figure, CLASS-20 includes

[1]Recognizing differently-named attributes that represent the same data item is done using the transitive closure of the μ relation, as described in Section 6.8.2

```
class CLASS-5 attributes BETA, XLAMDA, DIAM
  method CREATE-CLASS-5
      ( A-BETA, A-XLAMDA, A-DIAM ): a CLASS-5
    begin
    INST-CLASS-5 := new ( CLASS-5);
    SET-BETA ( INST-CLASS-5, A-BETA);
    SET-XLAMDA ( INST-CLASS-5, A-XLAMDA);
    SET-DIAM ( INST-CLASS-5, A-DIAM);
    CREATE-CLASS-5 := INST-CLASS-5
    end

class CLASS-SYSTEM attributes
  method BMDSIM1 ( )
    begin
    ...
    C-22 := CREATE-CLASS-5 ( BETA, XLAMDA, DIAM);
    C-32 := CREATE-CLASS-5 ( BETA, XLAMDA, DIAM);
    C-36 := CREATE-CLASS-5 ( BETA, XLAMDA, DIAM);
    ...
    LNKCAL-DWELLT ( C-46, C-22, C-32, C-36, ... );
    ...
    end
```

Figure 121   Duplicate object instances

all of the methods from CLASS-19 and CLASS-17. The new instance C-61 is created using the creation method from CLASS-20 and passed to the LNKCAL-DWELLT message in place of C-56 and C-57. The design is updated by removing CLASS-19 and CLASS-17 and adding CLASS-20. All instances of the classes CLASS-19 and CLASS-17 are converted to instances of the class CLASS-20 throughout the entire design. In this way, the design is updated to remove the overlapping classes when transforming the main program.

5.9 Converting Category 4 Subprograms

Category 4 subprograms produce multiple outputs without calling other subprograms. By using *program slicing* [22, 76], a multiple-output subprogram is converted into independent program slices that each produce a single output. These program slices are used to build new imperative subprograms, so a Category 4 subprogram is *sliced* into

```
class CLASS-SYSTEM attributes
  method BMDSIM1 ( )
    begin
    ...
    C-60 := CREATE-CLASS-5 ( BETA, XLAMDA, DIAM);
    ...
    LNKCAL-DWELLT ( C-46, C-60, C-60, C-60, ... );
    ...
    end
```

Figure 122 Duplicates removed

multiple Category 2 subprograms. This program slicing process is discussed in detail in Section 5.11.

5.10 Converting Category 5 Subprograms

A Category 5 subprogram produces multiple outputs and calls other subprograms. As with Category 4 subprograms, *program slicing* is used to convert a Category 5 subprogram into multiple slices. These slices are used to build new imperative subprograms that may or may not call other subprograms, so the Category 5 subprograms are sliced into new subprograms that are classified as either Category 3 or Category 2 subprograms. This process of program slicing is discussed in detail in Section 5.11.

5.11 Program Slicing

This section defines how *program slicing* is used in the PBOI method to simplify the transformation task. Program slicing has already been defined in the literature [22,30,76]. In this research, a new imperative subprogram is built by providing an appropriate name and using a program slice as the sequence of statements in the subprogram. The sequence of formal parameters for this new subprogram consists of all of the original formal parameters that are referenced in the program slice.

By using program slicing in this way, the behavior of a Category 4 or Category 5 subprogram is *projected* into multiple subprograms. Each of the subprograms produces one

```
class CLASS-19 attributes A, NVAL
  method MAXA-NMAX ( C-45, NMAX, AMAX )
    begin
    ...
    end

class CLASS-17 attributes DWELLT
  method DASET-DWELLT ( C-43, DT, TLASE)
    begin
    ...
    end

class CLASS-SYSTEM attributes
  method BMDSIM1 ( )
    begin
    ...
    C-56 := CREATE-CLASS-19 ( DWELLT, NMIRL);
    C-57 := CREATE-CLASS-17 ( DWELLT );
    LNKCAL-DWELLT ( C-46, ..., C-56, C-57);
    ...
    end
```

Figure 123   Overlapping object classes

```
class CLASS-20 attributes NMIRL, DWELLT
  method CREATE-CLASS-20
      ( A-NMIRL, A-DWELLT ): a CLASS-20
    begin
    INST-CLASS-20 := new ( CLASS-20);
    SET-NMIRL ( INST-CLASS-20, A-NMIRL);
    SET-DWELLT ( INST-CLASS-20, A-DWELLT);
    CREATE-CLASS-20 := INST-CLASS-20
    end
  method MAXA-NMAX ( C-45, NMAX, AMAX )
    begin
    ...
    end
  method DASET-DWELLT ( C-43, DT, TLASE)
    begin
    ...
    end

class CLASS-SYSTEM attributes
  method BMDSIM1 ( )
    begin
    ...
    C-61 := CREATE-CLASS-20 ( NMIRL, DWELLT);
    LNKCAL-DWELLT ( C-46, ..., C-61, C-61);
    ...
    end
```

Figure 124 Overlapping classes merged

of the data items originally produced by the Category 4 or Category 5 subprogram. Since the PBOI method relies on the parameters being passed from subprogram to subprogram and the new subprograms include only those parameters required to produce the data item, program slicing has grouped these parameters according to how they are used instead how they are organized. This new grouping helps the PBOI method move away from the *structured* analysis and design that was done on the original system (if any) and move towards an object-oriented design.

For example, Figure 125 shows the imperative subprogram BOOSTR using GIL syntax. This is a Category 4 subprogram because it produces the two data items R and V and calls no other (user-defined) subprograms. Figure 126 shows the new imperative subprogram

```
procedure BOOSTR ( ITYPE, T, R, V )
  begin
  RE := 6378.16d0;
  PI2 := 1.570796327d0;
  ITYP := ABS ( ITYPE);
  if ITYP = 1
    then ALT := -5.71d0 + (0.24d0 + 0.00363d0 * T) * T;
      RANG := -0.172d0 + (-0.419d0 + 0.0112d0 * T) * T;
      VEL := -0.196d0 + (0.0236d0 + 1.6d-6 * T) * T;
      GAMA := 1.12d0 + (-0.00751d0 + 1.82d-5 * T) * T
    else endif;
  if ALT < 0 then ALT := 0 else endif;
  if RANG < 0 then RANG := 0 else endif;
  if GAMA > PI2 then GAMA := PI2 else endif;
  RM := RE + ALT;
  THETA := RANG / RE;
  R ( 1) := RM * DCOS ( THETA);
  R ( 2) := RM * DSIN ( THETA);
  R ( 3) := 0;
  BETA := THETA + PI2 - GAMA;
  V ( 1) := VEL * DCOS ( BETA);
  V ( 2) := VEL * DSIN ( BETA);
  V ( 3) := 0
  end
```

Figure 125 Imperative subprogram BOOSTR

built from the program slice for the data item R. The name of the subprogram is built by appending the data item identifier onto the name of the original subprogram, hence the name BOOSTR-R. The formal parameters of the new subprogram are the data items ITYPE, T, and R. The new subprogram is classified as a Category 2 subprogram because it produces the single data item R and calls no other subprograms. Figure 127 shows the new imperative subprogram built from the program slice for the data item V. This new subprogram is classified as a Category 2 subprogram because it produces the single data item V and calls no other subprograms.

```
procedure BOOSTR-R ( ITYPE, T, R ) begin
  RE := 6378.16d0;
  ITYP := ABS ( ITYPE);
  if ITYP = 1
    then ALT := -5.71d0 + (0.24d0 + 0.00363d0 * T) * T;
      RANG := -0.172d0 + (-0.419d0 + 0.0112d0 * T) * T
    else endif;
  if ALT < 0 then ALT := 0 else endif;
  if RANG < 0 then RANG := 0 else endif;
  RM := RE + ALT;
  THETA := RANG / RE;
  R ( 1) := RM * DCOS ( THETA);
  R ( 2) := RM * DSIN ( THETA);
  R ( 3) := 0
  end
```

Figure 126 Imperative subprogram BOOSTR-R

5.12 Inter-Procedural Slicing

When a subprogram being sliced includes calls to other subprograms, a more robust program slicing process is required. *Inter-procedural slicing* [30] is a special form of program slicing that builds a program slice from a subprogram taking into consideration the calls being made to other subprograms. Only the calls to subprograms that are producing data items required for the slice are included in the program slice.

This research uses inter-procedural slicing when slicing Category 5 subprograms. A new subprogram is built from the program slice generated, as explained in Section 5.11. For example, Figure 128 shows the Category 5 subprogram TRAJ, which includes calls to the BOOSTR-R subprogram and BOOSTR-V subprogram. Originally, the TRAJ subprogram called the BOOSTR subprogram. Because BOOSTR is a Category 4 subprogram it has *already* been sliced and the calls to the new subprograms have replaced the call to BOOSTR. In this research, any calls in a Category 5 subprogram that sliced using inter-procedural slicing must produce a single data item, i.e. calls to Category 2 or Category 3 subprograms. An overall process that meets this constraint is used to convert Category 4 and Category 5 subprograms as explained in Section 5.15. Since the TRAJ subprogram includes the calls to BOOSTR-R and BOOSTR-V, *inter-procedural slicing* must be used. For example, Figure 129

```
procedure BOOSTR-V ( ITYPE, T, V ) begin
  RE := 6378.16d0;
  PI2 := 1.570796327d0;
  ITYP := ABS ( ITYPE);
  if ITYP = 1
    then RANG := -0.172d0 + (-0.419d0 + 0.0112d0 * T) * T;
      VEL := -0.196d0 + (0.0236d0 + 1.6d-6 * T) * T;
      GAMA := 1.12d0 + (-0.00751d0 + 1.82d-5 * T) * T
    else endif;
  if RANG < 0 then RANG := 0 else endif;
  if GAMA > PI2 then GAMA := PI2 else endif;
  THETA := RANG / RE;
  BETA := THETA + PI2 - GAMA;
  V ( 1) := VEL * DCOS ( BETA);
  V ( 2) := VEL * DSIN ( BETA);
  V ( 3) := 0
  end
```

Figure 127   Imperative subprogram BOOSTR-V

shows the result of inter-procedural slicing the TRAJ subprogram for the data item R. The TRAJ-R subprogram includes a call to the BOOSTR-R subprogram, but not a call to the BOOSTR-V subprogram. This is because the data item produced by the BOOSTR-V subprogram is not needed for the slice on data item R in TRAJ-R. Figure 130 shows the result of inter-procedural slicing the TRAJ subprogram for the data item V. The TRAJ-V subprogram includes a call to the BOOSTR-V subprogram but not the BOOSTR-R subprogram.

5.13 Masking Output Parameters

Program slicing does not always convert a Category 4 or Category 5 subprogram into a Category 2 or Category 3 subprogram. In some cases, one output data item must be *defined* in order to produce a second output data item. Thus, if both data items were originally produced by the subprogram, then the new subprogram built from a program slice on the second output data item will still include the first output data item and will be classified as either a Category 4 or Category 5 subprogram.

For example, Figure 131 shows the Category 4 subprogram LASP. This subprogram

```
procedure TRAJ
  ( ITYPE, T, VI, RBO, VBO, TBO, TFBOT, AR, R, V )
  begin
  XMU := 398601.2d0;
  WDOT := 7.292115600000001d-5;
  if T < TBO
    then BOOSTR-R ( ITYPE, T, RP);
      BOOSTR-V ( ITYPE, T, VP);
      MTRT ( RP, AR, RO);
      MTRT ( VP, AR, VT);
      TEMP-40 := 1;
      VADD ( TEMP-40, VT, VI, VO)
    else endif;
  DLONT := -WDOT * T;
  TEMP-41 := 1;
  MAT ( TEMP-41, DLONT, ARL);
  MTRT ( RO, ARL, R);
  MTRT ( VO, ARL, V)
  end
```

Figure 128 Original imperative subprogram TRAJ

produces the data items BFLUX, BFLU, and TD. Figure 132 shows the new subprograms LASP-BFLUX, LASP-BFLU, and LASP-TD, built from the program slices for each of these three data items. Note that the LASP-TD subprogram still includes definitions of the BFLUX and BFLU output data items. In fact, there is no difference between the LASP subprogram and the LASP-TD subprogram. The new subprogram is still classified as a Category 4 subprogram and has not been converted to a Category 2 subprogram.

To solve this problem, certain data items in the new subprogram can be *masked* so that they are not produced by the subprogram. To mask a data item, a new local variable is created and any references to the data item are replaced with references to the local variable. Because of Restriction III.1, the data items being masked cannot be both input and output parameters. This means there is no need to initialize the local variable used to mask a data item. The definition of the data item becomes a definition of the local variable, thus the data item is no longer produced by the new subprogram. Figure 133 shows the LASP-TD subprogram after the BFLUX and BFLU data items have been masked.

```
procedure TRAJ-R
  ( ITYPE, T, RBO, VBO, TBO, TFBOT, AR, R )
  begin
  XMU := 398601.2d0;
  WDOT := 7.292115600000001d-5;
  if T < TBO
    then BOOSTR-R ( ITYPE, T, RP);
      MTRT ( RP, AR, RO)
    else endif;
  DLONT := -WDOT * T;
  TEMP-41 := 1;
  MAT ( TEMP-41, DLONT, ARL);
  MTRT ( RO, ARL, R)
  end
```

Figure 129 Imperative subprogram TRAJ-R

These data items still appear in the signature of the LASP-TD subprogram in order to localize the masking process. The identifiers that are used for the formal parameters must not match the identifiers used for the local variables in order to created the disconnect that masks the output parameters.

5.14 Conservative Slicing

By including all statements from a subprogram that are needed to produce a data item, statements that appear to be unnecessary appear in the program slice. For example, Figure 134 shows the program slice built for the data item R from the subprogram KEP. The second while loop in the subprogram is unnecessary code, but appears in the program slice because definitions of the data item I are required to produce R. Extra statements such as this while loop are included in the slice because this research is using a *conservative* approach to program slicing. There is no attempt in the research to optimize the slices or filter unnecessary statements.

```
procedure TRAJ-V
  ( ITYPE, T, VI, RBO, VBO, TBO, TFBOT, AR, V )
  begin
  XMU := 398601.2d0;
  WDOT := 7.292115600000001d-5;
  if T < TBO
    then BOOSTR-V ( ITYPE, T, VP);
      MTRT ( VP, AR, VT);
      TEMP-40 := 1;
      VADD ( TEMP-40, VT, VI, VO)
    else endif;
  DLONT := -WDOT * T;
  TEMP-41 := 1;
  MAT ( TEMP-41, DLONT, ARL);
  MTRT ( VO, ARL, V)
  end
```

Figure 130 Imperative subprogram TRAJ-V

```
procedure LASP
  ( POWL, EFFNCY, AREA, CFLUM, CFLUS, XKF, BFLUX, BFLU, TD )
  begin
  BFLUX := EFFNCY * POWL / AREA;
  BFLU := CFLUM + XKF * CFLUS;
  TD := BFLU / BFLUX
  end
```

Figure 131 Imperative subprogram LASP

5.15 Slicing for the Main Program

This section defines the overall process used to ensure all Category 4 and Category 5 subprograms are converted to Category 2 or Category 3 subprograms. The overall collection of imperative subprograms to be converted to the object-oriented paradigm is stored in a *structured design*, as described in Section 3.7.

The first step in this process is to build a program slice for each of the data items produced by a Category 4 or Category 5 subprogram. Inter-procedural slicing is *not* used at this point. A new subprogram is built for each program slice and added to the structured design. Next, the original subprogram calls in all Category 3, Category 5, and Category 1

```
procedure LASP-BFLUX ( POWL, EFFNCY, AREA, BFLUX ) begin
  BFLUX := EFFNCY * POWL / AREA end

procedure LASP-BFLU ( CFLUM, CFLUS, XKF, BFLU ) begin
  BFLU := CFLUM + XKF * CFLUS end

procedure LASP-TD
  ( POWL, EFFNCY, AREA, CFLUM, CFLUS, XKF, BFLUX, BFLU, TD )
  begin
  BFLUX := EFFNCY * POWL / AREA;
  BFLU := CFLUM + XKF * CFLUS;
  TD := BFLU / BFLUX
  end
```

Figure 132   Imperative subprogram LASP-TD

```
procedure LASP-TD
  ( POWL, EFFNCY, AREA, CFLUM, CFLUS, XKF, BFLUX, BFLU, TD )
  begin
  LOCAL-1 := EFFNCY * POWL / AREA;
  LOCAL-2 := CFLUM + XKF * CFLUS;
  TD := LOCAL-2 / LOCAL-1
  end
```

Figure 133   BFLUX and BFLU masked

subprograms other than the main program are replaced by calls to the new subprograms just created. This means all the original Category 4 and Category 5 subprograms are still in the structured design, but the only calls to such subprograms are in the main program.

At this point, program slices are built using inter-procedural slicing for each data item produced by a subprogram that is called from the main program. New subprograms are built from these program slices. Each call to the original subprogram is replaced by calls to the new subprograms. Note that calls to Category 0 and Category 1 subprograms are not changed. The calls to Category 2 and Category 3 subprograms are also not changed because a subprogram built from the program slice of a Category 2 or Category 3 subprogram is identical to the original subprogram.

```
procedure KEP-R ( RO, VO, T, XMU, R )
  begin
  I := 1;
  while I <= 3 do
    begin
    R ( I) := RO ( I) + VO ( I);
    I := I + 1
    end;
  I := 1;
  while I <= 3 do
    begin
    I := I + 1
    end
  end
```

Figure 134 Imperative subprogram KEP-R

However, it is important to use inter-procedural slicing for Category 3 subprograms because they are allowed to call subprograms from any other category. The first two steps presented above ensure the calls to Category 4 or Category 5 subprograms have already been converted into calls to new subprograms. Inter-procedural slicing ensures that only the calls to subprograms that produce data items required for the program slice are included in the program slice.

For example, Figure 135 shows an alternate version of the `TRAJ` subprogram that is classified as a Category 3 subprogram producing the single data item `R`. By using inter-procedural slicing on this Category 3 subprogram, the subprogram calls not producing data items for the program slice are eliminated. Figure 136 shows the updated `TRAJ` subprogram. Note the call to `BOOSTR-R` and two calls to `MTRT` have been eliminated. Subprogram calls such as these constitute *dead code* [22] and are eliminated by inter-procedural slicing.

In the main program, the calls to Category 4 and Category 5 subprograms are replaced with calls to the new subprograms built from program slices. In this way, none of the Category 4 or Category 5 subprograms are called by any of the subprograms in the design. Furthermore, by using inter-procedural slicing, *all* of the subprograms called

```
procedure TRAJ
  ( ITYPE, T, VI, RBO, VBO, TBO, TFBOT, AR, R )
  begin
  XMU := 398601.2d0;
  WDOT := 7.292115600000001d-5;
  if T < TBO
    then BOOSTR-R ( ITYPE, T, RP);
         BOOSTR-V ( ITYPE, T, VP);
      MTRT ( RP, AR, RO);
      MTRT ( VP, AR, VT);
      TEMP-40 := 1;
      VADD ( TEMP-40, VT, VI, VO)
    else endif;
  DLONT := -WDOT * T;
  TEMP-41 := 1;
  MAT ( TEMP-41, DLONT, ARL);
  MTRT ( RO, ARL, R);
  MTRT ( VO, ARL, V)
  end
```

Figure 135 TRAJ as a Category 3 subprogram

by the main program produce exactly one data item. Several transformations have been defined that formalize this conversion process. Chapter VI presents these transformations.

5.16 Discussion

The conversion of imperative subprograms to the object paradigm using the PBOI methodology results in a rudimentary object-oriented design. The inheritance associations are rudimentary because the inheritance built during the conversion is a flat hierarchy where every class built in the object model inherits from the overall super class USER-OBJECT. The aggregation associations are also rudimentary because, while allowed, aggregation associations are not identified by the PBOI methodology. This discussion leads to the following limitations on the conversions from the imperative subprograms to the object paradigm.

Limitation V.1. *Every class in the design extracted using the PBOI methodology inherits from the overall super-class* USER-OBJECT.

```
procedure TRAJ
  ( ITYPE, T, RBO, VBO, TBO, TFBOT, AR, R )
  begin
  XMU := 398601.2d0;
  WDOT := 7.292115600000001d-5;
  if T < TBO
    then BOOSTR-R ( ITYPE, T, RP);
      MTRT ( RP, AR, RO)
    else endif;
  DLONT := -WDOT * T;
  TEMP-41 := 1;
  MAT ( TEMP-41, DLONT, ARL);
  MTRT ( RO, ARL, R)
  end
```

Figure 136 Updated Category 3 subprogram TRAJ

Limitation V.2. *Aggregation associations are not identified by the PBOI methodology.*

These two limitations are not unreasonable. Future research could possibly remove these limitations by expanding the PBOI methodology.

5.17 Summary

This chapter has provided an informal description of how imperative subprograms are transformed to the object-oriented paradigm along with some rationale for the approach. The Parameter-Based Object Identification (PBOI) method for extracting objects from imperative subprograms was presented. The taxonomy of imperative subprograms used to simplify the development of the PBOI method was also presented. Each category of imperative subprogram was explained and the PBOI transformation for converting that category of subprogram to the object-oriented paradigm was defined. The process of converting Category 4 and Category 5 subprograms to Category 2 and Category 3 subprograms by using program slicing and inter-procedural slicing was also explained. Transformations that formalize the PBOI method have been defined and are presented in the following chapter.

VI. Formal Transformations

6.1 Introduction

This chapter presents transformations that formalize the PBOI methodology, which was presented in Chapter V. Proof that these transformations maintain functional equivalence is presented in Chapter VII. Table 1 shows the order in which the transformations are presented in the chapter. The transformations that convert statements and expressions are presented first, since the transformations for subprograms build on them. Transformations for each category of subprogram are presented followed by the formalized transformations for program slicing. Finally, the transformations for the main program are presented along with the transformation for the entire imperative design.

In order to formally classify a subprogram S, let $Cat_i(S)$ indicate the category to which a subprogram belongs, such that $Cat_i(S)$ is true when S is a Category i subprogram. Let $Cat_M(S)$ indicate that the subprogram S is a Category 1 subprogram that is also the *main program.*

6.2 Transforming Statements (θ)

This section defines the formal transformation for converting GIM statements to GOM statements. This transformation is used when converting a subprogram to the object paradigm. Recall from Section 3.24 that a GIM subprogram is defined by the following tuple.

$$< id_S, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma >$$

The GOM entity that is analogous to a GIM subprogram is the *method.* In order to convert the GIM subprogram to the object paradigm, the GIM subprogram is converted to a GOM method, which is represented as a tuple of the following form.

$$< id_S, Q_{tar}, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi >$$

Transformation	Constructs
θ_v θ_V	variables
θ_e θ_E	expressions
θ	statements
δ_e	accesses in expressions
δ	variable accesses
δ_e^{-1}	accesses in expressions
δ^{-1}	attribute accesses
σ	subprograms
σ_2	Category 2 subprograms
T_A^{δ}	parameters to attributes
$T_A^{\delta^{-1}}$	attributes to parameters
γ	attribute accesses
T_A^{γ}	attributes
$T_A^{\gamma'}$	known attributes
T_{pboi}^1 T_{pboi}^2 T_{pboi}^3 T_{pboi}^4	subprograms
T_{pboi}	subprograms
v_m	subprogram calls
v_e	calls in expressions
v	calls in statements
T_l^v	calls in subprograms
T_o^v	messages in methods
T_{class}^{dup}	duplicate classes
σ_3	Category 3 subprograms
σ_1	Category 1 subprograms
T_{obj}^{dup}	duplicate objects
κ κ^{-1}	"GET-", "SET-", and "CREATE-" methods
T_{class}^{merge}	classes
T_m^{as} v_e^{as}	assignment statements
v_M	main program messages
T_M^v	main program messages
σ_M	main program
T_{ps}^{build} T_{ps}^{calls}	program slices
T_{ps}^{main}	main program slice calls
σ_F	imperative design

Table 1 Formal transformations

As an imperative subprogram is converted to an object-oriented method, each imperative statement in Σ must be converted to an object-oriented statement in Ψ. One difference between an imperative statement and an object-oriented statement is the way these statements access data items. In an imperative statement, data items are stored in variables including local variables and parameters. These variables are accessed by using the name of the variable. In object-oriented statements, data items may be stored in attributes of objects. These data items must be accessed either directly by using the name of the attribute and the name of the object or indirectly by sending a message to the appropriate object. Because of this difference, each GIM statement must be converted to a GOM statement.

Part of the conversion process is to transform each GIM variable into a GOM variable. Let θ_v be the transformation that converts a GIM variable into a GOM variable. Let v be a GIM variable and let v' be a GOM variable. In order to formalize the conversion, a notation is presented here that denotes the class of AST that represents the GIM entity and the GOM entity. For example, GOM variables are represented using the `gom-variable` class. This is formalized by using a boolean function named `gom-variable` that returns true if the entity is a `gom-variable` and false otherwise. The θ_v transformation is defined below.

$$\begin{aligned}
\theta_v(v) &= v' \text{ where} \\
&\quad \text{gim-variable}(v) \textit{ and } \text{gom-variable}(v') \textit{ and} \\
&\quad \forall v_1,\ v_2 \\
&\qquad \theta_v(v_1) = \theta_v(v_2) \Rightarrow v_1 = v_2
\end{aligned}$$

Since GIM input statements include sequences of variables, the following transformation of sequences of variables is presented. Let V represent a sequence of GIM variables and let V' represent a sequence of GOM variables. Let $pos(v, V)$ represent the ordinal position of the variable v in the sequence V. The θ_V transformation converts V to V' as defined below.

$$\begin{aligned}
\theta_V(V) &= V' \text{ where} \\
&\forall\, v \in V \\
&\quad v' = \theta_v(v) \textit{ and} \\
&\quad v' \in V' \textit{ and} \\
&\quad pos(v, V) = pos(v', V')
\end{aligned}$$

Another part of the conversion process is to transform each of the GIM expressions into GOM expressions. Let θ_e be the transformation that converts a GIM expression into a GOM expression. Let e be a GIM expression and let e' be a GOM expression. The θ_e transformation is defined as shown in Figure 137.

GIM output statements include a sequence of expressions. In order to transform such sequences, let θ_E represent a formal transformation of a sequence of expressions. Let E be a sequence of expressions. Let $pos(e, E)$ represent the ordinal position of a expression e in the sequence of expressions E. The transformation is defined below.

$$\begin{aligned}
\theta_E(E) &= E' \text{ where} \\
&\forall\, e \in E \\
&\quad e' = \theta_e(e) \textit{ and} \\
&\quad e' \in E' \textit{ and} \\
&\quad pos(e, E) = pos(e', E')
\end{aligned}$$

Let θ be a transformation that converts a sequence of imperative statements into a sequence of object-oriented and imperative statements. Let Σ represent a sequence of imperative statements and let Ψ' represent a sequence of object-oriented and imperative statements. Let $pos(s, \Sigma)$ represent the ordinal position of a statement s in the sequence Σ. The θ transformation uses the θ_v, θ_e, θ_V, and θ_E transformations and is defined as shown in Figure 138. GIM procedure call statements are not transformed to object-oriented statements by the θ transformation. GIM function calls in expressions are not transformed by the θ_e transformation. This explains why Ψ' is a mixture of imperative statements and object-oriented statements.

$\theta_e(e) \ = \ e'$ where

imperative-variable(e) $\Rightarrow e' \ = \ \theta_v(e)$ *and*

imperative-function-call(e) $\Rightarrow e' \ = \ e$ *and*

imperative-literal-boolean(e) $\Rightarrow$ gom-literal-boolean(e') *and* $e \ = \ e'$ *and*

imperative-literal-integer(e) $\Rightarrow$ gom-literal-integer(e') *and* $e \ = \ e'$ *and*

imperative-literal-real(e) $\Rightarrow$ gom-literal-real(e') *and* $e \ = \ e'$ *and*

imperative-literal-string(e) $\Rightarrow$ gom-literal-string(e') *and* $e \ = \ e'$ *and*

imperative-addition(e) *and* $e \ = < e_1, \ +, e_2 > \Rightarrow$

  gom-addition(e') *and* $e' \ = < \theta_e(e_1), \ +, \theta_e(e_2) >$ *and*

imperative-and(e) *and* $e \ = < e_1, \ \text{and}, e_2 > \Rightarrow$

  gom-and(e') *and* $e' \ = < \theta_e(e_1), \ \text{and}, \theta_e(e_2) >$ *and*

imperative-concat(e) *and* $e \ = < e_1, \ \&, e_2 > \Rightarrow$

  gom-concat(e') *and* $e' \ = < \theta_e(e_1), \ \&, \theta_e(e_2) >$ *and*

imperative-division(e) *and* $e \ = < e_1, \ /, e_2 > \Rightarrow$

  gom-division(e') *and* $e' \ = < \theta_e(e_1), \ /, \theta_e(e_2) >$ *and*

imperative-equal(e) *and* $e \ = < e_1, \ =, e_2 > \Rightarrow$

  gom-equal(e') *and* $e' \ = < \theta_e(e_1), \ =, \theta_e(e_2) >$ *and*

imperative-exponent(e) *and* $e \ = < e_1, \ **, e_2 > \Rightarrow$

  gom-exponent(e') *and* $e' \ = < \theta_e(e_1), \ **, \theta_e(e_2) >$ *and*

imperative-greater-than-or-equal(e) *and* $e \ = < e_1, >=, e_2 > \Rightarrow$

  gom-greater-than-or-equal(e') *and* $e' \ = < \theta_e(e_1), >=, \theta_e(e_2) >$ *and*

imperative-greater-than(e) *and* $e \ = < e_1, >, e_2 > \Rightarrow$

  gom-greater-than(e') *and* $e' \ = < \theta_e(e_1), >, \theta_e(e_2) >$ *and*

imperative-less-than-or-equal(e) *and* $e \ = < e_1, <=, e_2 > \Rightarrow$

  gom-less-than-or-equal(e') *and* $e' \ = < \theta_e(e_1), <=, \theta_e(e_2) >$ *and*

imperative-less-than(e) *and* $e \ = < e_1, <, e_2 > \Rightarrow$

  gom-less-than(e') *and* $e' \ = < \theta_e(e_1), <, \theta_e(e_2) >$ *and*

imperative-multiplication(e) *and* $e \ = < e_1, \ *, e_2 > \Rightarrow$

  gom-multiplication(e') *and* $e' \ = < \theta_e(e_1), \ *, \theta_e(e_2) >$ *and*

imperative-not-equal(e) *and* $e \ = < e_1, <>, e_2 > \Rightarrow$

  gom-not-equal(e') *and* $e' \ = < \theta_e(e_1), <>, \theta_e(e_2) >$ *and*

imperative-or(e) *and* $e \ = < e_1, \ \text{or}, e_2 > \Rightarrow$

  gom-or(e') *and* $e' \ = < \theta_e(e_1), \ \text{or}, \theta_e(e_2) >$ *and*

imperative-subtraction(e) *and* $e \ = < e_1, -, e_2 > \Rightarrow$

  gom-subtraction(e') *and* $e' \ = < \theta_e(e_1), -, \theta_e(e_2) >$ *and*

imperative-negate(e) *and* $e \ = < -, e_1 > \Rightarrow$

  gom-negate(e') *and* $e' \ = < -, \theta_e(e_1) >$ *and*

imperative-not(e) *and* $e \ = < \ \text{not}, e_1 > \Rightarrow$

  gom-not(e') *and* $e' \ = < \ \text{not}, \theta_e(e_1) >$

Figure 137 The θ_e transformation

$\theta(\Sigma) = \bar{\Psi}$ where
  For each $s \in \Sigma$
    imperative-procedure-call(s) $\Rightarrow s' = s$ *and*
    $s = <x,\ :=, e> \Rightarrow$
      $s' = <\theta_v(x),\ :=, \theta_e(e)>$ *and*
    $s = <\text{if}, e,\ \text{then}, S_1,\ \text{else}, S_2> \Rightarrow$
      $s' = <\text{if}, \theta_e(e),\ \text{then}, \theta(S_1),\ \text{else}, \theta(S_2)>$ *and*
    $s = <\text{while}, e, S_3> \Rightarrow s' = <\text{while}, \theta_e(e), \theta(S_3)>$ *and*
    $s = <\text{input},\ \text{iport}, V> \Rightarrow s' = <\text{input},\ \text{iport}, \theta_V(V)>$ *and*
    $s = <\text{output},\ \text{oport}, E> \Rightarrow s' = <\text{output},\ \text{oport}, \theta_E(E)>$ *and*
    $s' \in \Psi'$ *and*
    $pos(s, \Sigma) = pos(s', \bar{\Psi})$

Figure 138 The θ transformation

6.3 Transforming Accesses (δ)

As discussed in Section 5.3, when converting a GIM subprogram to the GOM using the PBOI method, certain variables in the subprogram are converted to attributes of the class built for the subprogram. After a variable has been converted to an attribute, the statements in the GOM method must be updated to access the data item from the object attribute instead of the variable. The formal transformation for converting variable accesses in GOM statements to attribute accesses of an object is defined in this section.

6.3.1 Transforming Expressions (δ_e). As part of the conversion of variable accesses to attribute accesses, it is necessary to transform the GOM expressions found in GOM statements. This section discusses a formal transformation of GOM expressions that replaces references to variables with accesses of attributes.

Let δ_e be a transformation that converts an expression with variable accesses into an expression with object attribute accesses. Let e be a GOM expression, let c be an instance of a class C, and let A be a set of GOM variables. Let `GET-a` represent the symbol resulting from prepending "GET-" to the GOM variable a. The δ_e transformation is defined as shown in Figure 139. This transformation replaces all occurrences of a in the

$\delta_e(e, c, A) \; = \; e'$ where

$\text{gom-variable}(e) \; \textit{and} \; e \; = \; a \; \textit{and} \; a \; \in A \; \Rightarrow e' \; = < \; \text{GET-a}, [c] >$

$\text{gom-variable}(e) \; \textit{and} \; e \; = \; a \; \textit{and} \; a \; \notin A \; \Rightarrow e' \; = \; e \; \textit{and}$

$e \; = < \; \text{GET-a}, [c] > \Rightarrow e' \; = \; e \; \textit{and}$

$\text{gom-function-call}(e) \Rightarrow e' \; = \; e \; \textit{and}$

$\text{gom-literal-boolean}(e) \Rightarrow e' \; = \; e \; \textit{and}$

$\text{gom-literal-integer}(e) \Rightarrow e' \; = \; e \; \textit{and}$

$\text{gom-literal-real}(e) \Rightarrow e' \; = \; e \; \textit{and}$

$\text{gom-literal-string}(e) \Rightarrow e' \; = \; e \; \textit{and}$

$e \; = < e_1, \; +, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), \; +, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, \; \text{and}, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), \; \text{and}, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, \; \&, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), \; \&, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, \; /, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), \; /, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, \; =, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), \; =, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, \; **, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), \; **, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, >=, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), >=, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, >, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), >, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, <=, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), <=, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, <, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), <, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, \; *, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), \; *, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, <>, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), <>, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, \; \text{or}, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), \; \text{or}, \delta_e(e_2) > \; \textit{and}$

$e \; = < e_1, -, e_2 > \Rightarrow \; e' \; = < \delta_e(e_1), -, \delta_e(e_2) > \; \textit{and}$

$e \; = < -, e_1 > \Rightarrow \; e' \; = < -, \delta_e(e_1) > \; \textit{and}$

$e \; = < \; \text{not}, e_1 > \Rightarrow \; e' \; = < \; \text{not}, \delta_e(e_1) >$

Figure 139 The δ_e transformation

expression e with GOM "GET-" messages that accesses the attribute a from the object instance c. If none of the GOM variables in the set A are part of the expression e or the set A is empty, then the original expression e is returned. For example, the following expression (shown in GOL syntax) shows a GOM multiplication expression with the XLAMDA and ZCOS variables.

```
XLAMDA * ZCOS * ZCOS * ZCOS
```

Let e represent this expression and let C-18 be an object instance variable. Consider the following transformation.

$$\delta_e(e,\ \text{C-18}, \{\ \text{ZCOS}\ \})$$

This transformation replaces all variable accesses of ZCOS in e with messages that access the attribute ZCOS from C-18. The following expression shows the result of this transformation.

```
XLAMDA * GET-ZCOS ( C-18) * GET-ZCOS ( C-18) * GET-ZCOS ( C-18)
```

All accesses to the data item ZCOS are now "GET-" messages that access the attribute from C-18.

6.3.2 Transforming GOM Variable Accesses. Now that accesses to variables in expressions can be transformed, a formal transformation is presented that converts a sequence of statements that include variable accesses into a sequence of statements that include attribute accesses. Let δ be a transformation that converts GOM variable accesses in sequences of statements to GOM object attribute accesses. Let Ψ be a sequence of GOM statements, let c be an instance of a class C and A be a set of GOM variables. Let SET-a represent the symbol resulting from prepending "SET-" to the name of the GOM variable a. Recall that the tuple $<$ SET-a, $[c, e]$ $>$ represents a message named SET-a with two parameters, c and e. Let $instance(c, C)$ represent whether or not an object c is an instance of class C. The δ transformation is defined as shown in Figure 140. This is a recursive transformation that uses the structure of the GOM statements to transform each access to the GOM variables a in A to GOM messages that access a as an attribute. If the statement in Ψ is an assignment to a, then the entire statement is transformed to a

```
$\delta(\Psi, c, A) \;=\; \Psi'$ $where$
    $instance(c, C)$ $and$
    $C \;= <id_C, \Phi_C, \Omega_C, \lambda>$
    $\forall\, a \in A$ $and$ $\forall\, s \in \Psi$
        $a \in \Phi_C$ $and$
        $s \;= <a,\; :=, e> \Rightarrow$
            $s' \;= <$ SET-a, $[c, \delta_e(e, c, A)]>$ $and$
        $s \;= <x,\; :=, e>$ $and$ $x \notin A \;\Rightarrow$
            $s' \;= <x,\; :=, \delta_e(e, c, A)>$ $and$
        $s \;= <$ if$, e,$ then$, S_1,$ else$, S_2> \Rightarrow$
            $s' \;= <$ if$, \delta_e(e, c, A),$ then$, \delta(S_1, c, A),$ else$, \delta(S_2, c, A)>$ $and$
        $s \;= <$ while$, e, S_3> \Rightarrow$
            $s' \;= <$ while$, \delta_e(e, c, A), \delta(S_3, c, A)>$ $and$
        $s \;= <$ output, oport$, E> \Rightarrow$
            $s' \;= <$ output, oport$, E'>$ $and$
                $E' \;=\; [\delta_e(e, c, A) \mid e \in E]$ $and$
        $s' \;\in \Psi'$
```

Figure 140   The δ transformation

GOM "SET-" message. The expression e in the assignment statement is transformed using the δ_e transformation in order to transform any accesses to a in e. If the statement is an assignment to a variable other than a, the expression e must be transformed using the δ_e transformation. If the statement is a selection statement, the expression e is transformed by the δ_e transformation and the sequences of statements S_1 and S_2 are transformed by using the δ transformation recursively. If the statement is an iteration statement, the expression e is transformed using the δ_e transformation and the sequence of statements S_3 is transformed by using the δ transformation recursively. If the statement is an input or an output statement, then the sequence of expressions, E, in the statement is transformed by using the δ_e transformation on each of the expressions in the sequence. If none of the GOM variables in the set A are accessed in the statements in Ψ or the set A is empty, then the original statements in Ψ are returned unchanged.

6.3.3 Transforming Attribute Accesses (δ^{-1}). As part of the PBOI method, in some cases it is necessary to convert an attribute of a class back into a formal parameter of a method. In this case, the attribute accesses must be converted back to accesses of a variable since a formal parameter in a method is a variable defined by the method. This means the δ and δ_e transformations must be reversible.

Let δ_e^{-1} be a transformation that converts object attribute accesses in GOM expressions to variable accesses. Let e be a GOM expression, let c be an instance of a class C, and let A be a set of GOM variables. Let `GET-a` represent the symbol resulting from prepending "GET-" to the GOM variable a. The δ_e^{-1} transformation is defined as shown in Figure 141. This transformation replaces all occurrences of GOM "GET-" messages that access the attribute a in the expression with a reference to a. If none of the GOM variables in the set A are part of the expression e or the set A is empty, then the original expression e is returned.

The inverse of the transformation of statements is defined below. Let δ^{-1} be a transformation that converts GOM object attribute accesses in statements to GOM variable accesses. Let Ψ be a sequence of GOM statements, let c be an instance of a class C and A be a set of GOM variables. Let `SET-a` represent the symbol resulting from prepending "SET-" to the name of the GOM variable a. The δ^{-1} transformation is defined as shown in Figure 142. This is a recursive transformation that uses the structure of the GOM statements to transform each attribute access of a into a reference to the variable a. If the statement in Ψ is a "SET-" message that sets the value of a, then the "SET-" message is replaced with an assignment statement that sets the value of the variable a. The expression e from the message is transformed using the δ_e^{-1} transformation in order to transform any accesses of a. If the statement is an assignment to a variable other than a, the expression e must be transformed using the δ_e^{-1} transformation. If the statement is a selection statement, the expression e is transformed by the δ_e^{-1} transformation and the sequences of statements S_1 and S_2 are transformed by using the δ^{-1} transformation recursively. If the statement is an iteration statement, the expression e is transformed using the δ_e^{-1} transformation and the sequence of statements S_3 is transformed by using the δ^{-1} transformation recursively. If the statement is an input or an output statement,

$\delta_e^{-1}(e, c, A) \;=\; e'$ where

$e \;= <\;$ GET-a, $[c] >$ *and* $a \;\in A \;\Rightarrow$ gom-variable(e') *and* $e' \;=\; a$ *and*

$e \;= <\;$ GET-a, $[c] >$ *and* $a \;\notin A \;\Rightarrow e' \;=\; e$ *and*

gom-variable(e) $\;\Rightarrow e' \;=\; e$ *and*

gom-function-call(e) $\Rightarrow e' \;=\; e$ *and*

gom-literal-boolean(e) $\Rightarrow e' \;=\; e$ *and*

gom-literal-integer(e) $\Rightarrow e' \;=\; e$ *and*

gom-literal-real(e) $\Rightarrow e' \;=\; e$ *and*

gom-literal-string(e) $\Rightarrow e' \;=\; e$ *and*

$e \;= < e_1,\; +, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1),\; +, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1,\;$ and, $e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1),\;$ and, $\delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1,\; \&, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1),\; \&, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1,\; /, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1),\; /, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1,\; =, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1),\; =, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1,\; **, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1),\; **, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1, >=, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1), >=, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1, >, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1), >, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1, <=, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1), <=, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1, <, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1), <, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1,\; *, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1),\; *, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1, <>, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1), <>, \delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1,\;$ or, $e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1),\;$ or, $\delta_e^{-1}(e_2) >$ *and*

$e \;= < e_1, -, e_2 > \Rightarrow\; e' \;= < \delta_e^{-1}(e_1), -, \delta_e^{-1}(e_2) >$ *and*

$e \;= < -, e_1 > \Rightarrow\; e' \;= < -, \delta_e^{-1}(e_1) >$ *and*

$e \;= <\;$ not, $e_1 > \Rightarrow\; e' \;= <\;$ not, $\delta_e^{-1}(e_1) >$

Figure 141 The δ_e^{-1} transformation

$$
\begin{array}{l}
\delta^{-1}(\Psi, c, A) \;=\; \Psi' \text{ where} \\
\quad \forall\, a \in A \; and \; \forall\, s \in \Psi \\
\qquad s \;=\; <\; \text{SET-a}, [c, e] \;>\Rightarrow \\
\qquad\quad s' \;=\; <a,\; :=, \delta_e^{-1}(e, c, A) > \; and \\
\qquad s \;=\; <x,\; :=, e > \;\Rightarrow \\
\qquad\quad s' \;=\; <x,\; :=, \delta_e^{-1}(e, c, A) > and \\
\qquad s \;=\; < \text{if}, e,\; \text{then}, S_1,\; \text{else}, S_2 > \;\Rightarrow \\
\qquad\quad s' \;=\; < \text{if}, \delta_e^{-1}(e, c, A),\; \text{then}, \delta^{-1}(S_1, c, A),\; \text{else}, \delta^{-1}(S_2, c, A) > \; and \\
\qquad s \;=\; < \text{while}, e, S_3 > \;\Rightarrow \\
\qquad\quad s' \;=\; < \text{while}, \delta_e^{-1}(e, c, A), \delta^{-1}(S_3, c, A) > \; and \\
\qquad s \;=\; < \text{input},\; \text{iport}, E > \;\Rightarrow \\
\qquad\quad s' \;=\; < \text{input},\; \text{iport}, E' > \; and \\
\qquad\qquad E' \;=\; [\delta_e^{-1}(e, c, A) \mid e \in E] \; and \\
\qquad s \;=\; < \text{output},\; \text{oport}, E > \;\Rightarrow \\
\qquad\quad s' \;=\; < \text{output},\; \text{oport}, E' > \; and \\
\qquad\qquad E' \;=\; [\delta_e^{-1}(e, c, A) \mid e \in E] \; and \\
\qquad s' \;\in \Psi'
\end{array}
$$

Figure 142 The δ^{-1} transformation

then the sequence of expressions, E, in the statement is transformed by using the δ_e^{-1} transformation on each of the expressions in the sequence. If none of the GOM variables in the set A are accessed in the statements in Ψ or the set A is empty, then the statements in Ψ are returned unchanged.

6.4 Transforming Subprograms (σ)

This section defines the high-level transformation, σ, that applies lower-level transformations to formalize the PBOI method defined in Section 5.3. The σ transformation uses the taxonomy of GIM subprograms defined in Section 5.2 to classify the input subprogram and apply the transformation appropriate for that category of subprograms.

Let S represent an imperative subprogram and *OOD* represent the object-oriented design developed so far. The σ transformation is defined as follows.

$$\begin{aligned}\sigma(OOD,S) &= OOD' \text{ where}\\ Cat_0(S) &\Rightarrow OOD' = \sigma_2(OOD,S)\ and\\ Cat_1(S) &\Rightarrow OOD' = \sigma_1(OOD,S)\ and\\ Cat_2(S) &\Rightarrow OOD' = \sigma_2(OOD,S)\ and\\ Cat_3(S) &\Rightarrow OOD' = \sigma_3(OOD,S)\end{aligned}$$

Note that Category 0 subprograms are treated as a special case of Category 2 subprograms, so the σ_2 transformation, defined in the following section, is used to transform both Category 0 and Category 2 subprograms. The transformations for Category 1 and Category 3 subprograms are defined in Section 6.8. The transformations that formalize the program slicing process used to convert Category 4 and Category 5 subprograms are defined in Section 6.12.

6.5 Transforming Category 2 Subprograms (σ_2)

This section describes the formal transformation of Category 2 subprograms to the object paradigm. As defined in Section 5.3, the PBOI method assumes Category 2 subprograms implement derived attribute queries. According to Proposition V.2, each Category 2 subprogram is converted to a method in a class where the attributes of the class are built from the formal parameters of the Category 2 subprogram (see Section 5.3).

The formal transformation σ_2 transforms an imperative subprogram, S, into a class, C, which includes a method that implements S. The new class, C, is added to the object-oriented design developed so far, OOD, and an updated design OOD' is returned from the transformation.

Let id_C be a symbol representing the name of the newly formed class C. This symbol can be a computer generated one or can be provided by the user. Let λ_0 be a symbol that represents the name of the overall super-class (`'USER-OBJECT`) from which every object inherits in the design being developed. Let c represent a parameter that holds an instance of the class C. Then, the transformation σ_2 for a Category 2 subprogram S is defined as follows.

$$
\begin{aligned}
\sigma_2(OOD,\ S) &= OOD' \text{ where} \\
S &= < id_S, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma_S > \ and \\
P_{form} &= P_{in} \oplus P_{out} \ and \\
C &= < id_C, \Phi_C, \Omega_C, \lambda_0 > \ and \\
\Phi_C &= range(\theta_V(P_{form})) \ and \\
Q_{ret} &= \theta_V(P_{ret}) \ and \\
Q_{loc} &= \theta_V(P_{loc}) \ and \\
\Psi_C &= \delta(\theta(\Sigma_S), c, \Phi_C) \ and \\
\Omega_C &= \{< id_S, c, \emptyset, \emptyset, Q_{ret}, Q_{loc}, \Psi_C >\} \ and \\
OOD' &= T_c^+(OOD, C)
\end{aligned}
$$

This transformation builds an object-oriented class C from the imperative subprogram S by first transforming all the input parameters in P_{in} and output parameters in P_{out} into instance attributes, Φ_C, of C by using the θ_V transformation. The return value and the local variables are also converted to GOM variables using θ_V. The newly formed class C is a subclass of λ_0 and includes one method in Ω_C that implements the functionality of the imperative subprogram. The newly formed method is named using the name of the imperative subprogram id_S. The method must be passed an instance of the class C as the target object and is not passed any other objects or input or output parameters. Each statement of the subprogram in Σ_S is transformed into an object-oriented statement using the θ transformation. In these statements, all accesses to the variables that are now included in Φ_C are changed to accesses of these attributes by using the δ transformation. The new class C is added to the object design by using the $T_c^+(OOD, C)$ transformation.

6.6 Transforming Parameters

This section presents formal transformations for manipulating parameters of methods and attributes of classes. As presented in Section 5.3, in some cases when using the PBOI method it is necessary to add or remove attributes of classes. In order to maintain the *visibility* to the data item, an attribute removed from a class is added as a parameter to each method of the class. Conversely, when an attribute is added to a class the corresponding parameter is removed from each of the methods of the class.

For these reasons, formal transformations that convert parameters of the methods of a class into attributes of that class (and vice versa) have been developed. The transformations presented in Section 4.32 for adding and removing an attribute of a class (T_a^+ and T_a^-) are sufficient for developing a design, but are not rigorous enough for re-engineering. They do not consider the effects on the methods of a class that adding or removing an attribute will have. The transformations T_a^+ and T_a^- will not be used when converting GIM subprograms to the GOM. Instead, the following transformations will be used to add and delete attributes of a class.

6.6.1 Moving Parameters to Attributes (T_A^δ).

Let T_A^δ be a transformation that moves parameters of the methods of a class to attributes of the class. Let OOD represent an object-oriented design, let C represent a class, and let A represent a set of GOM variables to be converted from parameters to attributes. Let $seq(A)$ represent the conversion of the set A to a sequence of arbitrary order. The T_A^δ transformation is defined as shown below.

$$\begin{aligned}
T_A^\delta(OOD, C, A) \;&=\; C' \text{ where} \\
&C \;=\; < id_C, \Phi_C, \Omega_C, \lambda > \; and \\
&\Phi'_C \;=\; \Phi_C \;\cup A \; and \\
&\text{For each } o \in \Omega_C \\
&\quad o \;=\; < id_o, c, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi > \; and \\
&\quad o' \;=\; < id_o, c, Q_{obj}, Q_{form} \ominus seq(A), Q_{ret}, Q_{loc}, \delta(\Psi, c, A) > \; and \\
&\quad o' \in \Omega'_C \; and \\
&C' \;=\; < id_C, \Phi'_C, \Omega'_C, \lambda >
\end{aligned}$$

In this transformation, the attributes in A are added to the set of attributes for the class, i.e. Φ_C. Each method o of the class C is transformed by using the sequence subtraction operation $\ominus$ to ensure the data items in A are not in the sequence of formal parameters. All accesses to the data items in A in the statements of o' are transformed to attribute accesses of c using the δ transformation (see Section 6.3). If the set A is empty, then the T_A^δ transformation adds no attributes and changes no statements, so the class C is not changed by the transformation. The transformed class C' is returned as the result of the transformation.

6.6.2 Moving Attributes to Parameters ($T_A^{\delta^{-1}}$). When using the PBOI method, it is also necessary to transform attributes of classes into parameters of its methods. For this reason, the "inverse" of the T_A^{δ} transformation is defined. Let $T_A^{\delta^{-1}}$ be a transformation that converts a set of attributes of a class into parameters of the methods of the class. Let OOD represent an object-oriented design, let C represent a class, and let A represent a set of data items to be converted from attributes to parameters. Let $seq(A)$ represent the conversion of the set A to a sequence of arbitrary order. The $T_A^{\delta^{-1}}$ transformation is defined as follows.

$$\begin{aligned}
T_A^{\delta^{-1}}(OOD, C, A) \;&=\; C' \text{ where}\\
&C \;=\; < id_C, \Phi_C, \Omega_C, \lambda > \; and\\
&\Phi'_C \;=\; \Phi_C - A \; and\\
&For\ each\ o \in \Omega_C\\
&\quad o \;=\; < id_o, c, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi > \; and\\
&\quad o' \;=\; < id_o, c, Q_{obj}, Q_{form} \oplus seq(A), Q_{ret}, Q_{loc}, \delta^{-1}(\Psi, c, A) > \; and\\
&\quad o' \in \Omega'_C \; and\\
&C' \;=\; < id_C, \Phi'_C, \Omega'_C, \lambda >
\end{aligned}$$

In this transformation, the attributes in A are removed from the set of attributes for the class, Φ_C. Each of the methods in the class are transformed by adding the data items in A as parameters of o'. Finally, the δ^{-1} transformation is used to convert all attribute accesses in the statements of o' to variable accesses. If the set A is empty, then the $T_A^{\delta^{-1}}$ transformation removes no attributes and changes no statements, so the class C is not changed by the transformation. The updated class C' is returned as the result of the transformation. Note that the T_A^{δ} transformation and the $T_A^{\delta^{-1}}$ transformation are not true inverses because $T_A^{\delta^{-1}}(T_A^{\delta}(OOD, C, A), C, A) \neq C$. This is because the order of some parameters may have changed.

6.7 Transforming Attributes (γ)

This section defines a formal transformation for moving an attribute from one class to another. In some cases when using the PBOI method (see Section 5.3), it may be determined that a parameter should be built as an attribute of one class instead of another.

To formalize this case, a transformation is needed that moves an attribute from one class to another. This section presents this formal transformation and a simplified version of the transformation.

To define the transformation that moves an attribute from one class, C_1, to another class, C_2, a transformation is needed that replaces instances of C_1 with instances of C_2 in any messages where C_1 is the target object. Let γ be a transformation that changes all attribute accesses of specific attributes from an instance of one class to an instance of a second class. Let A be a subset of attributes from a class C_1. Let c_1 be an instance of C_1 and c_2 be an instance of the class C_2. Let Ψ be a sequence of object-oriented statements. Then,

$$\Psi' = \gamma(\Psi, c_1, c_2, A)$$

where Ψ' is a sequence of object-oriented statements where all attribute accesses of the attributes in A from the instance c_1 are now attribute accesses of the attributes in A for the instance c_2.

For example, consider the following statement

```
X := GET-BETA ( C-8 )
```

The following transformation

$$\gamma([\text{ x := GET-BETA (C-8) }],\ \text{C-8},\ \text{C-9}, \{\text{ BETA}\})$$

results in the following updated statement

```
X := GET-BETA ( C-9 )
```

The access of `BETA` now comes from `C-9` instead of `C-8`.

Using the γ transformation, it is now possible to define the T_A^γ transformation that moves attributes from one class to another. Let *OOD* represent an object-oriented design, let C_1 be a class, let c_1 be an instance of C_1, let C_2 be a class, and let c_2 be an instance of C_2. Let A be a set of attributes of C_1 that will be moved from C_1 to C_2. The T_A^γ

transformation assumes that the instance c_1 is passed as a parameter to the methods of the class C_2 in order to have visibility to the attributes in A. The T_A^γ transformation is defined as follows.

$$
\begin{array}{l}
T_A^\gamma(OOD, C_1, C_2, A) \ = \ OOD' \text{ where} \\
\quad C_1 \ = <id_{C_1}, \Phi_{C_1}, \Omega_{C_1}, \lambda_{C_1}> \ and \\
\quad instance(c_1, C_1) \ and \\
\quad C_2 \ = <id_{C_2}, \Phi_{C_2}, \Omega_{C_2}, \lambda_{C_2}> \ and \\
\quad instance(c_2, C_2) \ and \\
\quad \Phi'_{C_1} \ = \ \Phi_{C_1} \ - \ A \ and \\
\quad \Phi'_{C_2} \ = \ \Phi_{C_2} \ \cup \ A \ and \\
\quad \text{For each } o \in \Omega_{C_1} \\
\qquad o \ = <id_o, c_1, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi> \ and \\
\qquad o' \ = <id_o, c_1, Q_{obj} \oplus [c_2], Q_{form}, Q_{ret}, Q_{loc}, \gamma(\Psi, c_1, c_2, A)> \ and \\
\qquad o' \in \Omega'_{C_1} \ and \\
\quad \text{For each } o \in \Omega_{C_2} \\
\qquad o \ = <id_o, c_2, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi> \ and \\
\qquad o' \ = <id_o, c_2, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \gamma(\Psi, c_1, c_2, A)> \ and \\
\qquad o' \in \Omega'_{C_2} \ and \\
\quad C'_1 \ = <id_{C_1}, \Phi'_{C_1}, \Omega'_{C_1}, \lambda_{C_1}> \ and \\
\quad C'_2 \ = <id_{C_2}, \Phi'_{C_2}, \Omega'_{C_2}, \lambda_{C_2}> \ and \\
\quad OOD' \ = \ T_c^+(T_c^-(T_c^+(T_c^-(OOD, C_1), C'_1), C_2), C'_2)
\end{array}
$$

This transformation removes the attributes in A from C_1 and adds them to the set of attributes in C_2. Each method in C_1 is updated by adding the instance c_2 to the sequence of objects passed into the method. The statements of each method are changed by the γ transformation to send any accessing and assigning messages to c_2 instead of c_1. The statements of each method are changed by the γ transformation to also send any accessing and assigning messages to c_2 instead of c_1. The methods in both C_1 and C_2 now access the attributes in A from the instance c_2. If the set A is empty, then the T_A^γ transformation moves no attributes and neither C_1 nor C_2 are changed by the transformation. The updated classes are added to the design OOD in order to produce the updated design OOD'.

As an example, Figure 143 shows an object-oriented design with two classes where the GOM variable `DIAM` is an attribute of `CLASS-4`. The `PRDIV` method is passed `C-9`, an instance of CLASS-4, in order to access the DIAM attribute. Figure 144 shows the result of the transformation

$$T_A^{\gamma}(OOD, \text{CLASS-4, CLASS-5, DIAM})$$

The DIAM attribute has been moved from CLASS-4 to CLASS-5 and the methods have been updated to reflect this change.

The T_A^{γ} transformation can be simplified if it is known that the attributes in A are already attributes of C_2 and the desired processing is to make appropriate updates to C_1. Given this precondition, define $T_A^{\gamma'}$ as follows.

$T_A^{\gamma'}(OOD, C_1, C_2, A) \ = \ C_1'$ where
$C_1 \ = <id_{C_1}, \Phi_{C_1}, \Omega_{C_1}, \lambda_{C_1}> \ and$
$instance(c_1, C_1) \ and$
$instance(c_2, C_2) \ and$
$\Phi'_{C_1} \ = \ \Phi_{C_1} \ - \ A \ and$
For each $o \in \Omega_{C_1}$
$o \ = <id_o, c_1, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi> \ and$
$o' \ = <id_o, c_1, Q_{obj} \oplus [c_2], Q_{form}, Q_{ret}, Q_{loc}, \gamma(\Psi, c_1, c_2, A)> \ and$
$o' \in \Omega'_{C_1} \ and$
$C_1' \ = <id_{C_1}, \Phi'_{C_1}, \Omega'_{C_1}, \lambda_{C_1}>$

Since it is assumed that the attributes in A have already been built as attributes of C_2, this transformation removes the attributes from the set of attributes for C_1 but makes no change to the set of attributes for C_2. Similarly, only the methods of C_1 are updated to access the attributes in A from c_2 instead of c_1. The methods in C_2 are assumed to already access the attributes from c_2. If the set of attributes A is empty, then the $T_A^{\gamma'}$ transformation removes no attributes, so C_1 is not changed by the transformation. The updated class C_1 is returned as the result of this transformation.

For example, Figure 145 shows the DIAM attribute as an attribute of both CLASS-4 and CLASS-5. This attribute is being accessed correctly in CLASS-5 and moving the attribute from CLASS-4 to CLASS-5 is a matter of making the correct updates to CLASS-4. Figure 146

```
class CLASS-4 attributes DIAM, UPLFAC, ZENITH
  method RELAY-PHASE ( C-8, PHASE )
    begin
    PHASE := 1;
    SIGPR := GET-DIAM ( C-8 );
    PHASE := UPLREQ ( SIGPR )
    end
  superclass USER-OBJECT

class CLASS-5 attributes BETA, XLAMDA
  method PRDIV ( C-3, C-9 ): real begin
    FAC := 1.22d0;
    QUAL := MAX ( GET-BETA ( C-3), 1.0d0);
    PDIAM := MAX ( GET-DIAM ( C-9), 0.1d0);
    WAVELN := GET-XLAMDA ( C-3) * 9.9d-7;
    PRDIV := QUAL * WAVELN * FAC / PDIAM
    end
  superclass USER-OBJECT
```

Figure 143 DIAM as Attribute of CLASS-4

```
class CLASS-4 attributes UPLFAC, ZENITH
  method RELAY-PHASE ( C-8, C-9, PHASE )
    begin
    PHASE := 1;
    SIGPR := GET-DIAM ( C-9 );
    PHASE := UPLREQ ( SIGPR )
    end
  superclass USER-OBJECT

class CLASS-5 attributes BETA, XLAMDA, DIAM
  method PRDIV ( C-3 ): real begin
    FAC := 1.22d0;
    QUAL := MAX ( GET-BETA ( C-3), 1.0d0);
    PDIAM := MAX ( GET-DIAM ( C-3), 0.1d0);
    WAVELN := GET-XLAMDA ( C-3) * 9.9d-7;
    PRDIV := QUAL * WAVELN * FAC / PDIAM
    end
  superclass USER-OBJECT
```

Figure 144 DIAM as Attribute of CLASS-5

```
class CLASS-4 attributes DIAM, UPLFAC, ZENITH
  method RELAY-PHASE ( C-8, PHASE )
    begin
    PHASE := 1;
    SIGPR := GET-DIAM ( C-8 );
    PHASE := UPLREQ ( SIGPR )
    end
  superclass USER-OBJECT

class CLASS-5 attributes BETA, XLAMDA, DIAM
  method PRDIV ( C-3 ): real begin
    FAC := 1.22d0;
    QUAL := MAX ( GET-BETA ( C-3), 1.0d0);
    PDIAM := MAX ( GET-DIAM ( C-3), 0.1d0);
    WAVELN := GET-XLAMDA ( C-3) * 9.9d-7;
    PRDIV := QUAL * WAVELN * FAC / PDIAM
    end
  superclass USER-OBJECT
```

Figure 145 DIAM as Attribute of CLASS-4 and CLASS-5.

shows the result of applying the following transformation

$$T_A^{\gamma'}(OOD,\ \text{CLASS-4, CLASS-5, DIAM})$$

The `DIAM` attribute is removed from `CLASS-4` and `C-9`, the instance of `CLASS-5`, is added as an object parameter to the `RELAY-PHASE` method. The `GET-DIAM` message is updated to access `DIAM` from `C-9` instead of `C-8`.

```
class CLASS-4 attributes UPLFAC, ZENITH
  method RELAY-PHASE ( C-8, C-9, PHASE )
    begin
    PHASE := 1;
    SIGPR := GET-DIAM ( C-9 );
    PHASE := UPLREQ ( SIGPR )
    end
  superclass USER-OBJECT

class CLASS-5 attributes BETA, XLAMDA, DIAM
  method PRDIV ( C-3 ): real begin
    FAC := 1.22d0;
    QUAL := MAX ( GET-BETA ( C-3), 1.0d0);
    PDIAM := MAX ( GET-DIAM ( C-3), 0.1d0);
    WAVELN := GET-XLAMDA ( C-3) * 9.9d-7;
    PRDIV := QUAL * WAVELN * FAC / PDIAM
    end
  superclass USER-OBJECT
```

Figure 146 DIAM as Attribute of CLASS-5

6.8 Transforming Category 3 Subprograms

This section discusses the transformations that formalize the conversion of Category 3 subprograms. As defined in Section 5.3, there are four cases to consider when extracting object attributes from the imperative subprogram calls found in Category 3 subprograms. Some fundamental mappings required for all the PBOI transformations are presented first followed by the transformations that formalize each of the four cases. The overall high-level transformation that determines which of these transformations to apply is presented at the end of this section.

6.8.1 Linking Classes and Subprograms (ρ). In order to build the PBOI transformations, a mapping is needed that links an imperative subprogram to the class that was built for the subprogram. Let ρ represent this mapping as defined below.

$$\begin{aligned}
\rho(S) \;=\; & C \text{ such that} \\
& S = < id_S, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma_S > \;\; and \\
& C = < id_C, \Phi_C, \Omega_C, \lambda_0 > \;\; and \\
& o = < id_S, c, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi > \;\; and \\
& o \in \Omega_C
\end{aligned}$$

This mapping links the imperative subprogram S to a class that has a method o with the same name as S, i.e. id_S.

Since, at this point in the development of the design, there is only one class built for each subprogram and each class is built from only one subprogram, the ρ mapping is invertible. Let the ρ^{-1} mapping map a class C to the subprogram from which C was built. The ρ^{-1} mapping is defined as follows.

$$\begin{aligned}
\rho^{-1}(C) \;=\; & S \text{ such that} \\
& C = < id_C, \Phi_C, \Omega_C, \lambda_0 > \;\; and \\
& o = < id_S, c, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi > \;\; and \\
& o \in \Omega_C \;\; and \\
& S = < id_S, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma_S >
\end{aligned}$$

This mapping links the class C that was built for the subprogram S to this subprogram S.

$$
\begin{aligned}
T^1_{pboi}(OOD, S, P) &= OOD' \text{ where} \\
C_1 &= \rho(S) \; and \\
\forall\, p \in P & \\
P' &= \{p \mid \exists\, C_2 \in \rho^*(S) \text{ such that} \\
a' &= \theta_v(\mu^*(p)) \; and \; a' \in \Phi_{C_2} \} \; and \\
C_1' &= T_A^{\gamma'}(OOD, C_1, C_2, \theta_v(P')) \; and \\
C_1' \in OOD' & \wedge \; C_1 \notin OOD'
\end{aligned}
$$

Figure 147 The T^1_{pboi} transformation

6.8.2 Formalizing PBOI Case 1. The first case for converting Category 3 subprograms is that the formal parameter is an attribute and the actual parameter is also a formal. This section develops the transformation that formalizes the PBOI Case 1 as presented in Section 5.3. When transforming a subprogram S, the transformation required for PBOI Case 1 is to move attributes from the class built for S to another class in the design. This transformation is done on any of the formal parameters in S that can be linked to the attributes of another class through the θ_v transformation and the μ^* relation. An example is given at the end of the section.

Let T^1_{pboi} be the transformation that formalizes the first PBOI case. Recall from Section 3.25, the μ relation maps an actual parameter of a subprogram call to a formal parameter in the called subprogram. Let μ^* represent the transitive closure of the μ relation. Let $call(S_1, S_2)$ be a relation from subprogram S_1 to subprogram S_2 that indicates S_1 includes an imperative subprogram call to S_2. Let $call^*(S)$ represent the transitive closure of the *call* relation for an imperative subprogram S. This is also termed the *call tree* of S. Let $\rho^*(S)$ represent the set of classes built for the subprograms in the call tree of S. Let OOD represent an object-oriented design, let S represent an imperative subprogram, and let P represent the set of actual parameters from S that are also formal parameters of S. The T^1_{pboi} transformation is defined as shown in Figure 147. This transformation moves each attribute in C_1 that has already been built as an attribute in C_2 from C_1 to C_2 using the $T_A^{\gamma'}$ transformation. The transitive closure of the links between actual parameters and formal parameters, μ^*, is used to find formal parameters of subprograms that are in the call tree of S that have been converted to attributes (using θ_v) of the corresponding class. The set P' collects each of the parameters from P that is linked to an attribute a' through $\mu^*(p)$ and θ_v. Each a' is an attribute of a class in the set of classes built for the call tree of S. The set P' is used in the $T_A^{\gamma'}$ transformation to indicate which attributes of C_1 to move from class C_1 to class C_2. The $T_A^{\gamma'}$ transformation is used because the attribute a' is already an attribute of class C_2. The original class C_1 is removed from the design and the updated class C_1' is added to the design in order to produce the updated design OOD'.

For example, Figure 148 shows the subprograms, `RADIUS`, `CAPTURE`, and `BOUNCE`. The `BOUNCE` subprogram calls the `RADIUS` and `CAPTURE` subprograms, so the mappings $call(BOUNCE, RADIUS)$ and $call(BOUNCE, CAPTURE)$ are in the transitive closure $call^*(BOUNCE)$. Figure 149 shows the classes built for `RADIUS`, `CAPTURE`, and `BOUNCE` before applying the T_{pboi}^1 transformation. Since `RADIUS` and `CAPTURE` are called by `BOUNCE`, the transformations from a subprogram to a class have already been done. The `RADIUS` subprogram has been transformed into `CLASS-1`, which is shown in the figure. The `CAPTURE` subprogram has been transformed into `CLASS-2`, which is also shown in Figure 149. The transformation of the subprogram `BOUNCE` is incomplete in this example, but the class being built for `BOUNCE` is shown as `CLASS-4` in the figure. Notice at this point in the transformation of `BOUNCE`, none of the subprogram calls to other subprograms have been converted to messages. Also note that `CLASS-4` currently has the attributes `AXMAJ`, `AXMIN`, `ANGFAC`, `PROJRA`, `SIGB`, `RNGFAC`, `RANGE`, and other attributes.

Let OOD represent the design that includes these classes built for `BOUNCE`, `CAPTURE`, and `RADIUS`. Consider the following transformation.

$T_{pboi}^1(OOD,$ BOUNCE, { AXMAJ, AXMIN, ANGFAC, RANGE, RNGFAC, SIGB, PROJRA })

This transformation is used to update OOD based on whether the parameters `AXMAJ`, `AXMIN`, `ANGFAC`, `PROJRA`, `SIGB`, `RNGFAC`, and `RANGE` have been built as attributes of any of the classes built for the call tree of `BOUNCE`. Note in Figure 148, `BOUNCE` calls `RADIUS` passing the `PROJRA`, `SIGB`, `RNGFAC`, and `RANGE` data items as actual parameters. The formal parameters that correspond to these actual parameters are the parameters `PROJRA`, `SIGB`, `RNGFAC`, and

```
real function RADIUS ( PROJRA, SIGB, RNGFAC, RANGE )
  begin
  if RNGFAC > 0.0 and RANGE > 0.0
    then SIGABS := DABS ( SIGB);
      SLOPE := SIGABS - PROJRA / RANGE;
      SPOT := SIGABS * RANGE;
      RAD := DABS ( PROJRA + SLOPE * RNGFAC);
      RADIUS := DMAX1 ( SPOT, RAD)
    else
      RADIUS := PROJRA
    endif
  end

real function CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC )
  begin
  TILTFA := DSIN ( ANGFAC);
  RMAREA := AXMAJ * AXMIN;
  EFAREA := RMAREA * TILTFA;
  BMAREA := BEAMRA * BEAMRA;
  if EFAREA <= BMAREA * 6.0d0
    then USED := EFAREA / BMAREA;
      CAPTURE := 1.0d0 - DEXP ( -USED)
    else CAPTURE := 1.0d0
    endif
  end

real function BOUNCE ( ANGFAC, RNGFAC, PROJRA, SIGB,
      RANGE, AXMAJ, AXMIN, XLAMDA)
  begin
  RHOSTD := 0.95 * PROJRA * ANGFAC;
  BEAMRA := RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
  BOUNCE :=
    CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
      * RHO ( RHOSTD, XLAMDA)
  end
```

Figure 148 Subprograms RADIUS, CAPTURE, and BOUNCE

```
class CLASS-1 attributes RANGE, RNGFAC, SIGB, PROJRA
  method RADIUS ( C-1 ): real begin
    if GET-RNGFAC ( C-1) > 0.0 and GET-RANGE ( C-1) > 0.0
   then SIGABS := DABS ( GET-SIGB ( C-1));
        SLOPE := SIGABS - GET-PROJRA ( C-1) / GET-RANGE ( C-1);
        SPOT := SIGABS * GET-RANGE ( C-1);
        RAD := DABS ( GET-PROJRA ( C-1) + SLOPE * GET-RNGFAC ( C-1));
        RADIUS := DMAX1 ( SPOT, RAD)
     else RADIUS := GET-PROJRA ( C-1) endif
    end
  superclass USER-OBJECT

class CLASS-2 attributes  BEAMRA, AXMAJ, AXMIN, ANGFAC
  method CAPTURE ( C-2 ): real
    begin
    TILTFA := DSIN ( GET-ANGFAC ( C-2));
    RMAREA := GET-AXMAJ ( C-2) * GET-AXMIN ( C-2);
    EFAREA := RMAREA * TILTFA;
    BMAREA := GET-BEAMRA ( C-2) * GET-BEAMRA ( C-2);
    if EFAREA <= BMAREA * 6.0d0
      then USED := EFAREA / BMAREA;
        CAPTURE := 1.0d0 - DEXP ( -USED)
      else CAPTURE := 1.0d0
      endif
    end
  superclass USER-OBJECT

class CLASS-4 attributes ANGFAC, RNGFAC, PROJRA, SIGB,
      RANGE, AXMAJ, AXMIN, XLAMDA
  method BOUNCE ( C-4 ): real
    begin
    RHOSTD := 0.95 * GET-PROJRA ( C-4)
      * GET-ANGFAC ( C-4);
    BEAMRA := RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
    BOUNCE :=
      CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
        * RHO ( RHOSTD, XLAMDA)
    end
  superclass USER-OBJECT
```

Figure 149 Classes Built for RADIUS, CAPTURE, and BOUNCE

RANGE declared in RADIUS. This means the following mappings are in the transitive closure μ^*.

$$\begin{aligned}
BOUNCE :: PROJRA &\xrightarrow{\mu^*} RADIUS :: PROJRA \\
BOUNCE :: SIGB &\xrightarrow{\mu^*} RADIUS :: SIGB \\
BOUNCE :: RNGFAC &\xrightarrow{\mu^*} RADIUS :: RNGFAC \\
BOUNCE :: RANGE &\xrightarrow{\mu^*} RADIUS :: RANGE
\end{aligned}$$

Also note that each of these formal parameters have been built as an attribute of the class built for RADIUS, i.e. CLASS-1 shown in Figure 149. All of this information means the T^1_{pboi} transformation is performed and the PROJRA, SIGB, RNGFAC, and RANGE attributes are moved from CLASS-4 to CLASS-1 using the $T^{\gamma'}_A$ transformation. All accesses to PROJRA, SIGB, RNGFAC, and RANGE in CLASS-4 are changed from accessing instances of CLASS-4 to accessing instances of CLASS-1.

Similarly, BOUNCE calls CAPTURE passing the BEAMRA, AXMAJ, AXMIN, and ANGFAC data items as actual parameters. The formal parameters that correspond to these actual parameters are the BEAMRA, AXMAJ, AXMIN, and ANGFAC parameters. The following mappings are in the transitive closure μ^*.

$$\begin{aligned}
BOUNCE :: BEAMRA &\xrightarrow{\mu^*} CAPTURE :: BEAMRA \\
BOUNCE :: AXMAJ &\xrightarrow{\mu^*} CAPTURE :: AXMAJ \\
BOUNCE :: AXMIN &\xrightarrow{\mu^*} CAPTURE :: AXMIN \\
BOUNCE :: ANGFAC &\xrightarrow{\mu^*} CAPTURE :: ANGFAC
\end{aligned}$$

Note that only the AXMAJ, AXMIN, and ANGFAC parameters are both formal and actual parameters in BOUNCE. In adherence with PBOI Case 1, these data items are sent to the T^1_{pboi} transformation, but the BEAMRA data item is not. Also note that the AXMAJ, AXMIN, and ANGFAC parameters have been built as attributes of the class CLASS-2. This means the AXMAJ, AXMIN, and ANGFAC attributes of CLASS-4 are moved from CLASS-4 to CLASS-2 using the $T^{\gamma'}_A$ transformation. All accesses of AXMAJ, AXMIN, and ANGFAC attributes are converted from accessing instances of CLASS-4 to accessing instances of CLASS-1.

Figure 150 shows the result of the T^1_{pboi} transformation. Note in the GET-PROJRA message, the PROJRA attribute is now obtained from C-5, an instance of CLASS-1, instead

```
class CLASS-4 attributes XLAMDA
  method BOUNCE ( C-4, C-5, C-6): real
    begin
    RHOSTD := 0.95 * GET-PROJRA ( C-5)
      * GET-ANGFAC ( C-6);
    BEAMRA := RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
    BOUNCE :=
      CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
        * RHO ( RHOSTD, XLAMDA)
    end
  superclass USER-OBJECT
```

Figure 150   Updated CLASS-4 for BOUNCE

of C-4, an instance of CLASS-4. In the GET-ANGFAC message, the ANGFAC attribute is now obtained from C-6, an instance of CLASS-2, instead of C-4, an instance of CLASS-4. The $T_A^{\gamma'}$ transformation has added C-5 and C-6 as formal parameters of the method BOUNCE.

6.8.3 Formalizing PBOI Case 2. The second case for converting Category 3 subprograms is that the formal parameter is a parameter of the method and the actual parameter is a formal. This section defines the transformation that formalizes the second PBOI case as presented in Section 5.3. The transformation needed in this case is to identify parameters of a called subprogram that are not attributes and move them to parameters of the method built for the calling subprogram. The transformation is defined and an example is given.

Recall from Section 6.8.2, μ^* represents the transitive closure of the μ relation, $call^*(S)$ (the *call tree* of S) represents the transitive closure of the *call* relation, and $\rho^*(S)$ represents the set of classes built for the subprograms in the call tree of S. Let OOD represent an object-oriented design, let S represent the subprogram to be transformed, and let P be the set of data items to consider in the transformation.

Using these definitions, the T_{pboi}^2 transformation that formalizes PBOI Case 2 is defined as shown in Figure 151. This transformation checks each of the classes built for the subprograms in the call tree of S. For any data item p, if there are no classes that have built an attribute that is linked to the parameter p through the μ^* mapping, then

$$\begin{aligned}
T^2_{pboi}(OOD, S, P) &= OOD' \text{ where} \\
P' &= \{p \mid p \in P \textit{ and } \neg\exists\, C_2 \in \rho^*(S) \text{ such that} \\
a' &= \theta_v(\mu^*(p)) \textit{ and } a' \in \Phi_{C_2}\} \textit{ and} \\
C_1 &= \rho(S) \textit{ and} \\
C_1' &= T_A^{\delta^{-1}}(OOD, C_1, \theta_v(P')) \textit{ and} \\
OOD' &= T_c^+(T_c^-(OOD, C_1), C_1')
\end{aligned}$$

Figure 151 The T^2_{pboi} transformation

an instance of PBOI Case 2 has been found. The set P' collects these data items so they can be changed from attributes to parameters. All of the data items in P' are changed from attributes of the class built for S, i.e. C_1, into parameters of the methods of C_1 using the $T_A^{\delta^{-1}}$ transformation. Once p is converted to a GOM variable using the θ_v transformation, it is appropriate to move the variable from an attribute to a parameter because both attributes and parameters are variables in the object paradigm. For example, Figure 152 shows the subprograms CAPTURE and BOUNCE (whose statements do not match those shown in Figure 148). The BOUNCE subprogram calls the CAPTURE subprogram, so the mapping *call(BOUNCE, CAPTURE)* is one of the mappings in the transitive closure *call*(BOUNCE)*. Figure 153 shows the classes built for CAPTURE and BOUNCE. The subprogram CAPTURE has been completely transformed since it is called by BOUNCE. CLASS-4 in the figure is the incomplete class being built for BOUNCE. At this point in the transformation, none of the subprogram calls have been converted to messages in CLASS-4.

Let *OOD* represent the design that includes the classes built for BOUNCE and CAPTURE. Consider the transformation

$$T^2_{pboi}(OOD,\ \text{BOUNCE},\ \{\ \text{AXMAJ, AXMIN, ANGFAC}\ \}\ )$$

This transformation is used to update *OOD* based on whether the parameters AXMAJ, AXMIN, or ANGFAC have been built as attributes of any of the classes in the call tree of BOUNCE. Note in the figure that even though BOUNCE calls the other subprograms RADIUS and RHO, the AXMAJ, AXMIN, and ANGFAC data items are only passed to the CAPTURE subprogram.

```
real function CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC )
  begin
  TILTFA := DSIN ( ANGFAC);
  RMAREA := AXMAJ * AXMIN;
  EFAREA := RMAREA * TILTFA;
  BMAREA := BEAMRA * BEAMRA;
  if EFAREA <= BMAREA * 6.0d0
    then USED := EFAREA / BMAREA;
      CAPTURE := 1.0d0 - DEXP ( -USED)
    else CAPTURE := 1.0d0
    endif
  end

real function BOUNCE ( ANGFAC, RNGFAC, PROJRA, SIGB,
      RANGE, AXMAJ, AXMIN, XLAMDA)
  begin
  RHOSTD := 0.95 * ANGFAC;
  BEAMRA := RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
  BOUNCE :=
    CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
      * RHO ( RHOSTD, XLAMDA)
  end
```

Figure 152 Subprograms CAPTURE and BOUNCE

This means, for this example, the only possible class that could have attributes that are linked to these data items through the μ^* relation is the class built for CAPTURE. As is seen in Figure 152, the actual parameters AXMAJ, AXMIN, and ANGFAC are linked to the formal parameters AXMAJ, AXMIN, and ANGFAC, respectively, declared in the CAPTURE subprogram. This means the following mappings are in the transitive closure μ^*.

$$\begin{aligned} BOUNCE :: AXMAJ &\xrightarrow{\mu^*} CAPTURE :: AXMAJ \\ BOUNCE :: AXMIN &\xrightarrow{\mu^*} CAPTURE :: AXMIN \\ BOUNCE :: ANGFAC &\xrightarrow{\mu^*} CAPTURE :: ANGFAC \end{aligned}$$

As shown in Figure 153, the AXMAJ and AXMIN data items have been built as attributes of the class CLASS-2, but the ANGFAC data item has not been built as an attribute. The ANGFAC data item is a parameter of the method that implements CAPTURE. This means in the T^2_{pboi} transformation, the only data item in the set P' is ANGFAC. This attribute is

```
class CLASS-2 attributes  BEAMRA, AXMAJ, AXMIN
  method CAPTURE ( C-2, ANGFAC ): real
    begin
    TILTFA := DSIN ( ANGFAC);
    RMAREA := GET-AXMAJ ( C-2) * GET-AXMIN ( C-2);
    EFAREA := RMAREA * TILTFA;
    BMAREA := GET-BEAMRA ( C-2) * GET-BEAMRA ( C-2);
    if EFAREA <= BMAREA * 6.0d0
      then USED := EFAREA / BMAREA;
        CAPTURE := 1.0d0 - DEXP ( -USED)
      else CAPTURE := 1.0d0
      endif
    end
  superclass USER-OBJECT

class CLASS-4 attributes ANGFAC, RNGFAC, PROJRA, SIGB,
      RANGE, AXMAJ, AXMIN, XLAMDA
  method BOUNCE ( C-4 ): real
    begin
    RHOSTD := 0.95 * GET-ANGFAC ( C-4);
    BEAMRA := RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
    BOUNCE :=
      CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
        * RHO ( RHOSTD, XLAMDA)
    end
  superclass USER-OBJECT
```

Figure 153   Classes Built for CAPTURE and BOUNCE

removed from CLASS-4 and added as a parameter to the method that implements BOUNCE by using the $T_A^{\delta^{-1}}$ transformation. Figure 154 shows the result of the T_{pboi}^2 transformation. The ANGFAC data item has been changed from an attribute of CLASS-4 to a parameter of the BOUNCE method. The message GET-ANGFAC (C-6), which accesses the attribute ANGFAC, has been changed to access the parameter ANGFAC. This updated class replaces the original class in the design OOD to produce the updated design OOD'.

6.8.4 Formalizing PBOI Case 3. The third case for converting Category 3 subprograms is that the formal parameter is an attribute and the actual parameter is not a formal. This section defines the transformation that formalizes the third case for PBOI

```
class CLASS-4 attributes RNGFAC, PROJRA, SIGB,
      RANGE, AXMAJ, AXMIN, XLAMDA
  method BOUNCE ( C-4, ANGFAC ): real
    begin
    RHOSTD := 0.95 * ANGFAC;
    BEAMRA := RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
    BOUNCE :=
      CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
        * RHO ( RHOSTD, XLAMDA)
    end
  superclass USER-OBJECT
```

Figure 154 Updated Class for BOUNCE

as described in Section 5.3. This case identifies parameters from the called subprogram that have been built as attributes of a class but are not formal parameters of the calling subprogram. The transformation needed for this case is to move the identified data items from attributes to parameters. The transformation for case three is defined below followed by an example.

As defined in Section 6.8.2, μ^* represents the transitive closure of the μ relation, $call^*(S_1)$ (the *call tree* of S_1) represents the transitive closure of the *call* relation, and $\rho^*(S_1)$ represents the set of classes built for the subprograms in the call tree of S_1. Let OOD represent an object-oriented design, let S_1 represent the subprogram being transformed, and let P be the set of data items being considered in the transformation. Let $\mu^*(P')$ represent the set of data items resulting from the application of μ^* to the data items in P'.

Using these definitions, the transformation that formalizes PBOI Case 3 is defined as shown in Figure 155. This transformation is fundamentally different from the transformations that formalize PBOI Case 1 and PBOI Case 2. The changes that were made to the design OOD in Case 1 and Case 2 were made only to the class that was built for the subprogram S. In the transformation for Case 3, no change is required for the class built for S_1, but other classes in the call tree of S_1 may be changed. If a class is found in $\rho^*(S_1)$ that has an attribute that can be linked to a parameter p through θ_v and the μ^* relation, then each of the classes built for the subprograms that include p as a formal parameter is

$$
\begin{aligned}
& T^3_{pboi}(OOD, S_1, P) \;=\; OOD' \text{ where} \\
& \quad C_1 \;=\; \rho(S_1) \; and \\
& \quad P' \;=\; \{p \mid p \in P \; and \; \exists \; C_3 \in \rho^*(S_1) \text{ such that} \\
& \quad\quad b' \;=\; \theta_v(\mu^*(p)) \; and \; b' \in \Phi_{C_3}\} \; and \\
& \quad \forall \; C_2 \in OOD \\
& \quad\quad S_2 \;=\; \rho^{-1}(C_2) \; and \\
& \quad\quad S_2 \;=\; < id_{S_2}, P_{ins_{S_2}}, P_{outs_{S_2}}, P_{ret}, P_{loc}, \Sigma_{S_2} > \; and \\
& \quad\quad P_{forms_{S_2}} \;=\; P_{ins_{S_2}} \oplus P_{outs_{S_2}} \; and \\
& \quad\quad A \;=\; \{a' \mid p' \in P' \; and \; a' = \; \mu^*(p') \; and \; a' \in P_{forms_{S_2}}\} \\
& \quad\quad C_2' \;=\; T_A^{\delta^{-1}}(OOD, C_2, \theta_v(A)) \; and \\
& \quad\quad C_2' \in OOD' \; \wedge \; C_2 \notin OOD'
\end{aligned}
$$

Figure 155 The T^3_{pboi} transformation

transformed. The set P' collects any such parameters from P. All the classes in the design OOD are examined to see if they require an update. The subprogram that is associated with each class C_2 is determined (by using ρ^{-1}) and the formal parameters of this subprogram S_2 are examined. If S_2 has a formal parameter that can be linked to $p' \in P'$ by the $\mu^*(p')$ transitive closure, then the T^3_{pboi} transformation must ensure this parameter has not been built as an attribute. The $T_A^{\delta^{-1}}$ transformation is used to transform the attributes associated with such parameters into parameters of the methods of C_2.

Let the term *call path* represent a mapping in the $call^*(S_1)$ transitive closure such that $call(S_1, S_2)$ indicates a call path from S_1 to S_2. Since the $call^*(S_1)$ relation is a transitive closure, there may be multiple call mappings in $call^*(S_1)$ of the form $call(S_1, S_n)$ and $call(S_n, S_m)$ and $call(S_m, S_2)$ that were used to build the mapping $call(S_1, S_2)$. All these subprograms, S_1, S_n, S_m, and S_2, are considered to be *in* the call path from S_1 to S_2. Let $S_3 \;=\; \rho^{-1}(C_1)$, where C_1 is the class that has built the parameter p as an attribute. The T^3_{pboi} transformation is built specifically to update all the classes that have been built for the subprograms in the call path from S_1 and S_3. This is a critical part of the transformation because each method that implements the subprograms in the call path from S_1 to S_3 must now access p as a parameter instead of as an attribute of some object.

This is why each of the classes in the design OOD is checked and the $\mu^*(p')$ transitive closure is used to identify such parameters.

In some classes, the data item a' will not be an attribute of the class C_2. The $T_A^{\delta^{-1}}$ transformation will make the correct transformation because set difference is used by the $T_A^{\delta^{-1}}$ transformation to remove a' from the set of attributes for the class. Whether or not a' is in the set of attributes, the result is a set of attributes that does not contain a'. In some cases, the data item a' will already be a parameter of the methods of a class. In this case the $T_A^{\delta^{-1}}$ transformation will make the correct transformation because the $\oplus$ operation (defined in Section 3.24) is used to add a' to the sequence of formal parameters of each method. The $\oplus$ operation adds a parameter to a sequence of parameters without allowing duplicates. Whether or not a' is originally in the sequence of parameters for the method, the result is a sequence of parameters that now includes a'.

As an example, Figure 156 shows the subprograms RHO, BOUNCE, and RELAY-PHASE. These subprograms have been altered from previous examples. The RELAY-PHASE subprogram calls the BOUNCE subprogram, so the $call(RELAY\text{-}PHASE, BOUNCE)$ mapping is included in the transitive closure $call^*(RELAY\text{-}PHASE)$. The BOUNCE subprogram calls the RHO subprogram, so the $call(BOUNCE, RHO)$ and the $call(RELAY\text{-}PHASE, RHO)$ mappings are also included in the transitive closure $call^*(RELAY\text{-}PHASE)$. In this example, the call path from RELAY-PHASE to RHO includes the subprograms RHO, BOUNCE, and RELAY-PHASE.

The classes built for these subprograms are shown in Figure 157. The subprograms BOUNCE and RHO have already been transformed since they are called by RELAY-PHASE. The incomplete class being built for RELAY-PHASE is shown in the figure as CLASS-5. The subprogram call to BOUNCE has not yet been converted to a message.

Let OOD represent the design that includes these three classes. Consider the transformation

$$T^3_{pboi}(OOD, \text{ RELAY-PHASE, \{ RHOSTD, XLAMDA \} })$$

```
real function RHO ( RHOSTD, LAMBDA)
  begin
    RHO := RHOSTD * LAMBDA
  end

real function BOUNCE ( ANGFAC, RNGFAC, PROJRA, SIGB,
      RANGE, AXMAJ, AXMIN, RHOSTD, XLAMDA)
  begin
  BEAMRA := RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
  BOUNCE :=
    CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
      * RHO ( RHOSTD, XLAMDA)
  end

procedure RELAY-PHASE ( DIAM, UPLFAC, RNGFAC, ZENITH, SIGB,
       AXMAJ, AXMIN, PHASE )
    begin
    PHASE := 1;
    RHOSTD := 0.95;
    ANGFAC := UPLFAC * 3.14159;
    PROJRA := DIAM / 2;
    RANGE := ZENITH * PROJRA * ANGFAC;
    XLAMDA := 0.625;
    PHASE := BOUNCE ( ANGFAC, RNGFAC, PROJRA, SIGB,
      RANGE, AXMAJ, AXMIN, RHOSTD, XLAMDA)
    end
```

Figure 156   Subprograms RHO, BOUNCE, and RELAY-PHASE

```
class CLASS-3 attributes RHOSTD, LAMBDA
  method RHO ( C-3 ): real
    begin
      RHO := GET-RHOSTD ( C-3) * GET-LAMBDA ( C-3)
    end
  superclass USER-OBJECT

class CLASS-4 attributes
  method BOUNCE ( C-4, C-5, C-6, C-7 ): real
    begin
    BEAMRA := RADIUS ( C-5);
    BOUNCE := CAPTURE ( C-6, BEAMRA) * RHO ( C-7)
    end
  superclass USER-OBJECT

class CLASS-5 attributes DIAM, UPLFAC, RNGFAC, ZENITH, SIGB,
       AXMAJ, AXMIN, PHASE
  method RELAY-PHASE ( C-8 )
    begin
    SET-PHASE ( C-8, 1);
    RHOSTD := 0.95;
    ANGFAC := GET-UPLFAC ( C-8) * 3.14159;
    PROJRA := GET-DIAM ( C-8) / 2;
    RANGE := GET-ZENITH ( C-8) * PROJRA * ANGFAC;
    XLAMDA := 0.625;
    SET-PHASE ( C-8, BOUNCE ( ANGFAC, RNGFAC, PROJRA, SIGB,
      RANGE, AXMAJ, AXMIN, RHOSTD, XLAMDA))
    end
  superclass USER-OBJECT
```

Figure 157   Classes Built for RHO, BOUNCE, and RELAY-PHASE

This transformation updates the *OOD* to comply with PBOI Case 3 since the data items RHOSTD and XLAMDA are actual parameters but are not formal parameters in the subprogram RELAY-PHASE. As shown in the figure, the RELAY-PHASE subprogram calls the BOUNCE subprogram passing RHOSTD and XLAMDA as actual parameters. The corresponding formal parameters in BOUNCE are the RHOSTD and XLAMDA formal parameters. This means the following mappings are in the transitive closure μ^*.

$$\begin{aligned} \textit{RELAY-PHASE} :: \textit{RHOSTD} &\xrightarrow{\mu^*} \textit{BOUNCE} :: \textit{RHOSTD} \\ \textit{RELAY-PHASE} :: \textit{XLAMDA} &\xrightarrow{\mu^*} \textit{BOUNCE} :: \textit{XLAMDA} \end{aligned}$$

The subprogram BOUNCE calls the subprogram RHO passing the RHOSTD and XLAMDA parameters as actual parameters. The formal parameters that correspond to these actual parameters are the RHOSTD and LAMBDA formal parameters declared in RHO. This means the following mappings are in the transitive closure μ^*.

$$\begin{aligned} \textit{BOUNCE} :: \textit{RHOSTD} &\xrightarrow{\mu^*} \textit{RHO} :: \textit{RHOSTD} \\ \textit{BOUNCE} :: \textit{XLAMDA} &\xrightarrow{\mu^*} \textit{RHO} :: \textit{LAMBDA} \\ \textit{RELAY-PHASE} :: \textit{RHOSTD} &\xrightarrow{\mu^*} \textit{RHO} :: \textit{RHOSTD} \\ \textit{RELAY-PHASE} :: \textit{XLAMDA} &\xrightarrow{\mu^*} \textit{RHO} :: \textit{LAMBDA} \end{aligned}$$

Notice in Figure 157, the class CLASS-3 built for RHO has built the RHOSTD and LAMBDA data items as attributes of the class. This means a class exists in the call tree for RELAY-PHASE that includes an attribute that can be linked to the RHOSTD and XLAMDA data items in RELAY-PHASE. Both RHOSTD and XLAMDA are collected into the set P' for further evaluation. The T^3_{pboi} transformation continues by examining each class in the design *OOD* to see if it needs to be transformed.

For this example, *OOD* includes the three classes CLASS-3, CLASS-4, and CLASS-5. For CLASS-5, the T^3_{pboi} transformation uses the ρ^{-1} mapping to find the subprogram RELAY-PHASE. The set A for this subprogram is empty since, in this example, each p' is a variable defined in RELAY-PHASE and there are no mappings in μ^* that map data items in RELAY-PHASE to other data items in RELAY-PHASE. The only way for those mappings to be in μ^* would be from a *recursive* call to RELAY-PHASE, and recursion is not allowed in the

GOM. This means the empty set A is passed to the $T_A^{\delta^{-1}}$ transformation and no change is made to CLASS-5, as required.

For CLASS-4, the subprogram associated with this class is the BOUNCE subprogram. The set A for BOUNCE is { RHOSTD, XLAMDA } since these data items have mappings in μ^* to the RHOSTD and XLAMDA data items, respectively, in RELAY-PHASE. The $T_A^{\delta^{-1}}$ transformation is passed these data items which ensures RHOSTD and XLAMDA aren't attributes of CLASS-4, but are parameters of the methods of CLASS-4.

```
class CLASS-4 attributes
  method BOUNCE ( C-4, C-5, C-6, C-7, RHOSTD, XLAMDA ): real
    begin
    BEAMRA := RADIUS ( C-5);
    BOUNCE := CAPTURE ( C-6, BEAMRA)
      * RHO ( C-7, RHOSTD, XLAMDA)
    end
  superclass USER-OBJECT
```

Figure 158 Updated Class for BOUNCE

Figure 158 shows the result of this transformation. The RHOSTD and XLAMDA data items are not attributes of CLASS-4, but are formal parameters of the method BOUNCE and actual parameters in the message RHO.

For CLASS-3, the subprogram associated with this class is the RHO subprogram. The set A for RHO is { RHOSTD, LAMBDA } since these data items have mappings in μ^* to the RHOSTD and XLAMDA data items, respectively, in RELAY-PHASE. The $T_A^{\delta^{-1}}$ transformation is passed { RHOSTD, LAMBDA } which ensures RHOSTD and LAMBDA are not attributes of CLASS-3, but are parameters of the methods of CLASS-3.

Figure 159 shows the result of the $T_A^{\delta^{-1}}$ transformation on CLASS-3. The RHOSTD and LAMBDA data items are now parameters of the RHO method instead of attributes of the CLASS-3 class. The three updated classes are included in the updated design returned from the T_{pboi}^3 transformation.

```
class CLASS-3 attributes
  method RHO ( C-3, RHOSTD, LAMBDA ): real
    begin
      RHO := RHOSTD * LAMBDA
    end
  superclass USER-OBJECT
```

Figure 159 Updated Class for RHO

$$\begin{aligned} T^4_{pboi}(OOD, S, P) &= OOD' \text{ where} \\ P' &= \{p \mid p \in P \; and \; \neg\exists \; C_1 \in \rho^*(S) \text{ such that} \\ a' &= \theta_v(\mu^*(p)) \; and \; a' \in \Phi_{C_1}\} \; and \\ OOD' &= OOD \end{aligned}$$

Figure 160 The T^4_{pboi} transformation

6.8.5 Formalizing PBOI Case 4. The fourth case for converting Category 3 subprograms is that the formal parameter is a parameter of the method and the actual parameter is not a formal. This section defines the formal transformation that formalizes the fourth case of the PBOI method, as defined in Section 5.3. This case identifies parameters of the called subprogram that are not attributes of a class and are not formal parameters of the calling subprogram. There is no change to the object-oriented design required for this case. The transformation is defined below and an example is given at the end of this section.

As defined in Section 6.8.2, μ^* represents the transitive closure of the μ relation, *call**(S) (the *call tree* of S) represents the transitive closure of the *call* relation, and $\rho^*(S)$ represents the set of classes built for the subprograms in the call tree of S. The T^4_{pboi} formal transformation is defined in Figure 160. This transformation checks each class in the set of classes built for the subprograms in the call tree of S. If none of the classes have an attribute that can be linked through the μ^* relation to the parameter p, then there is no change required in the design.

For example, consider the RADIUS and BOUNCE subprograms shown in Figure 161. These subprograms have been changed slightly to illustrate PBOI Case 4. The PROJRA

```
real function RADIUS ( PROJRA, SIGB, RNGFAC, RANGE )
  begin
  if RNGFAC > 0.0 and RANGE > 0.0
    then SIGABS := DABS ( SIGB);
      SLOPE := SIGABS - PROJRA / RANGE;
      SPOT := SIGABS * RANGE;
      RAD := DABS ( PROJRA + SLOPE * RNGFAC);
      RADIUS := DMAX1 ( SPOT, RAD)
    else
      RADIUS := PROJRA
    endif
  end

real function BOUNCE ( ANGFAC, RNGFAC, SIGB,
      RANGE, AXMAJ, AXMIN, XLAMDA)
  begin
  PROJRA := XLAMDA * 3.14159;
  RHOSTD := 0.95;
  BEAMRA := RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
  BOUNCE :=
    CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
      * RHO ( RHOSTD, XLAMDA)
  end
```

Figure 161 Subprograms RADIUS and BOUNCE

```
class CLASS-1 attributes RANGE, RNGFAC, SIGB
  method RADIUS ( C-1, PROJRA ): real begin
    if GET-RNGFAC ( C-1) > 0.0 and GET-RANGE ( C-1) > 0.0
      then SIGABS := DABS ( GET-SIGB ( C-1));
        SLOPE := SIGABS - PROJRA / GET-RANGE ( C-1);
        SPOT := SIGABS * GET-RANGE ( C-1);
        RAD := DABS ( PROJRA + SLOPE * GET-RNGFAC ( C-1));
        RADIUS := DMAX1 ( SPOT, RAD)
      else
        RADIUS := PROJRA
      endif
    end
  superclass USER-OBJECT

class CLASS-4 attributes ANGFAC, RNGFAC, SIGB,
      RANGE, AXMAJ, AXMIN, XLAMDA
  method BOUNCE ( C-4 ): real
    begin
    PROJRA := XLAMDA * 3.14159;
    RHOSTD := 0.95;
    BEAMRA := RADIUS ( PROJRA, SIGB, RNGFAC, RANGE);
    BOUNCE :=
      CAPTURE ( BEAMRA, AXMAJ, AXMIN, ANGFAC)
        * RHO ( RHOSTD, XLAMDA)
    end
  superclass USER-OBJECT
```

Figure 162 Classes Built for RADIUS and BOUNCE

data item is now a local variable of the BOUNCE subprogram. In the example, the BOUNCE subprogram calls the RADIUS subprogram, so the *call*$(BOUNCE, RADIUS)$ mapping is included in the transitive closure *call**$(BOUNCE)$. Figure 162 shows the classes that are built for the RADIUS and BOUNCE subprograms. The RADIUS subprogram has been completely transformed into class CLASS-1 since it is called by BOUNCE. This class is different than the other example transformation of RADIUS (see Figure 149). In Figure 162, the PROJRA data item has *not* been built as an attribute of the class built for RADIUS. The incomplete class being built for BOUNCE is CLASS-4 as shown in the figure.

Let OOD represent the design developed so far that includes these two classes. Consider the transformation

$$T^4_{pboi}(OOD,\ \text{BOUNCE, PROJRA})$$

This transformation checks all the classes built for the subprograms in the call tree of `BOUNCE` to see if any of the classes have an attribute that can be linked through the μ^* relation to the `PROJRA` data item. The `BOUNCE` subprogram calls the `RADIUS` subprogram passing in `PROJRA` as an actual parameter. The formal parameter that corresponds to this actual parameter is the `PROJRA` formal parameter declared in `RADIUS`. This means the following mapping is in the transitive closure $\mu^*(BOUNCE :: PROJRA)$.

$$BOUNCE :: PROJRA \xrightarrow{\mu^*} RADIUS :: PROJRA$$

The `PROJRA` data item in `RADIUS` has not been built as an attribute. Since in the call tree of `BOUNCE` the `PROJRA` data item is passed only to the `RADIUS` subprogram, it can be concluded that there is not a class in the call tree of `BOUNCE` that has built `PROJRA` as an attribute.

6.8.6 Formalizing PBOI. This section defines the high-level transformation combining the four transformations that formalize the four PBOI cases. The formal transformation is defined and an example is given.

Let OOD represent an object-oriented design, let S represent the subprogram being transformed. Let C_{subs_S} be the set of imperative subprogram calls in S. Let P^*_{act} be the sequence collecting all the actual parameters from all the subprogram calls in S, such that duplicate parameters are not included and the order of the sequence is arbitrary. Let T_{pboi} be the high-level transformation that determines which of the four PBOI transformations to apply. The T_{pboi} transformation is defined as shown in Figure 163. This transformation is used to update the object-oriented design developed so far by applying the four PBOI transformations defined in the previous sections. This high-level transformation examines the subprogram calls that the input subprogram S makes and determines which PBOI formal transformation to apply for each actual parameter p of the call. The *reduce* operation

$$
\begin{aligned}
T_{pboi}(OOD, S) \;&=\; OOD_4 \text{ where}\\
&S \;= < id_S, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma_S > \; and\\
&P_{form} \;=\; P_{in} \,\oplus\, P_{out} \; and\\
&P^*_{act} \;=\; reduce(\oplus, [P_{act_n} \;\mid\; l \in C_{subs_S} \; and\\
&\quad l \;= < id_{S_n}, P_{act_n} >]) \; and\\
&OOD_1 \;=\; T^1_{pboi}(OOD, S, P_{form} \;\cap\; P^*_{act}) \; and\\
&OOD_2 \;=\; T^2_{pboi}(OOD_1, S, P_{form} \;\cap\; P^*_{act}) \; and\\
&OOD_3 \;=\; T^3_{pboi}(OOD_2, S, P_{form} \;-\; P^*_{act}) \; and\\
&OOD_4 \;=\; T^4_{pboi}(OOD_3, S, P_{form} \;-\; P^*_{act})
\end{aligned}
$$

Figure 163 The T_{pboi} transformation

is used to apply the $\oplus$ operator in order to combine the sequences of actual parameters. Each actual parameter is checked to see if it is also a formal parameter of the calling subprogram S. If an actual parameter is also a formal parameter, then either PBOI Case 1 or PBOI Case 2 applies. If the actual parameter is not a formal parameter then either PBOI Case 3 or PBOI Case 4 applies.

6.9 Transforming Subprogram Calls

This section defines the formal transformations used to convert imperative subprogram calls to object-oriented messages. These transformations are used when converting Category 3 subprograms. Let υ_m be the transformation that converts subprogram calls to messages. In order to transform a subprogram call l into a message m, it is assumed that the method to be invoked by m already exists in some class C. Let $instance(c, C)$ represent whether or not an object c is an instance of class C. The υ_m transformation is defined as shown in Figure 164. This transformation ensures that each of the actual parameters in the subprogram call l is either an attribute of the target object c, an attribute of one of the other objects passed to the method, or a formal parameter of the method. The target object, each of the other objects required, and the formal parameters are built as actual parameters of the message m.

$$
\begin{array}{l}
v_m(S,l) \;=\; m \;\text{ where} \\
\quad l \;=<id_{S_n}, P_{act}>\;\; and \\
\quad (\exists\; C_2 \in \rho^*(S) \;\text{ such that} \\
\quad\quad C_2 \;=<id_{C_2}, \Phi_{C_2}, \Omega_{C_2}, \lambda_0>\;\; and \\
\quad\quad \Omega_{C_2} \;=\; \{<id_{S_n}, \tau_2, Q_{obj2}, Q_{form_2}, Q_{ret_2}, Q_{loc_2}, \Psi_{C_2}>\}\;)and \\
\quad instance(c, C_2)\;\; and \\
\quad \forall\; p \in P_{act} \\
\quad\quad [a' \;=\; \theta_v(\mu^*(p))\; and\; a' \in \Phi_{C_2}]\;\; or \\
\quad\quad [\exists\; C_3 \in \rho^*(S) \;\text{ such that} \\
\quad\quad\quad C_3 \;=<id_{C_3}, \Phi_{C_3}, \Omega_{C_3}, \lambda_0>\;\; and \\
\quad\quad\quad b' \;=\; \theta_v(\mu^*(p))\; and\; b' \in \Phi_{C_3}\;\; and \\
\quad\quad\quad instance(q, C_3)\;\; and \\
\quad\quad\quad q \in Q_{obj2}\;\; and\;\; q \in Q_{obj}\;\; and\;\; q \in Q'_{obj}]\;\; or \\
\quad\quad [d' \;=\; \theta_v(\mu^*(p))\; and\; d' \in Q_{form_2}\;\; and \\
\quad\quad\quad p' \;=\; \theta_v(p)\; and\; p' \in Q'_{form}]\;\; and \\
\quad Q'_{act} \;=\; [c] \oplus Q'_{obj} \oplus Q'_{form}\;\; and \\
\quad m \;=<id_{S_n}, Q'_{act}>
\end{array}
$$

Figure 164 The v_m transformation

The v_e transformation, shown in Figure 165, is defined to use the v_m transformation in order to transform each function call in an imperative expression. This transformation uses the formal definition for each imperative expression in its definition.

In order to transform subprogram calls in each statement and expression, the v transformation has been defined as shown in Figure 166. This transformation uses the formal definitions of imperative statements in its definition. Recall that $pos(s, \Psi)$ indicates the ordinal position of a statement, s, in the sequence, Ψ. This transformation takes as input a sequence of imperative statements and object-oriented statements. The resulting sequence is a sequence of object-oriented statements where the subprogram calls from the imperative statements and expressions have been replaced by messages.

Let T_l^v be the formal transformation that takes an object-oriented design, a subprogram, and a class and replaces all the subprogram calls in the statements of the class' method with messages. This transformation is defined as follows.

$$\begin{aligned}
T_l^v(OOD, S, C) &= OOD' \text{ where} \\
C &= < id_C, \Phi_C, \Omega_C, \lambda_0 > \ and \\
\Omega_C &= \{< id_o, \tau, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi_o >\} \ and \\
\Psi'_o &= v(S,\ \Psi_o) \ and \\
\Omega'_C &= \{< id_o, \tau, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi'_o >\} \ and \\
C' &= < id_C, \Phi_C, \Omega'_C, \lambda_0 > \ and \\
OOD' &= T_c^+(T_c^-(OOD, C), C')
\end{aligned}$$

Each subprogram call in Ψ_o is replaced by corresponding message by using the v transformation. These updated statements are used to update the method in Ω'_C in the updated class C'. This class replaces the class C in the design and the updated design is returned as the result of the transformation.

For example, Figure 167 shows the class `CLASS-3` containing the method `RHO`. The original imperative subprogram call to the subprogram `RHO` is given below.

```
RHO ( RHOSTD, XLAMDA, ANGFAC)
```

Let `C-7` be an instance of `CLASS-3`. The subprogram call is transformed into the following message.

$v_e(S, e) \ = \ e'$ where

$e \ = < id_{S_n}, P_{act} > \Rightarrow e' \ = \ v_m(S, e) \ \mathit{and}$

gom-variable(e) $\Rightarrow \ e' \ = \ e \ \mathit{and}$

$e \ = <$ GET-a, $[c] > \Rightarrow e' \ = \ e \ \mathit{and}$

gom-literal-boolean(e) $\Rightarrow e' \ = \ e \ \mathit{and}$

gom-literal-integer(e) $\Rightarrow e' \ = \ e \ \mathit{and}$

gom-literal-real(e) $\Rightarrow e' \ = \ e \ \mathit{and}$

gom-literal-string(e) $\Rightarrow e' \ = \ e \ \mathit{and}$

$e \ = < e_1, \ +, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), \ +, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, \ \mathrm{and}, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), \ \mathrm{and}, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, \ \&, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), \ \&, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, \ /, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), \ /, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, \ =, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), \ =, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, \ **, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), \ **, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, >=, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), >=, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, >, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), >, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, <=, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), <=, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, <, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), <, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, \ *, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), \ *, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, <>, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), <>, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, \ \mathrm{or}, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), \ \mathrm{or}, v_e(S, e_2) > \ \mathit{and}$

$e \ = < e_1, -, e_2 > \Rightarrow \ e' \ = < v_e(S, e_1), -, v_e(S, e_2) > \ \mathit{and}$

$e \ = < -, e_1 > \Rightarrow \ e' \ = < -, v_e(S, e_1) > \ \mathit{and}$

$e \ = < \ \mathrm{not}, e_1 > \Rightarrow \ e' \ = < \ \mathrm{not}, v_e(S, e_1) >$

Figure 165 The v_e transformation

$$
\begin{aligned}
v(S,\ \Psi) \ &= \ \Psi' \ where \\
&\forall\, s \in \Psi \\
&\quad s \ = <id_{S_n}, P_{act}> \Rightarrow \\
&\qquad s' \ = \ v_m(S,\ s) \ and \\
&\quad s \ = <x,\ :=, e> \Rightarrow \\
&\qquad s' \ = <x,\ :=, v_e(S,e)> \ and \\
&\quad s \ = <\text{if}, e,\ \text{then}, S_1,\ \text{else}, S_2> \Rightarrow \\
&\qquad s' \ = <\text{if}, v_e(S,e),\ \text{then}, v(S,S_1),\ \text{else}, v(S,S_2)> \ and \\
&\quad s \ = <\text{while}, e, S_3> \Rightarrow \\
&\qquad s' \ = <\text{while}, v_e(S,e), v(S,S_3)> \ and \\
&\quad s \ = <\text{output},\ \text{oport}, E> \Rightarrow \\
&\qquad s' \ = <\text{output},\ \text{oport}, E'> \ and \\
&\qquad\quad E' \ = \ [v_e(S,e) \mid e \in E] \ and \\
&\quad s' \ \in \Psi' \ and \\
&\quad pos(s,\Psi) \ = \ pos(s',\Psi')
\end{aligned}
$$

Figure 166 The v transformation

```
RHO ( C-7, RHOSTD)
```

using the T_l^v transformation. The RHOSTD actual parameter is passed as a parameter of the message, the XLAMDA actual parameter has become the LAMBDA attribute of CLASS-3, and the actual parameter ANGFAC has also become an attribute of CLASS-3.

When duplicate classes are eliminated from the object-oriented design using the PBOI methodology, the signature of methods in the resulting class may be changed. The signatures of any messages that call these methods must be updated to match. Let the

```
class CLASS-3 attributes LAMBDA, ANGFAC
  method RHO ( C-3, RHOSTD ): real
    begin
      RHO := RHOSTD
    end
  superclass USER-OBJECT
```

Figure 167 Class CLASS-3 with Method RHO

$$
\begin{array}{l}
T_o^v(OOD, C) \;=\; OOD' \text{ where} \\
\quad C \;= < id_C, \Phi_C, \Omega_C, \lambda_0 > \quad \text{where} \\
\quad \Omega_C \;=\; \{< id_{S_n}, \tau, Q_{obj}, Q_{form}, Q_{ret}, Q_{loc}, \Psi_o >\} \; and \\
\quad \forall C_1 \in OOD \\
\qquad C_1 \;= < id_{C_1}, \Phi_{C_1}, \Omega_{C_1}, \lambda_0 > \; and \\
\qquad \Omega_{C_1} \;=\; \{< id_o, \tau, Q_{obj1}, Q_{form1}, Q_{ret1}, Q_{loc1}, \Psi_{o1} >\} \; and \\
\qquad \text{For each } m \; \in C_{sub_o} \\
\qquad\quad m \;= < id_{S_n}, Q_{act} > \Rightarrow \\
\qquad\qquad v_s(Q'_{act}, Q_{form}) \; and \\
\qquad\qquad m' \;= < id_{S_n}, Q'_{act} > \; and \; m' \in C'_{sub_o} \; and \\
\qquad\quad m \;\neq < id_{S_n}, Q_{act} > \Rightarrow \\
\qquad\qquad m' \;=\; m \; and \; m' \in C'_{sub_o} \; and \\
\qquad \Psi'_{o1} \;=\; \Psi_{o1} \text{ with } m \; \in C_{sub_o} \; \text{ replaced by } m' \; \in C'_{sub_o} \; and \\
\qquad \Omega'_{C_1} \;=\; \{< id_o, \tau, Q_{obj1}, Q_{form1}, Q_{ret1}, Q_{loc1}, \Psi'_{o1} >\} \; and \\
\qquad C'_1 \;= < id_{C_1}, \Phi_{C_1}, \Omega'_{C_1}, \lambda_0 > \; and \\
\qquad C_1 \notin OOD' \; and \; C'_1 \in OOD'
\end{array}
$$

Figure 168 The T_o^v transformation

T_o^v transformation represent this updating of message signatures as defined below. Let OOD represent an object-oriented design, let C_{msg_o} represent the collection of messages in a method. Let $v_s(Q'_{act}, Q_{form})$ represent whether or not the number, order, and type of the actual parameters in Q'_{act} match the number, order, and type of the formal parameters in Q_{form}. The T_o^v transformation is shown in Figure 168. This transformation checks each method in each class in OOD for messages being sent to the method in C. If the call identifier of the message m matches the name of the method in C, then the v_s condition is used to ensure the signature of m matches the formals in Q_{form}. The updated message, m', replaces m in C'_1 and this updated class replaces C_1 in OOD. If the call identifier of m is not referring to the method in C, then no update is required and the original message, m, is stored in C'_1.

6.10 Category 3 Subprogram Transformation (σ_3)

This section defines the formal transformation used to convert Category 3 subprograms to the object paradigm. As discussed in Section 5.3, the transformation of Category 3 subprograms to the object paradigm uses the PBOI method to update the object design developed so far.

At this point of the transformation, there may be several classes in the design that have been built for the same subprogram. This is because a subprogram may be called multiple times and a new class is built for the subprogram for each of the calls. These classes built for the same subprogram are termed *duplicate* classes because they have the same name. Duplicate classes are eliminated from the design by merging any duplicate classes into a new class that replaces the duplicates throughout the entire design. The set of attributes for the new class is the *intersection* of the sets of attributes from the duplicate classes. The set of operations for the new class is the *union* of the sets of operations from the duplicate classes. The formal transformation that defines the removal of duplicate classes is defined below.

Let T_{class}^{dup} represent the formal transformation that removes duplicate classes from the object-oriented design. Let OOD represent an object-oriented design. The T_{class}^{dup} transformation is defined as shown in Figure 169.

This transformation examines all pairs of classes in OOD. Duplicate classes are identified by comparing the name of the class. This implies the generated name of a class must be the same each time a class is built for a subprogram. If the names of two classes C_1 and C_2 are the same, then a new class C_3 is built to represent the intersection of these two classes. The attributes of C_3 are built by initially unioning Φ_{C_1} and Φ_{C_2} and then using the $T_A^{\delta-1}$ transformation to remove those attributes not in the intersection of Φ_{C_1} and Φ_{C_2}. The operations of C_3 are built by unioning Ω_{C_1} and Ω_{C_2}. The union is based on the fact that two operations are equal if they have the same name. The C_3 class replaces the two duplicate classes in OOD'. The final step of the transformation is to ensure any calls to the updated method in C_3 match the signature of the method.

$$
\begin{aligned}
& T^{dup}_{class}(OOD) \; = \; OOD'' \text{ where} \\
& \quad \forall C_1 \; and \; C_2 \in OOD \\
& \qquad C_1 \; \neq \; C_2 \; and \\
& \qquad C_1 \; = < id_C, \Phi_{C_1}, \Omega_{C_1}, \lambda_0 > \; and \\
& \qquad C_2 \; = < id_C, \Phi_{C_2}, \Omega_{C_2}, \lambda_0 > \Rightarrow \\
& \qquad\quad C_3 \; = < id_C, \Phi_{C_3}, \Omega_{C_3}, \lambda_0 > \; and \\
& \qquad\qquad \Phi_{C_3} \; = \; \Phi_{C_1} \; \cup \; \Phi_{C_2} \; and \\
& \qquad\qquad \Omega_{C_3} \; = \; \Omega_{C_1} \; \cup \; \Omega_{C_2} \; and \\
& \qquad\quad A \; = \; (\Phi_{C_1} \cup \Phi_{C_2}) - (\Phi_{C_1} \cap \Phi_{C_2}) \; and \\
& \qquad\quad C_3' \; = \; T_A^{\delta^{-1}}(OOD, C_3, A) \; and \\
& \qquad\quad C_1 \notin OOD' \; and \; C_2 \notin OOD' \; and \\
& \qquad\quad C_3' \in OOD' \; and \\
& \qquad\quad OOD'' \; = \; T_o^v(OOD', C_3')
\end{aligned}
$$

Figure 169 The T^{dup}_{class} transformation

Now that the transformation for eliminating duplicate classes has been defined, the transformation of Category 3 subprograms is defined as follows. Let OOD represent an object-oriented design, let S represent the Category 3 subprogram to be transformed. Let OOD' represent the updated object-oriented design that results from converting S to the object paradigm. Let P_{form} be the sequence of formal parameters from the subprogram S. Let C_{subs_S} be the set of imperative subprogram calls in S. The formal transformation for Category 3 subprograms is defined as shown in Figure 170. This transformation converts each of the subprograms called by S and unions all the designs that are returned. This is done by applying the σ transformation to each of the subprograms in $C^*_{subs_S}$. Initially, a class is built to represent the Category 3 subprogram by building all the formal parameters of the subprogram as attributes of the class. The subprogram S is initially transformed into a method included in Ω_C. Each statement in the subprogram is transformed into an object-oriented statement by θ and accesses to parameters of the subprogram are transformed by δ into attribute accesses of the instance object τ. This initial class is added to the design built so far, viz. OOD', and the resulting design is passed to the PBOI transformation along with S. Since the PBOI transformation has the potential to change the class C,

$$
\begin{aligned}
\sigma_3(OOD, S) \; &= \; OOD^{(4)} \text{ where} \\
&S \;=< id_S, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma_S > \; and \\
&P_{form} \;=\; P_{in} \oplus P_{out} \; and \\
&\forall\, l \in C_{subs_S} \\
&\quad l \;=< id_{S_n}, P_{act_n} > \; and \\
&\quad S_n \in C^*_{subs_S} \\
&OOD' \;=\; \bigcup_{i=1}^{n}(\sigma(OOD, S_i)) \;\text{ where } S_i \in C^*_{subs_S} \; and \\
&C \;=< id_C, \Phi_C, \Omega_C, \lambda_0 > \; and \\
&\Phi_C \;=\; range(\theta_v(P_{form})) \; and \\
&\tau \text{ is an instance of } C \; and \\
&\Psi \;=\; \delta(\theta(\Sigma_S), \tau, \Phi_C) \; and \\
&Q_{ret} \;=\; \theta_v(P_{ret}) \; and \\
&Q_{loc} \;=\; \theta_v(P_{loc}) \; and \\
&\Omega_C \;=\; \{< id_S, \tau, \emptyset, \emptyset, Q_{ret}, Q_{loc}, \Psi >\} \; and \\
&OOD'' \;=\; T_{pboi}(T_c^+(OOD', C), S) \; and \\
&C' \;=\; \text{updated } C \in OOD'' \; and \\
&OOD''' \;=\; T_l^v(OOD'', S, C') \; and \\
&OOD^{(4)} \;=\; T_{class}^{dup}(OOD''')
\end{aligned}
$$

Figure 170 The σ_3 transformation

```
procedure RELAY-EFFNCY
  ( BETA, XLAMDA, DIAM, UPLFAC, SIGJ, ZENITH, MIRRORS, GEOM,
    AXMAJ, AXMIN, SIGJR, EFFNCY )
begin
  ...
  if MIRRORS > 0
    then RNGFAC := 0.0;
      K := 1;
      while K <= MIRRORS do begin
        ANGFAC := GEOM ( 1, K);
        RNGFAC := RNGFAC + GEOM ( 2, K);
        TRANS := TRANS
            * BOUNCE ( ANGFAC, RNGFAC, PROJRA, SIGB,
                       RANGE, AXMAJ, AXMIN, XLAMDA);
        K := K + 1
        end
   else endif;
  ANGFAC := GEOM ( 1, MIRRORS + 1);
  EFFNCY := TRANS
  end
```

Figure 171   Partial declaration of RELAY-EFFNCY

the class C' represents this updated class after the PBOI transformation. This updated class C' is passed to the T_l^v transformation in order to transform the subprogram calls in C' to messages. The class returned from the T_l^v transformation replaces C' in the design. The T_{class}^{dup} transformation is used to remove duplicate classes and the resulting design is returned as the result of the transformation.

For example, Figure 171 shows the partial declaration of the imperative subprogram RELAY-EFFNCY (using GIL syntax). This subprogram is a Category 3 subprogram that calls the BOUNCE subprogram. The first step in converting RELAY-EFFNCY to the object paradigm is to transform the BOUNCE subprogram and store the resulting design in OOD'. Next, an initial class C is built for the RELAY-EFFNCY subprogram and added to this design. The partial declaration of this class is shown in Figure 172. This class has an attribute built from each of the formal parameters of the RELAY-EFFNCY subprogram. The operations for this class include one method built to implement RELAY-EFFNCY. For brevity,

```
class CLASS-9 attributes  BETA, XLAMDA, DIAM, UPLFAC,
    SIGJ, ZENITH, MIRRORS, GEOM, AXMAJ, AXMIN, SIGJR, EFFNCY
  method RELAY-EFFNCY ( C-39)
    begin
    ...
    end
  superclass USER-OBJECT
```

Figure 172   Classes after PBOI Transformation

the statements of this method are not shown. The class CLASS-9 is added to the design OOD' and the resulting design is passed to the T_{pboi} transformation.

6.11 Transforming Category 1 Subprograms (σ_1)

This section defines the formal transformation for converting Category 1 subprograms to the object-oriented paradigm. As explained in Section 5.8, the main program is typically classified as a Category 1 subprogram. Category 1 subprograms other than the main program are transformed to the object paradigm using the Category 3 transformation, σ_3.

More formally, let OOD represent an object-oriented design, let S be an imperative subprogram, and let M be the main program. Recall from Section 5.8 that the main program for a system of imperative subprograms is identified by the user. The transformation of any Category 1 subprogram is defined as follows.

$$\begin{aligned}\sigma_1(OOD,S) &= OOD' \text{ where}\\ S \neq M \Rightarrow OOD' &= \sigma_3(OOD,S) \textit{ and}\\ S = M \Rightarrow OOD' &= \sigma_M(OOD,S)\end{aligned}$$

This transformation determines whether the input subprogram is the main program or not. If it is not the main program, then the σ_3 transformation is used. If it is the main program, then the σ_M transformation specifically defined for the main program is used.

The rationale for the transformation of the main program is given in Section 5.8. Specifically, there are three updates done on the design while transforming the main program. First, duplicate object instances are replaced in the main program with a single object instance. Second, the "GET-", "SET-", and "CREATE-" methods are added to each class in the design. Finally, overlapping classes in the design are merged into a new class and instances of the overlapping classes are replaced with a single instance of the new class. The transformations that formalize these updates are defined below and an example of each transformation is provided.

6.11.1 Removing duplicate objects. This section defines the transformations needed to remove *duplicate objects* from the main program. Two duplicate objects are instances of the same class that are built using the same data items. Duplicate objects are eliminated by replacing the two assignment statements that are used to build the instances with a new assignment statement that builds just one instance. In order to replace the two variables that hold the duplicate objects, the v_e^{dup} and v^{dup} transformations are defined in this section.

The v_e^{dup} transformation replaces any occurrences of the v_1 variable in an expression e with the v_2 variable, which results in a new expression e. Let the $pos(v, P)$ predicate indicate the ordinal position of a variable, v, in the sequence, P. The v_e^{dup} transformation is defined as shown in Figure 173.

The v^{dup} transformation has been defined as shown in Figure 174. This transformation uses the formal definitions of imperative statements in its definition.

Let T_{obj}^{dup} represent the transformation that removes duplicate object instances from the object-oriented design. Let OOD represent the object-oriented design, let C represent the overall system class built for the main program, and let id_{main} be the name of the main program. The T_{obj}^{dup} transformation is defined as shown in Figure 175. This transformation compares the assignment statements in the method built for the main program in order to find "CREATE-" messages that are instantiating duplicate objects. The transformation replaces the i_1 and i_2 assignment statements with the i_3 assignment statement and the variables v_1 and v_2 with the variable v_3.

For example, Figure 176 shows the overall system class `CLASS-SYSTEM`. The variables `C-5` and `C-6` hold object instances that are passed to the `BOUNCE` method. Both object instances are built from `CLASS-2` using the data items `ANGFAC` and `BEAMRA` and are duplicate

$v_e^{dup}(e, v_1, v_2) \; = \; e'$ where

$e \; = < id_{S_n}, P_{act} > \; and \; v_1 \in P_{act} \Rightarrow$

$e' \; = < id_{S_n}, P'_{act} > \; and$

$v_1 \notin P'_{act} \; and \; v_2 \in P'_{act} \; and$

$pos(v_1, \; P'_{act}) \; = \; pos(v_2, \; P'_{act}) \; and$

gom-variable(e) $ande \; = \; v_1 \; \Rightarrow$

$e' \; = \; v_2 \; and$

gom-literal-boolean(e) $\Rightarrow e' \; = \; e \; and$

gom-literal-integer(e) $\Rightarrow e' \; = \; e \; and$

gom-literal-real(e) $\Rightarrow e' \; = \; e \; and$

gom-literal-string(e) $\Rightarrow e' \; = \; e \; and$

$e \; = < e_1, \; +, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), \; +, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, \; \text{and}, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), \; \text{and}, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, \; \&, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), \; \&, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, \; /, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), \; /, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, \; =, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), \; =, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, \; **, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), \; **, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, >=, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), >=, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, >, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), >, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, <=, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), <=, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, <, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), <, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, \; *, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), \; *, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, <>, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), <>, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, \; \text{or}, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), \; \text{or}, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < e_1, -, e_2 > \Rightarrow \; e' \; = < v_e^{dup}(e_1, v_1, v_2), -, v_e^{dup}(e_2, v_1, v_2) > \; and$

$e \; = < -, e_1 > \Rightarrow \; e' \; = < -, v_e^{dup}(e_1, v_1, v_2) > \; and$

$e \; = < \; \text{not}, e_1 > \Rightarrow \; e' \; = < \; \text{not}, v_e^{dup}(e_1, v_1, v_2) >$

Figure 173 The v_e^{dup} transformation

$$
\begin{array}{l}
v^{dup}(\Psi, v_1, v_2) \ = \ \Psi' \ where \\
\quad \forall\, s \in \Psi \\
\qquad s \ = < id_{S_n}, P_{act} > \ and \ v_1 \in P_{act} \ \Rightarrow \\
\qquad\quad s' \ = < id_{S_n}, P'_{act} > \ and \\
\qquad\quad v_1 \notin P'_{act} \ and \ v_2 \in P'_{act} \ and \\
\qquad\quad pos(v_1, \ P'_{act}) \ = \ pos(v_2, \ P'_{act}) \ and \\
\qquad s \ = < x, \ :=, e > \Rightarrow \\
\qquad\quad s' \ = < x, \ :=, v_e^{dup}(e, v_1, v_2) > \ and \\
\qquad s \ = < \text{if}, e, \ \text{then}, S_1, \ \text{else}, S_2 > \Rightarrow \\
\qquad\quad s' \ = < \text{if}, v_e^{dup}(e, v_1, v_2), \ \text{then}, v^{dup}(S_1, v_1, v_2), \ \text{else}, v^{dup}(S_2, v_1, v_2) > \ and \\
\qquad s \ = < \text{while}, e, S_3 > \Rightarrow \\
\qquad\quad s' \ = < \text{while}, v_e^{dup}(e, v_1, v_2), v^{dup}(S_3, v_1, v_2) > \ and \\
\qquad s \ = < \text{output}, \ \text{oport}, E > \Rightarrow \\
\qquad\quad s' \ = < \text{output}, \ \text{oport}, E' > \ and \\
\qquad\qquad E' \ = \ [v_e^{dup}(e, v_1, v_2) \mid e \in E] \ and \\
\qquad s' \ \in \Psi' \ and \\
\qquad pos(s, \Psi) \ = \ pos(s', \Psi')
\end{array}
$$

Figure 174 The v^{dup} transformation

$$
\begin{array}{l}
T_{obj}^{dup}(OOD, C) \ = \ OOD' \ \text{where} \\
\quad C \ = < \text{CLASS-SYSTEM}, \Phi_C, \Omega_C, \lambda_0 > \ and \\
\quad \Omega_C \ = \ \{< id_{main}, \tau, \emptyset, \emptyset, Q_{ret}, Q_{loc}, \Psi >\} \ and \\
\quad \forall\, i_1 \ and \ i_2 \in \Psi \ and \\
\qquad [i_1 \ = < v_1, \ :=, < CREATE - C_1, [d_1, d_2, \ldots, d_n] > > \ and \\
\qquad i_2 \ = < v_2, \ :=, < CREATE - C_1, [d_1, d_2, \ldots, d_n] > > \ ] \Rightarrow \\
\qquad\quad i_3 \ = < v_3, \ :=, < CREATE - C_1, [d_1, d_2, \ldots, d_n] > > \ and \\
\qquad\quad i_1 \notin \Psi' \ and \ i_2 \notin \Psi' \ and \ i_3 \in \Psi \ and \\
\qquad\quad pos(i_1, \Psi) \ = \ pos(i_3, \Psi') \ and \\
\qquad\quad \Psi'' \ = \ v^{dup}(\Psi, v_1, v_3) \ and \\
\qquad\quad \Psi''' \ = \ v^{dup}(\Psi, v_2, v_3) \ and \\
\quad \Omega'_C \ = \ \{< id_{main}, \tau, \emptyset, \emptyset, Q_{ret}, Q_{loc}, \Psi''' >\} \ and \\
\quad C' \ = < \text{CLASS-SYSTEM}, \Phi_C, \Omega'_C, \lambda_0 > \ and \\
\quad OOD' \ = \ T_c^{+}(T_c^{-}(OOD, C), C')
\end{array}
$$

Figure 175 The T_{obj}^{dup} transformation

```
class CLASS-2 attributes ANGFAC, BEAMRA
  method CREATE-CLASS-2
      ( A-ANGFAC, A-BEAMRA ): a CLASS-2
    begin
    ...
    end

class CLASS-SYSTEM attributes
  method BMDSIM1 ( C-9 )
    begin
    ...
    C-5 := CREATE-CLASS-2 ( ANGFAC, BEAMRA );
    C-6 := CREATE-CLASS-2 ( ANGFAC, BEAMRA );
    BOUNCE ( C-4, C-5, C-6 );
    ...
    end
```

Figure 176 Duplicate object instances

object instances. Figure 177 shows that the method BMDSIM1 has been updated by replacing

```
class CLASS-SYSTEM attributes
  method BMDSIM1 ( C-9 )
    begin
    ...
    C-7 := CREATE-CLASS-2 ( ANGFAC, BEAMRA );
    BOUNCE ( C-4, C-7, C-7 );
    ...
    end
```

Figure 177 Duplicates removed

the duplicate object instances. The variables C-5 and C-6 have been replaced with the variable C-7. The new instance is instantiated using the CREATE-CLASS-2 method, stored in C-7, and passed to the BOUNCE method in place of C-5 and C-6.

6.11.2 Generating Methods (κ). As part of the transformation to merge overlapping classes, the "SET-", "GET-", and "CREATE-" methods are added to each class

```
class CLASS-17 attributes DWELLT
  method CREATE-CLASS-17 ( A-DWELLT ): a CLASS-17
    begin
    INST-CLASS-17 := new ( CLASS-17 );
    SET-DWELLT ( INST-CLASS-17, A-DWELLT );
    CREATE-CLASS-17 := INST-CLASS-17
    end
  method GET-DWELLT ( C-10 ): real
    begin
    GET-DWELLT := C-10.DWELLT
    end
  method SET-DWELLT ( C-11, VAL-10 )
    begin
    C-11.DWELLT := VAL-10
    end
```

Figure 178 Instantiation Method for CLASS-17

in the object-oriented design. This section defines a formal transformation that generates these methods.

Figure 178 shows the "SET-", "GET-", and "CREATE-" methods for the class `CLASS-17`. The mechanics of "SET-" and "GET-" methods were covered in Section 6.3, so they won't be covered further here. The "CREATE-" method for this class is explained in detail.

The `CLASS-17` class has only one attribute `DWELLT`, so the "CREATE-" method for this class expects one parameter that is used to initialize the attribute. The "CREATE-" method includes a call to the intrinsic functional method named `new`, which returns a new instance of the class. The `INST-CLASS-17` local variable holds the newly instantiated object and is sent the "SET-" method to assign the input value of the `DWELLT` attribute. Finally, the new initialized instance is returned as the result of this method call.

Let κ be a transformation that adds the "SET-", "GET-", and "CREATE-" methods to a class. Let C be a class, and let τ be the target object, which is an instance of C. Let β be an expression that will be passed as the new value in "SET-" methods. Let `SET-a` represent a symbol that results from prepending "SET-" to the name of the attribute

```
$\kappa(C) \;=\; C'$ where
    $C \;= <id_C, \Phi_C, \Omega_C, \lambda> \;\; and$
    For each $a \in \Phi_C$
        $s \;= <\text{SET-a}, \tau, [\,], [\beta], [\,], [\,], [\,], \Pi> \;\; and$
        $\Pi \;=\; [<\tau.a,\; :=, \beta>] \; and$
        $g \;= <\text{GET-a}, \tau, [\,], [\,], [\,], [a], [\,], \Psi> \;\; and$
        $\Psi \;=\; [<\text{GET-a},\; :=, \tau.a>] \; and$
        $s \in \Omega'_C \; and \; g \in \Omega'_C$
    For each $a \in \Phi_C$
        $\text{A-a} \;\in \chi \; and$
        $<\text{SET-a}, [\;\text{INST-}id_C,\;\; \text{A-a}\;]> \;\in \psi \; and$
    $\Xi \;=\; [<\text{INST-}id_C,\; :=,\; \text{new}(id_C)>,$
            $\psi,$
            $<\text{CREATE-}id_C,\; :=,\; \text{INST-}id_C>] \; and$
    $r \;= <\text{CREATE-}id_C, \tau, [\,], \chi, [\text{INST-}id_C], [\text{INST-}id_C], \Xi> \;\; and$
    $r \in \Omega'_C \; and$
    $C' \;= <id_C, \Phi_C, \Omega'_C, \lambda>$
```

Figure 179 The κ transformation

$a \in \Phi_C$. Similarly, let GET-a represent the symbol resulting from the prepending of "GET-" to the name of a. Let CREATE-id_C represent the symbol that results from prepending "CREATE-" to the name of the class C. Let A-a represent the symbol resulting from prepending "A-" to the name of a. This symbol is used for formal parameters in the "CREATE-" method. Let χ represent the sequence of these formal parameters. Let INST-id_{C} represent the symbol resulting from prepending "INST-" to the name of the class C. Let ψ represent the sequence of "SET-" messages used to assign the input values to the object attributes. The κ transformation is defined as shown in Figure 179.

All the methods shown in Figure 178 are generated by the transformation

$$\kappa(\text{CLASS-17 })$$

$\kappa^{-1}(C) \;=\; C'$ where

$C \;= <id_C, \Phi_C, \Omega_C, \lambda> \; and$

For each $a \in \Phi_C$

$s \;= <\text{SET-a}, \tau, [\,], [\beta], [\,], [\,], [\,], \Pi> \; and$

$g \;= <\text{GET-a}, \tau, [\,], [\,], [\,], [a], [\,], \Psi> \; and$

$s \notin \Omega'_C \; and \; g \notin \Omega'_C \; and$

$r \;= <\text{CREATE-}id_C, \tau, [\,], \chi, [\text{INST-}id_C], [\text{INST-}id_C], \Xi> \; and$

$r \notin \Omega'_C \; and$

$C' \;= <id_C, \Phi_C, \Omega'_C, \lambda>$

Figure 180 The κ^{-1} transformation

The κ transformation returns a new class C' that must replace the old class C in the object design. This updating of the object design is not shown here.

As part of the PBOI method, it is in some cases required to remove the "SET-", "GET-", and "CREATE-" methods from a class. For example, when merging two classes. For this reason, the κ transformation is reversible. The "inverse" transformation for κ is defined below.

Let κ^{-1} be the inverse of κ. The κ transformation is defined as shown in Figure 180. All the methods shown in Figure 178 are removed by the transformation

$$\kappa^{-1}(\text{CLASS-17 })$$

As with κ, the κ^{-1} transformation returns a new class C' that must replace the old class C in the object design. This updating of the object design is not shown.

6.11.3 Merging overlapping classes. The T^{merge}_{class} transformation is used in the main program to merge overlapping classes. Let OOD represent the design built so far and let C represent the class built for the main program. Let $set(Q)$ represent the conversion of the sequence Q to a set. Let Q_{act} represent a sequence of actual parameters and let Q_{form} represent a sequence of formal parameters. Let $v_s(Q_{act}, Q_{form})$ indicate the number, order, and type of the parameters in Q_{act} matches the number, order, and type of the parameters in Q_{form}. The T_{class}^{merge} transformation is defined as shown in Figure 181. This transformation examines each assignment statement in the method built for the main program in order to find "CREATE-" messages that are building instances of overlapping classes. The classes from the "CREATE-" messages in the i_1 and i_2 assignment statements are merged to form the new class C_3. The overlapping classes C_1 and C_2 do not appear in the updated design, but the new class does. The actual parameters for the "CREATE-" message in the new assignment statement i_3 are built from the actual parameters in the "CREATE-" messages in i_1 and i_2. The v predicate ensures that this new sequence of actual parameters matches the number, order, and type of the formal parameters in the "CREATE-" method of the new class C_3. The i_1 and i_2 assignment statements are replaced by the new assignment statement i_3 and the variables v_1 and v_2 are replaced by the variable v_3. The updated system class replaces the original system class in the OOD' design and the resulting design is returned as the result of the transformation.

For example, Figure 182 shows the classes `CLASS-2`, `CLASS-3`, and `CLASS-SYSTEM`. In the `BMDSIM1` method, the variable `C-5` holds an instance of `CLASS-2` that is built using the data items `ANGFAC` and `BEAMRA`. The variable `C-8` holds an instance of `CLASS-3` that is built using the data items `ANGFAC` and `RHOSTD`. Since the sequence of actuals have the `ANGFAC` data item in common, the classes `CLASS-2` and `CLASS-3` overlap and are merged into one new class.

Figure 183 shows the result of merging `CLASS-2` and `CLASS-3` into the new class `CLASS-10`. In the `BMDSIM1` method, the variable `C-11` holds the new instance instantiated from `CLASS-10`. The `C-11` variable is passed to the `BOUNCE` message in place of the variables `C-5` and `C-8`. The design is updated by removing `CLASS-2` and `CLASS-3` and adding `CLASS-10`.

6.11.4 Converting the Main Program.

This section defines the formal transformation required to convert the main program. In the main program, as the calls to other subprograms are transformed into messages, the object instances required by the signature of the method being invoked must be built. For each message being built, it is assumed that the corresponding method exists in some class in the design.

$T_{class}^{merge}(OOD, C) \; = \; OOD''$ where

$C \; = <\text{CLASS-SYSTEM}, \emptyset, \Omega_C, \lambda_0 > \; and$

$\Omega_C \; = \; \{< id_{main}, \tau, \emptyset, \emptyset, \emptyset, Q_{loc}, \Psi >\} \; and$

$i_1 \; and \; i_2 \in \Psi \; and$

$pos(i_1, \Psi) \; < \; pos(i_2, \Psi) \; and$

$i_1 \; = < v_1, \; :=, < CREATE - C_1, Q_{act_1} > > \; and$

$i_2 \; = < v_2, \; :=, < CREATE - C_2, Q_{act_2} > > \; and$

$set(Q_{act_1}) \; \cap \; set(Q_{act_2}) \; \neq \; \emptyset \; \Rightarrow$

$C_1 \in OOD \; and \; C_1' \; = \; \kappa^{-1}(C_1) \; and$

$C_2 \in OOD \; and \; C_2' \; = \; \kappa^{-1}(C_2) \; and$

$C_1' \; = < id_{C_1'}, \Phi_{C_1'}, \Omega_{C_1'}, \lambda_0 > \; and$

$C_2' \; = < id_{C_2'}, \Phi_{C_2'}, \Omega_{C_2'}, \lambda_0 > \; and$

$C_3 \; = < id_{C_3}, \Phi_{C_3}, \Omega_{C_3}, \lambda_0 > \; and$

$\Phi_{C_3} \; = \; \Phi_{C_1'} \; \cup \; \Phi_{C_2'} \; and$

$\Omega_{C_3} \; = \; \Omega_{C_1'} \; \cup \; \Omega_{C_2'} \; and$

$C_1 \notin OOD' \; and \; C_2 \notin OOD' \; and \; \kappa(C_3) \in OOD' \; and$

$i_3 \; = < v_3, \; :=, < CREATE - C_3, Q_{act_3} > >$ where

$Q_{act_3} \; = \; Q_{act_1} \; \oplus \; Q_{act_2} \; and$

$v_s(Q_{act_3}, Q_{form_{CREATE-C_3}}) \; and$

$i_1 \notin \Psi' \; and \; i_2 \notin \Psi' \; and \; i_3 \in \Psi \; and$

$pos(i_1, \Psi) \; = \; pos(i_3, \Psi') \; and$

$\Psi'' \; = \; v^{dup}(\Psi, v_1, v_3) \; and$

$\Psi''' \; = \; v^{dup}(\Psi, v_2, v_3) \; and$

$\Omega_C' \; = \; \{< id_{main}, \tau, \emptyset, \emptyset, Q_{ret}, Q_{loc}, \Psi''' >\} \; and$

$C' \; = <\text{CLASS-SYSTEM}, \Phi_C, \Omega_C', \lambda_0 > \; and$

$OOD'' \; = \; T_c^+(T_c^-(OOD', C), C')$

Figure 181 The T_{class}^{merge} transformation

```
class CLASS-2 attributes ANGFAC, BEAMRA
  method CREATE-CLASS-2
      ( A-ANGFAC, A-BEAMRA ): a CLASS-2
    begin
    ...
    end
  method CAPTURE ( C-2 ) : real
    begin
    ...
    end

class CLASS-3 attributes ANGFAC, RHOSTD
  method CREATE-CLASS-3
      ( A-ANGFAC, A-RHOSTD ): a CLASS-3
    begin
    ...
    end
  method RHO ( C-3 ) : real
    begin
    ...
    end


class CLASS-SYSTEM attributes
  method BMDSIM1 ( C-9 )
    begin
    ...
    C-5 := CREATE-CLASS-2 ( ANGFAC, BEAMRA );
    C-8 := CREATE-CLASS-3 ( ANGFAC, RHOSTD );
    BOUNCE ( C-4, C-5, C-8 );
    ...
    end
```

Figure 182   Overlapping object classes

```
class CLASS-10 attributes ANGFAC, BEAMRA, RHOSTD
  method CREATE-CLASS-10
      ( A-ANGFAC, A-BEAMRA, A-RHOSTD ): a CLASS-10
    begin
    ...
    end
  method CAPTURE ( C-10 ) : real
    begin
    ...
    end
  method RHO ( C-10 ) : real
    begin
    ...
    end

class CLASS-SYSTEM attributes
  method BMDSIM1 ( C-9 )
    begin
    ...
    C-11 :=
      CREATE-CLASS-10 ( ANGFAC, BEAMRA, RHOSTD );
    BOUNCE ( C-4, C-11, C-11 );
    ...
    end
```

Figure 183   Overlapping classes merged

The T_m^{as} transformation generates a sequence of assignment statements that instantiate the object instances required and stores them in variables used as actual parameters in the message. Let v_c represent a GOM variable generated to hold the target object instance. Let v_i represent one of the variables generated to hold the other object instances required for a specific message, m. Let the $v_s(Q_{act_1}, Q_{form_o})$ predicate indicate whether or not the number, order, and type of the actual parameters in Q_{act_1} match the number, order, and type of the parameters in Q_{form_o}. The T_m^{as} transformation is shown in Figure 184. This transformation builds an assignment statement, as_1, that includes a call to the "CREATE-" message from class C_2 in order to instantiate the target object c. The actual parameters to be sent to the "CREATE-" message are determined by using the μ^* transitive closure and the actual parameters from the subprogram call l. The assignment statement as_1 stores the instance into the variable v_c. For each object instance q in Q_{obj}, an assignment statement is built that includes a call to the "CREATE-" message from q's class C_i. The assignment statement stores the instance in the variable v_i.

For example, consider the following message from the main program.

```
BOUNCE ( C-4, C-5, C-8 );
```

This message is passing the C-4, C-5, and C-8 object instances to the BOUNCE method. The target object of the message is C-4, an instance of CLASS-4, which includes the BOUNCE method. For this example, assume that C-5 and C-8 are instances of CLASS-2 and CLASS-3, respectively. The original call to the subprogram BOUNCE is shown below.

```
BOUNCE ( XLAMDA, ANGFAC, BEAMRA, RHOSTD );
```

The sequence of statements generated to build the instances required for this message are shown below.

```
C-4 := CREATE-CLASS-4 ( XLAMDA );
C-5 := CREATE-CLASS-2 ( ANGFAC, BEAMRA );
C-8 := CREATE-CLASS-3 ( ANGFAC, RHOSTD );
```

The sequence of assignment statements returned from the T_m^{as} transformation must be spliced into the sequence of statements in the main program right before the message

$T^{as}_{m}(OOD, l, m) \ = \ AS$ where

$l \ = < id_S, P_{act} > \ and$

$m \ = < id_S, Q_{act} > \ and$

$Q_{act} \ = \ [c] \oplus Q_{obj} \oplus Q_{form} \ and$

$instance(c, C_2) \ and$

$C_2 \in OOD \ and$

$C_2 \ = < id_{C_2}, \Phi_{C_2}, \Omega_{C_2}, \lambda_0 > \ and$

$o \in \Omega_{C_2} \ and$

$o \ = \ \{< \text{CREATE-}id_{C_2}, \tau, [\,], Q_{form_o}, Q_{ret_o}, Q_{loc_o}, \Psi_o >\} \ and$

$Q_{act_1} \ = \ [\theta_v(d_k) \ | \ d_k \in P_{act} \ and$

$\quad \theta_v(\mu^*(d_k)) \in Q_{form_o} \ ] \ and$

$v_s(Q_{act_1}, Q_{form_o}) \ and$

$AS_1 \ = \ [< v_c, \ :=, < \text{CREATE-C}_2, Q_{act_1} > > \ ] \ and$

$\forall \ q \in Q_{obj}$

$\quad instance(q, C_i) \ and$

$\quad C_i \in OOD \ and$

$\quad C_i \ = < id_{C_i}, \Phi_{C_i}, \Omega_{C_i}, \lambda_0 > \ and$

$\quad o_i \in \Omega_{C_i} \ and$

$\quad o_i \ = \ \{< \text{CREATE-C}_i, \tau_{o_i}, [\,], Q_{form_{o_i}}, Q_{ret_{o_i}}, Q_{loc_{o_i}}, \Psi_{o_i} >\} \ and$

$\quad Q_{act_i} \ = \ [d_j \ | \ \mu^*(d_j) \in Q_{form_{o_i}} \ ] \ and$

$\quad v_s(Q_{act_i}, Q_{form_{o_i}}) \ and$

$\quad as_i \ = < v_i, \ :=, < \text{CREATE-}id_{C_i}, Q_{act_i} > > \ and$

$\quad as_i \in AS_2 \ and$

$AS \ = \ AS_1 \ \oplus \ AS_2$

Figure 184 The T^{as}_{m} transformation

$$
\begin{aligned}
v_e^{as}(OOD,S,e) \;=\; & AS \text{ where} \\
& e = < id_{S_n}, Q_{act} > \Rightarrow \\
& \quad AS = T_m^{as}(OOD, e, v_m(OOD,S,e)) \; and \\
& e = < e_1,\ +, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1,\ \text{and}, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1,\ \&, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1,\ /, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1,\ =, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1,\ **, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1, >=, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1, >, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1, <=, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1, <, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1,\ *, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1, <>, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1,\ \text{or}, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < e_1, -, e_2 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \oplus v_e^{as}(OOD,S,e_2) \; and \\
& e = < -, e_1 > \Rightarrow AS = v_e^{as}(OOD,S,e_1) \; and \\
& e = < \ \text{not}, e_1 > \Rightarrow AS = v_e^{as}(OOD,S,e_1)
\end{aligned}
$$

Figure 185 The v_e^{as} transformation

m. The v_e^{as} transformation, defined in Figure 185, is used to build the sequence of assignment statements for functional messages that appear in expressions. Let OOD represent the object-oriented design, and let S represent the original subprogram. The v_M transformation is defined to update the statements from the main program, Ψ, by replacing all subprogram calls with messages. For procedural messages, the sequence of assignment statements generated by the T_m^{as} transformation is spliced into Ψ before the message. For functional messages, the sequence of assignment statements is spliced into Ψ right before the statement that includes the message. The v_M transformation is defined in Figure 186. This transformation uses the v_m and v_e transformations to transform subprogram calls into messages. The *reduce* operation is used for output statements in order to combine the results of the v_e^{as} transformation of the expressions using the $\oplus$ operator.

```
$v_M(S,\ \Psi)\ =\ \Psi'\ where$
    $\forall\ s \in \Psi$
        $\Psi\ =\ \Psi_1\ \oplus\ [s]\ \oplus\ \Psi_2\ and$
        $s\ = <id_{S_n}, P_{act} > \Rightarrow$
            $s'\ =\ v_m(S,\ s)\ and$
            $AS\ =\ T_m^{as}(OOD, s, s')\ and$
        $s\ = < x,\ :=, e > \Rightarrow$
            $s'\ = < x,\ :=, v_e(S,e) >\ and$
            $AS\ =\ v_e^{as}(OOD, S, e)\ and$
        $s\ = < \text{if}, e,\ \text{then}, S_1,\ \text{else}, S_2 > \Rightarrow$
            $s'\ = < \text{if}, v_e(S,e),\ \text{then}, v_M(S,S_1),\ \text{else}, v_M(S,S_2) >\ and$
            $AS\ =\ v_e^{as}(OOD, S, e)\ and$
        $s\ = < \text{while}, e, S_3 > \Rightarrow$
            $s'\ = < \text{while}, v_e(S,e), v_M(S,S_3) >\ and$
            $AS\ =\ v_e^{as}(OOD, S, e)\ and$
        $s\ = < \text{input},\ \text{iport}, E > \Rightarrow$
            $s'\ =\ s\ and$
        $s\ = < \text{output},\ \text{oport}, E > \Rightarrow$
            $s'\ = < \text{output},\ \text{oport}, E' >\ and$
                $E'\ =\ [v_e(S,e)\ |\ e \in E]\ and$
            $AS\ =\ reduce(\oplus, [v_e^{as}(OOD, S, e)\ \ |\ e \in E])\ and$
        $\Psi'\ =\ \Psi_1\ \oplus\ AS\ \oplus\ [s']\ \oplus\ \Psi_2\ and$
```

Figure 186 The v_M transformation

```
class CLASS-SYSTEM attributes
   method BMDSIM1 ( C-9 )
     begin
     ...
     BOUNCE ( XLAMDA, ANGFAC, BEAMRA, RHOSTD );
     ...
     end
```

Figure 187   BMDSIM1 before BOUNCE converted

The formal transformation T^{v}_{M} applies the v_M transformation to the main method of the overall system class and is defined as follows. Let OOD represent an object-oriented design, let M represent the main program from the imperative system, let C represent the `CLASS-SYSTEM` built for the main program.

$$\begin{aligned}
T^{v}_{M}(OOD, M, C) &= OOD' \text{ where} \\
C &= <\text{CLASS-SYSTEM}, \emptyset, \Omega_C, \lambda_0> \textit{ and} \\
\Omega_C &= \{< id_{main}, \tau, \emptyset, \emptyset, \emptyset, Q_{loc}, \Psi_{main} >\} \textit{ and} \\
\Psi'_{main} &= v_M(M, \Psi_{main}) \textit{ and} \\
\Omega'_C &= \{< id_{main}, \tau, \emptyset, \emptyset, \emptyset, Q_{loc}, \Psi'_{main} >\} \textit{ and} \\
C' &= <\text{CLASS-SYSTEM}, \emptyset, \Omega'_C, \lambda_0> \textit{ and} \\
OOD' &= T^{+}_{c}(T^{-}_{c}(OOD, C), C')
\end{aligned}$$

For example, Figure 187 shows the `CLASS-SYSTEM` main program method `BMDSIM1` before the T^{v}_{main} transformation. The `BOUNCE` subprogram call has not yet been converted to a message. Figure 188 shows the `BMDSIM1` method after the `BOUNCE` subprogram call has been converted to a message and the object instances required for the message have been inserted.

In order to convert the main program, the σ_M transformation is defined in Figure 189. Let OOD represent an object-oriented design and let M represent the main program. This transformation takes the main program, M, as input and unions the designs built from applying σ to each of the subprograms called by M. Each subprogram called by the main program will also apply σ to any subprograms it calls. This formalizes the depth-first approach used in the PBOI methodology. The resulting designs are unioned

```
class CLASS-SYSTEM attributes
  method BMDSIM1 ( C-9 )
    begin
    ...
    C-4 := CREATE-CLASS-4 ( XLAMDA );
    C-5 := CREATE-CLASS-2 ( ANGFAC, BEAMRA );
    C-8 := CREATE-CLASS-3 ( ANGFAC, RHOSTD );
    BOUNCE ( C-4, C-5, C-8 );
    ...
    end
```

Figure 188 BMDSIM1 after BOUNCE converted

together to build OOD'. The CLASS-SYSTEM class is built with no attributes and the main program statements are converted using the θ transformation. There is no need to use the δ transformation because CLASS-SYSTEM has no attributes. Each of the subprogram calls is converted to a message using the T_M^v transformation. Duplicate classes are removed by the T_{class}^{dup} transformation, duplicate objects are removed by the T_{obj}^{dup} transformation, and overlapping classes are merged using the T_{class}^{merge} transformation.

6.12 Eliminating Category 4 and Category 5 Subprograms

This section defines the formal transformations used to convert Category 4 and Category 5 subprograms (in the GIM) into collections of Category 2 or Category 3 subprograms (in the GIM). These transformations are built using the formal definitions of a subprogram and subprogram call provided in Chapter III. Since only *procedures* are allowed to have multiple outputs, the transformations are restricted to this form of subprogram.

As presented in Chapter III, a procedure, S, is represented by the following tuple.

$$< id_S, P_{in}, P_{out}, [\,], P_{loc}, \Sigma >$$

Let Σ_p represent the *program slice* on the statements in Σ required to produce p. Weiser [76] defines a program slice as a *projection* of the behavior of the original procedure. Let π represent a function that indicates a sequence of statements, Σ_p, projects the behavior

$$
\begin{aligned}
&\sigma_M(OOD, M) \;=\; OOD^{(5)} \text{ where} \\
&\quad M \;= < id_{main}, \emptyset, \emptyset, \emptyset, P_{loc}, \Sigma_M > \; and \\
&\quad \forall\, l \in C_{sub_M} \\
&\qquad l \;= < id_{S_n}, P_{act_n} > \; and \\
&\qquad S_n \in C^*_{sub_M} \\
&\quad OOD' \;=\; \bigcup_{i=1}^{n} (\sigma(OOD, S_i)) \;\text{ where } S_i \in C^*_{sub_M} \; and \\
&\quad OOD'' \;=\; T^{dup}_{class}(OOD') \; and \\
&\quad \forall\, C_i \in OOD'' \\
&\qquad \kappa(C_i) \;\in OOD''' \; and \\
&\quad C \;= < \text{CLASS-SYSTEM}, \emptyset, \Omega_C, \lambda_0 > \; and \\
&\quad instance(\tau, C) \; and \\
&\quad \Psi \;=\; \theta(\Sigma_M) \; and \\
&\quad Q_{loc} \;=\; \theta_v(P_{loc}) \; and \\
&\quad \Omega_C \;=\; \{< id_{main}, \tau, \emptyset, \emptyset, \emptyset, Q_{loc}, \Psi >\} \; and \\
&\quad OOD^{(4)} \;=\; T^v_M(OOD''', M, C) \; and \\
&\quad C' \;=\; \text{ updated } C \in OOD^{(4)} \; and \\
&\quad OOD^{(5)} \;=\; T^{dup}_{obj}(OOD^{(4)}, C') \; and \\
&\quad C'' \;=\; \text{ updated } C' \in OOD^{(5)} \; and \\
&\quad OOD^{(6)} \;=\; T^{merge}_{class}(OOD^{(5)}, C'')
\end{aligned}
$$

Figure 189 The σ_M transformation

of another sequence of statements, Σ, in order to produce a variable p. Formally,

$$\Sigma_p \;=\; \pi(\Sigma, p)$$

Any sequence of statements that meets the condition, π, is a program slice of the original procedure and produces the single data item p. A new subprogram built from this program slice is represented as a tuple that is defined as follows. Let id_{S_p} represent the name of the new subprogram. Let P_{in_p} represent the input parameters that are referenced in the program slice, such that $P_{in_p} \subseteq P_{in}$. Let P_{loc_p} represent any local variables referenced in the program slice, such that $P_{loc_p} \subseteq P_{loc_p}$. Given a data item, $p \in P_{out}$ that is produced by Σ, the program slice, Σ_p, is defined as shown below.

$$< id_{S_p}, P_{in_p}, [p], [\ ], P_{loc_p}, \pi(\Sigma_p) >$$

As presented in Chapter III, a call to procedure S is defined by the following tuple.

$$< id_S, P_{act} >$$

In order to differentiate between the actual parameters that are input parameters and the actual parameters that are output parameters, the representation of a procedure call is extended as follows. Let $P_{act_{in}}$ represent the input actual parameters and let $P_{act_{out}}$ represent the output actual parameters, such that $P_{act_{in}} \oplus P_{act_{out}} = P_{act}$. The procedure call is represented by the following tuple.

$$< id_S, P_{act_{in}}, P_{act_{out}} >$$

A call to the new subprogram S_p is represented as follows. Let $P_{act_{in_p}}$ represent the input actual parameters that match formal parameters from the new subprogram. Let a represent the output actual parameter that corresponds to the output formal parameter p from the new subprogram, i.e. $\mu(a) = p$. The procedure call to S_p is represented by the following tuple.

$$< id_{S_p}, P_{act_{in_p}}, [a] >$$

6.12.1 Build Slices Transformation. As defined in Section 5.15, the first step in converting Category 4 and Category 5 subprograms is to build one new subprogram for each of the data items produced by the Category 4 or Category 5 subprograms. This section defines the formal transformation used to generate the program slices for all Category 4 and Category 5 subprograms. Once the slices are built, the transformation for inter-procedural slicing is used to call each program slice as needed. That transformation is also defined in this section.

```
$T_{ps}^{build}(SD) \; = \; SD'$ where
    For each $S \in SD$
        $S \; = < id_S, P_{in}, P_{out}, [\,], P_{loc}, \Sigma_S > \; and$
        $S \in SD' \; and$
        $Cat_4(S) \; or \; Cat_5(S) \Rightarrow$
            $[\forall \; p \in P_{out}$
                $S_p \; = < id_{S_p}, P_{in_p}, [p], [\,], P_{loc_p}, \pi(\Sigma_S) > \; and$
                $S_p \in SD' \,]$
```

Figure 190 The T_{ps}^{build} transformation

Let T_{ps}^{build} represent the transformation that builds each of the program slices for Category 4 and Category 5 subprograms. Let SD represent a collection of imperative subprograms. Let Σ_{S_p} represent the program slice generated for the data item p from the subprogram S. The T_{ps}^{build} transformation is defined as shown in Figure 190.

This transformation generates a program slice for each output parameter, p, of Category 4 and Category 5 subprograms. Each of the original subprograms is included in SD', and the subprograms built from the program slices are also included. The next step is to change each subprogram call to a Category 4 or Category 5 subprogram into a sequence of calls to the new subprograms built from its program slices.

Let T_{ps}^{calls} represent the transformation that builds the sequence of calls to the new subprograms. Let SD represent a collection of imperative subprograms, and let S represent one of these subprograms. The T_{ps}^{calls} transformation is defined in Figure 191. This transformation creates a sequence of subprogram calls, L_{S_n} for each call in S that is invoking a Category 4 or Category 5 subprogram. The L_{S_n} sequence collects the calls to the new subprograms that are built from the program slices. Each data item in S that is produced from the Category 4 or Category 5 subprogram is now produced in S' by calling the new subprogram. The updated subprogram, S', is added to the updated design, SD', which is returned as the result of the transformation.

```
$T_{ps}^{calls}(SD, S) \;=\; SD'$ where
  $S \;= < id_S, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma_S > \;\; and$
  $\forall\; s \in \Sigma_S$
    $\Sigma_S \;=\; \Sigma_{S_1} \;\oplus\; [s] \;\oplus\; \Sigma_{S_2} \; and$
    $s \;= < id_{S_n}, P_{act_{in}}, P_{act_{out}} > \Rightarrow$
      $S_n \in SD \; and$
      $S_n \;= < id_{S_n}, P_{in}, P_{out}, [\;], P_{loc}, \Sigma_{S_n} > \;\; and$
      $\mid P_{out} \mid \;>\; 1 \Rightarrow$
        $\forall\; p \in P_{out}$
          $S_p \;= < id_{S_p}, P_{in_p}, [p], [\;], P_{loc_p}, \Sigma_{S_p} > \;\; and$
          $S_p \in SD' \; and$
          $L_{S_n} \;=\; [l_p \mid l_p \;= < id_{S_p}, P_{act_{in_p}}, [p] > \;\; and$
            $P_{act_{in_p}} \;=\; [a \mid a \in P_{act_{in}} \; and$
              $b \in P_{in_p} \; and \; \mu(a) \;=\; b] \; and$
      $\mid P_{out} \mid \; <= \; 1 \Rightarrow$
        $L_{S_n} \;=\; [s] \; and$
      $\Sigma'_S \;=\; \Sigma_{S_1} \;\oplus\; L_{S_n} \;\oplus\; \Sigma_{S_2} \; and$
    $s \;\neq < id_{S_n}, P_{act_{in}}, P_{act_{out}} > \Rightarrow$
      $\Sigma'_S \;=\; \Sigma_{S_1} \;\oplus\; [s] \;\oplus\; \Sigma_{S_2} \; and$
  $S' \;= < id_S, P_{in}, P_{out}, P_{ret}, P_{loc}, \Sigma'_S > \;\; and$
  $SD' \;=\; T_s^{+}(T_s^{-}(SD, S), S')$
```

Figure 191 The T_{ps}^{calls} transformation

Let T_{ps}^{main} represent the transformation that builds all the program slices and updates the imperative subprograms to call the program slices instead of the Category 4 and Category 5 subprograms. Let SD represent a collection of imperative subprograms.

$$\begin{aligned} T_{ps}^{main}(SD) &= SD'' \text{ where} \\ SD' &= T_{ps}^{build}(SD) \textit{ and} \\ SD'' &= T_{ps}^{calls}(SD', S) \end{aligned}$$

The result of this transformation is an imperative design where each Category 4 and Category 5 subprogram has been replaced by the procedures built from program slicing the subprogram. Each subprogram in this design is updated to call the program slices instead of the Category 4 or Category 5 subprogram.

6.13 Converting an Imperative Design

The PBOI method, as a whole, converts a collection of imperative subprograms into a collection of classes and methods. The collection of imperative subprograms is represented as an imperative design, and the collection of classes and methods is represented as an object-oriented design. The conversion of the imperative design begins with the main program, which is identified by the user (see Assumption V.1 in Chapter V). The transformation that formalizes the conversion of the imperative design to an object-oriented design is the σ_F transformation. This transformation is defined below.

Let SD represent an imperative design, and let OOD represent an object-oriented design. The σ_F transformation is defined as follows.

$$\begin{aligned} &\sigma_F(SD) = OOD \text{ where} \\ &\quad SD' = T_{ps}^{main}(SD) \textit{ and} \\ &\quad M' \in SD' \textit{ and } Cat_M(M') \textit{ and} \\ &\quad OOD = \sigma(\emptyset, M') \end{aligned}$$

This transformation uses the T_{ps}^{main} transformation to build a new imperative design without Category 4 or Category 5 subprograms. The σ_M transformation is used to convert the main program of this imperative design. The result of this transformation is the object-oriented design OOD.

6.14 Summary

This chapter has presented transformations that formalize the conversion of imperative statements, expressions, parameters, attributes, and subprograms using the PBOI method. Section 6.1 provides a table listing the transformations that are presented in this chapter. The transformations that formalize the program slicing process have also been presented in this chapter. Using these transformations, a collection of imperative subprograms can be converted to a functionally equivalent collection of classes and methods that implement the subprograms. This claim is discussed and proven in the following chapter.

VII. Proving Functional Equivalence

7.1 Introduction

In order to evaluate the Object-Oriented (OO) design that is extracted using the PBOI methodology, a proof is developed in this chapter that shows the extracted OO code (represented in the GOM) is a consistent implementation of the original imperative code (represented in the GIM). Given the same input, a ***consistent*** implementation produces the same output as the original code. To build this proof, the behavior of both the original imperative code and the extracted OO code are defined.

Let F be the original imperative code as represented in the GIM. Let F' be the extracted OO code as represented in the GOM. One undesirable approach for showing F' is a consistent implementation of F is to delineate all the inputs and outputs for F and show that an input given to F' produces the same output as F. This is an impractical approach and is not discussed further. At the highest level, F represents the main program from the imperative design and F' represents the main method from the object-oriented design. The transformation that converts F to F' is the σ_F transformation, i.e. $F' = \sigma_F(F)$. Since the semantics of both F and F' are expressed using the weakest precondition notation, the proof of functional equivalence is based on their weakest preconditions. Given a postcondition R, the semantics of F are defined as follows.

$$\{\ wp(F,\ R)\ \}\ F\ \{\ R\ \}$$

Since the postcondition and weakest precondition are predicates (expressions) defined in terms of imperative variables, they must be transformed as well. The θ_e transformation built to convert expressions is used to convert these predicates. Hence, the semantics of F' are defined as follows.

$$\{\ wp(F',\ \theta_e(R)\ \}\ F'\ \{\ \theta_e(R)\ \}$$

If it can be proven that the transformation of imperative subprograms is a *morphism* [46], it can be proven that the *weakest precondition* predicate is "preserved" by the transformation. If this can be proven, then the semantics for the imperative subprograms are equivalent to the semantics of the object-oriented methods, i.e. they are functionally equivalent. The σ_F transformation is a morphism if the following equality can be proven correct.

$$\theta_e(wp(F,\ R)) \Leftrightarrow\ wp(\sigma_F(F),\ \theta_e(R))$$

The rest of this chapter provides lemmas, theorems, and proofs which are used to prove that this equality holds. Since the σ_F transformation uses several other transformations, proofs are presented that prove these transformations maintain functional equivalence. The proofs for the θ and δ transformations are presented first, followed by proofs for the v and σ transformations. The proof for the σ_F transformation is presented in Section 7.6, at the end of the chapter.

7.2 Proof for Statement Transformations (θ)

The θ transformation is used to convert a sequence of GIM statements to a sequence of GOM statements. Theorems are presented and proven below showing that the θ transformation of assignment statements, input statements, output statements, sequential control flow, selective control flow, and iterative control flow maintains functional equivalence. Imperative subprograms and subprogram calls are not transformed by θ, but by σ and v, respectively. The proofs for σ and v appear in Section 7.4.

Since the ***textual substitution*** [19] notation, R^x_e, is used for defining the semantics of assignment statements, the following lemma is provided for $\theta_e(R^x_e)$.

Lemma VII.1. $\theta_e(R^x_e)\ \Leftrightarrow\ \theta_e(R)^{\theta_v(x)}_{\theta_e(e)}$

Proof. 1. The notation R^x_e symbolizes the simultaneous replacement of all free occurrences of x with e in the predicate R.

2. When θ_e is applied to R^x_e on the left-hand-side of the equality, the textual substitution is done first and the θ_e transformation is applied to R as well as the newly substituted occurrences of e, i.e. $\theta_e(e)$.

3. When the θ_e transformation is applied to R on the right-hand-side of the equality, the occurrences of x are converted using $\theta_v(x)$ so the substitution in $\theta_e(R)$ works properly.
4. The θ_e transformation is applied to e before it replaces $\theta_v(x)$ in $\theta_e(R)$ so that the θ_e transformation of R is complete after the substitution.
5. Thus, on both sides of the equality, θ_e is applied to all newly substituted occurrences of e in R.

□

7.2.1 Proof for Assignment Statements. The following theorem is proven in order to shown that the transformation of assignment statements preserves functional equivalence.

Theorem VII.1. *The θ transformation of* assignment *statements in the GIM preserves functional equivalence.*

Proof.

$$\begin{aligned}
&wp(\theta(<\ \mathrm{x}, :=, \mathrm{e}\ >),\ \theta_e(R)) \Leftrightarrow \\
&\quad wp(<\theta_v(x), :=, \theta_e(e)>,\ \theta_e(R)) \Leftrightarrow \\
&\quad \theta_e(R)^{\theta_v(x)}_{\theta_e(e)} \Leftrightarrow \\
&\quad \theta_e(R^x_e) \Leftrightarrow \text{ by Lemma VII.1} \\
&\quad \theta_e(wp(<\ \mathrm{x}, :=, \mathrm{e}\ >,\ R))
\end{aligned}$$

□

7.2.2 Proof for Input and Output Statements. The following theorem is proven in order to shown that the transformation of input statements preserves functional equivalence. The proof is given for a sequence of 2 inputs. The extension to arbitrary length sequences is obvious.

Theorem VII.2. *The θ transformation of* input *statements in the GIM preserves functional equivalence.*

Proof.

$$
\begin{aligned}
&wp(\theta(<\text{ input, iport, } [v_1,\ v_2] >),\ \theta_e(R)) \Leftrightarrow \\
&\quad wp(<\text{ input, iport, } \theta_V([v_1,\ v_2]) >,\ \theta_e(R)) \Leftrightarrow \\
&\quad wp(<\text{ input, iport, } [\theta_v(v_1),\ \theta_v(v_2)] >,\ \theta_e(R)) \Leftrightarrow \\
&\quad wp([< \theta_v(v_1), :=, \theta_e(x_1) >,\ < \theta_v(v_2), :=, \theta_e(x_2) >], \theta_e(R)) \Leftrightarrow \\
&\quad wp([< \theta_v(v_1), :=, \theta_e(x_1) >,\ \theta_e(R)^{\theta_v(v_2)}_{\theta_e(x_2)} \Leftrightarrow \\
&\quad (\theta_e(R)^{\theta_v(v_1)}_{\theta_e(x_1)})^{\theta_v(v_2)}_{\theta_e(x_2)} \Leftrightarrow \\
&\quad \theta_e((R^{v_1}_{x_1})^{v_2}_{x_2}) \Leftrightarrow \text{ by Lemma VII.1} \\
&\quad \theta_e(wp(< v_1, :=, x_1 >,\ R^{v_2}_{x_2})) \Leftrightarrow \\
&\quad \theta_e(wp(< v_1, :=, x_1 >,\ wp(< v_2, :=, x_2 >,\ R))) \Leftrightarrow \\
&\quad \theta_e(wp([< v_1, :=, x_1 >,\ < v_2, :=, x_2 >], R)) \Leftrightarrow \\
&\quad \theta_e(wp(<\text{ input, iport, } [v_1,\ v_2] >,\ R))
\end{aligned}
$$

$\square$

The following theorem is proven in order to shown that the transformation of output statements preserves functional equivalence.

Theorem VII.3. *The θ transformation of* output *statements in the GIM preserves functional equivalence.*

Proof.

$$
\begin{aligned}
&wp(\theta(<\text{ output, oport, } E >),\ \theta_e(R)) \Leftrightarrow \\
&\quad wp(<\text{ output, oport, } \theta_E(E) >,\ \theta_e(R)) \Leftrightarrow \\
&\quad \theta_e(R) \Leftrightarrow \\
&\quad \theta_e(wp(<\text{ output, oport, } E >,\ R))
\end{aligned}
$$

$\square$

7.2.3 Proof for Skip Statement. The following theorem is proven in order to shown that the transformation of skip statements preserves functional equivalence.

Theorem VII.4. *The θ transformation of* skip *statements in the GIM preserves functional equivalence.*[1]

[1] *There is no surface syntax defined for the skip statement.*

Proof.

$$\begin{aligned}
&wp(\theta(skip),\ \theta_e(R)) \Leftrightarrow\\
&\quad wp(skip,\ \theta_e(R)) \ \Leftrightarrow\\
&\quad \theta_e(R) \ \Leftrightarrow\\
&\quad \theta_e(wp(skip,\ R))
\end{aligned}$$

□

7.2.4 Proof for Sequential Control Flow. The following theorem is proven in order to shown that the transformation of sequences of statements preserves functional equivalence.

Theorem VII.5. *The θ transformation of* sequential *control flow for sequences of assignment, input, output, and skip statements preserves functional equivalence.*

Proof. Since it has already been proven that the θ transformation maintains functional equivalence for the assignment, input, output, and skip statements, the following equality holds.

$$\begin{aligned}
&wp(\theta([s_1, s_2]),\ \theta_e(R)) \Leftrightarrow\\
&\quad wp([\theta(s_1), \theta(s_2)],\ \theta_e(R)) \ \Leftrightarrow\\
&\quad wp(\theta(s_1), wp(\theta(s_2),\ \theta_e(R))) \ \Leftrightarrow\\
&\quad wp(\theta(s_1), \theta_e(wp(s_2,\ R))) \ \Leftrightarrow\\
&\quad \theta_e(wp(s_1, wp(s_2,\ R))) \ \Leftrightarrow\\
&\quad \theta_e(wp([s_1, s_2], R))
\end{aligned}$$

□

7.2.5 Proof for Selective Control Flow. This section provides a proof showing that the θ transformation of selective control flow statements maintains functional equivalence. The following theorem is proven below.

Theorem VII.6. *The θ transformation of* selective *control flow in the GIM preserves functional equivalence.*

Proof by Induction. 1. *Base Case:* The proof of Theorem VII.5 shows that θ maintains functional equivalence for sequences of assignment, input, output, or skip statements. Thus, the following equality holds.

$$\begin{aligned}
&wp(\theta(<\text{ if}, e, \text{ then}, S_1, \text{ else}, S_2 >), \theta_e(R)) \Leftrightarrow \\
&\quad wp(<\text{ if}, \theta_e(e), \text{ then}, \theta(S_1), \text{ else}, \theta(S_2) >), \theta_e(R)) \Leftrightarrow \\
&\quad ((\theta_e(e) \Rightarrow wp(\theta(S_1), \theta_e(R))) \wedge (\neg\theta_e(e) \Rightarrow wp(\theta(S_2), \theta_e(R)))) \Leftrightarrow \\
&\quad ((\theta_e(e) \Rightarrow \theta_e(wp(S_1, R))) \wedge (\neg\theta_e(e) \Rightarrow \theta_e(wp(S_2, R)))) \Leftrightarrow \\
&\quad (\theta_e((e \Rightarrow wp(S_1, R))) \wedge \theta_e((\neg e \Rightarrow wp(S_2, R)))) \Leftrightarrow \\
&\quad \theta_e(((e \Rightarrow wp(S_1, R)) \wedge (\neg e \Rightarrow wp(S_2, R)))) \Leftrightarrow \\
&\quad \theta_e(wp(<\text{ if}, e, \text{ then}, S_1, \text{ else}, S_2 >, R))
\end{aligned}$$

2. *Inductive Hypothesis:* Assume the sequences of statements S_1 and S_2 also include if-then-else statements, and the θ transformation correctly transforms an if-then-else statement with an arbitrary number, n, of nested if-then-else statements.

3. *Inductive Step:* Prove that adding the $n+1$ nested if-then-else statement maintains the functional equivalence. Since this new nested if-then-else is the final nested statement, it includes only assignment, input, output, or skip statements in the sequences S_1 and S_2. The functional equivalence is maintained for this statement as shown for the base case.

□

The corollary shown below follows from the proof that the θ transformation of the selective statement maintains functional equivalence.

Corollary VII.1. *The θ transformation of sequential control flow for sequences of assignment, input, output, skip, and* selective *statements preserves functional equivalence.*

7.2.6 Proof for Iterative Control Flow. This section provides a proof showing that the θ transformation of iterative control flow statements maintains functional equivalence.

Theorem VII.7. *The θ transformation for* iterative *control flow in the GIM preserves functional equivalence.*

Proof by Induction. 1. *Base Case:* The proof of Theorem VII.5 shows that θ maintains functional equivalence for sequences of assignment, input, output, or skip statements. Thus, the equality shown in Figure 192 holds.

2. *Inductive Hypothesis:* Assume the sequence of statements S_3 also includes while statements, and the θ transformation correctly transforms a while statement with an arbitrary number, n, of nested while statements.

3. *Inductive Step:* Prove that adding the $n+1$ nested while statement maintains functional equivalence. Since this new nested while is the final nested statement, it includes only assignment, input, output, or skip statements in the sequence S_3. The functional equivalence is maintained for this statement as shown for the base case.

□

The corollary shown below follows from the proof that the transformation of iterative statements maintains functional equivalence.

Corollary VII.2. *The θ transformation of sequential control flow for sequences of assignment, input, output, skip, selective, and* iterative *statements preserves functional equivalence.*

Given that proofs have been provided for selective and iterative control flow, the following two corollaries follow from these proofs.

Corollary VII.3. *The θ transformation of selective control flow for sequences of assignment, input, output, skip,* selective, *and* iterative *statements preserves functional equivalence.*

Corollary VII.4. *The θ transformation of iterative control flow for sequences of assignment, input, output, skip,* selective, *and* iterative *statements preserves functional equivalence.*

Finally, the corollary shown below follows from all of the proofs and corollaries provided for the θ transformation to this point. This corollary provides an assertion that the θ transformation preserves functional equivalence for each of the statements defined in the GIM, except subprogram definitions and calls.

$$\begin{array}{ll}
wp(\theta(<\ \text{while},\ e,\ S_3\ >),\ \theta_e(R)) \Leftrightarrow & \\
\quad wp(<\ \text{while},\ \theta_e(e),\ \theta(S_3)\ >),\ \theta_e(R)) \Leftrightarrow & \\
\quad wp([<\ \text{if},\ \theta_e(e),\ \text{then},\ \theta(S_3),\ \text{else},\ \theta([skip])>, & \text{iteration 1} \\
\qquad <\ \text{if},\ \theta_e(e),\ \text{then},\ \theta(S_3),\ \text{else},\ \theta([skip])>, & \text{iteration 2} \\
\qquad\qquad\qquad \ldots & \\
\qquad <\ \text{if},\ \theta_e(e),\ \text{then},\ \theta(S_3),\ \text{else},\ \theta([skip])>], \theta_e(R)) \Leftrightarrow & \text{iteration k} \\
\quad wp(<\ \text{if},\ \theta_e(e),\ \text{then},\ \theta(S_3),\ \text{else},\ \theta([skip])>, & \text{iteration 1} \\
\qquad wp(<\ \text{if},\ \theta_e(e),\ \text{then},\ \theta(S_3),\ \text{else},\ \theta([skip])>, & \text{iteration 2} \\
\qquad\qquad\qquad \ldots & \\
\qquad\quad wp(<\ \text{if},\ \theta_e(e),\ \text{then},\ \theta(S_3),\ \text{else},\ \theta([skip])>, \theta_e(R)))) \Leftrightarrow & \text{iteration k} \\
\quad wp(<\ \text{if},\ \theta_e(e),\ \text{then},\ \theta(S_3),\ \text{else},\ \theta([skip])>, & \text{iteration 1} \\
\qquad wp(<\ \text{if},\ \theta_e(e),\ \text{then},\ \theta(S_3),\ \text{else},\ \theta([skip])>, & \text{iteration 2} \\
\qquad\qquad\qquad \ldots & \\
\qquad\quad \theta_e(wp(<\ \text{if},\ e,\ \text{then},\ S_3,\ \text{else},\ [skip]>,\ R)))) \Leftrightarrow & \text{iteration k} \\
\quad wp(<\ \text{if},\ \theta_e(e),\ \text{then},\ \theta(S_3),\ \text{else},\ \theta([skip])>, & \text{iteration 1} \\
\qquad \theta_e(wp(<\ \text{if},\ e,\ \text{then},\ S_3,\ \text{else},\ [skip]>, & \text{iteration 2} \\
\qquad\qquad\qquad \ldots & \\
\qquad\quad wp(<\ \text{if},\ e,\ \text{then},\ S_3,\ \text{else},\ [skip]>,\ R)))) \Leftrightarrow & \text{iteration k} \\
\quad \theta_e(wp(<\ \text{if},\ e,\ \text{then},\ S_3,\ \text{else},\ [skip]>, & \text{iteration 1} \\
\qquad wp(<\ \text{if},\ e,\ \text{then},\ S_3,\ \text{else},\ [skip]>, & \text{iteration 2} \\
\qquad\qquad\qquad \ldots & \\
\qquad\quad wp(<\ \text{if},\ e,\ \text{then},\ S_3,\ \text{else},\ [skip]>,\ R)))) \Leftrightarrow & \text{iteration k} \\
\quad \theta_e(wp([<\ \text{if},\ e,\ \text{then},\ S_3,\ \text{else},\ [skip]>, & \text{iteration 1} \\
\qquad\quad <\ \text{if},\ e,\ \text{then},\ S_3,\ \text{else},\ [skip]>, & \text{iteration 2} \\
\qquad\qquad\qquad \ldots & \\
\qquad\quad <\ \text{if},\ e,\ \text{then},\ S_3,\ \text{else},\ [skip]>],\ R)) \Leftrightarrow & \text{iteration k} \\
\quad \theta_e(wp(<\ \text{while},\ e,\ S_3\ >,\ R)) &
\end{array}$$

Figure 192 Proof for iterative base case.

Corollary VII.5. *The θ transformation of sequences of assignment, input, output, skip, selective, and iterative statements preserves functional equivalence.*

7.3 Proof for Parameters and Attributes (δ)

This section presents theorems and proofs that the δ transformation preserves functional equivalence. This transformation is used by the σ transformation when GOM parameters are moved to attributes and vice versa. Since the δ transformation converts accesses of GOM variables into "GET-" and "SET-" messages, it must proven that an access to variable is functionally equivalent to a "GET-" message. It must also be proven that the assignment of a GOM variable is functionally equivalent to a "SET-" message. These transformations involve only GOM entities, so there is no need to convert the postcondition using θ_e. Let R' represent a postcondition represented using GOM variables.

The following theorem is proven below in order to show that the δ transformation of an access of a variable into a "GET-" message preserves functional equivalence.

Theorem VII.8. *The access of a variable, a, in a statement is functionally equivalent to a call to the "GET-a" message.*

As an example of this theorem, consider the expression that appears on the right-hand-side of an assignment statement. The following proof shows that if this expression is the variable a, then the semantics of the assignment statement are preserved.

Proof.

$$
\begin{aligned}
& wp(\delta([< x',\ :=, a >], c, \{a\}), R') \Leftrightarrow \\
& \quad wp(< x',\ :=, \delta_e(a) >,\ R') \Leftrightarrow \\
& \quad wp(< x',\ :=, <\ \text{GET-}a, [c] >>,\ R') \Leftrightarrow \\
& \quad wp([y := c.a,\ b := y, < x',\ :=, b >],\ R') \Leftrightarrow \\
& \quad wp(y := c.a, wp(b := y, wp(< x',\ :=, b >,\ R'))) \Leftrightarrow \\
& \quad wp(y := c.a, wp(b := y,\ R'^{x'}_{b})) \Leftrightarrow \\
& \quad wp(y := c.a,\ (R'^{x'}_{b})^{b}_{y}) \Leftrightarrow \\
& \quad ((R'^{x'}_{b})^{b}_{y})^{y}_{c.a} \Leftrightarrow \\
& \quad R'^{x'}_{c.a} \Leftrightarrow \\
& \quad R'^{x'}_{a} \Leftrightarrow \\
& \quad wp(< x',\ :=, a >, R')
\end{aligned}
$$

□

Other examples of these expressions are limited to the expressions that appear in the boolean tests of selective and iterative statements. Similar arguments are made for these expressions, so in general, Theorem VII.8 holds for any statement in the GOM.

The following theorem is proven below in order to show that the δ transformation of an assignment of a variable into a "SET-" message preserves functional equivalence. Let R' represent a postcondition represented using GOM variables.

Theorem VII.9. *The assignment of a variable, a, is functionally equivalent to a call to the "SET-a" message.*

Proof.

$$
\begin{aligned}
& wp(\delta([< a,\ :=, e >], c, \{a\}),\ R') \Leftrightarrow \\
& \quad wp(<\ \text{SET-a}, [c, \delta_e(e)] >,\ R') \Leftrightarrow \\
& \quad R'^{c.a}_{\delta_e(e)} \Leftrightarrow \\
& \quad R'^{a}_{\delta_e(e)}) \Leftrightarrow \\
& \quad wp(< a,\ :=, \delta_e(e) >, R') \Leftrightarrow \\
& \quad wp(< a,\ :=, e >, R')\ \text{ from Theorem VII.8.}
\end{aligned}
$$

□

Similarly, the δ^{-1} transformation converts an access of a specific object attribute into an access of a GOM variable. Specifically, each "GET-a" message is converted into a reference to the variable a, and each "SET-a" message is converted into an assignment to a. The following theorem is proven in order to show that replacing the "GET-a" message with an access of a variable, a, in an expression maintains functional equivalence. Let R' represent a postcondition represented using GOM variables.

Theorem VII.10. *The call to a "GET-a" message in a statement is functionally equivalent to the access of the variable, a.*

As an example of this theorem, consider the expression that appears on the right-hand-side of an assignment statement. The following proof shows that replacing the "GET-a" message with an access to the variable a maintains the functionality of the assignment statement.

Proof.

$$\begin{aligned}
&wp(\delta^{-1}([< x,\ :=, <\ \text{GET-}a, [c] >>], c, \{a\}), R') \Leftrightarrow \\
&\quad wp(< x,\ :=, a >,\ R') \Leftrightarrow \\
&\quad R'^{x}_{a} \Leftrightarrow \\
&\quad R'^{x}_{c.a} \Leftrightarrow \\
&\quad ((R'^{x}_{b})^{b}_{y})^{y}_{c.a} \Leftrightarrow \\
&\quad wp(y := c.a,\ (R'^{x}_{b})^{b}_{y}) \Leftrightarrow \\
&\quad wp(y := c.a, wp(b := y,\ R'^{x}_{b})) \Leftrightarrow \\
&\quad wp(y := c.a, wp(b := y, wp(< x,\ :=, b >,\ R'))) \Leftrightarrow \\
&\quad wp([y := c.a,\ b := y, < x,\ :=, b >],\ R') \Leftrightarrow \\
&\quad wp(< x,\ :=, <\ \text{GET-}a, [c] >>,\ R')
\end{aligned}$$

□

Other examples of these expressions are limited to the expressions that appear in the boolean tests of selective and iterative statements. Similar arguments are made for these expressions, so in general, Theorem VII.10 holds for any statement in the GIM.

The following theorem is proven in order to show that replacing the "SET-a" message with an assignment of a variable, a, maintains functional equivalence. Let R' represent a postcondition represented using GOM variables.

Theorem VII.11. *The "SET-a" message is functionally equivalent to an assignment of the variable a.*

Proof.

$$
\begin{aligned}
& wp(\delta^{-1}(<\text{ SET-a}, [c, e] >),\ R') \Leftrightarrow \\
& \quad wp(< a,\ :=, e >,\ R') \Leftrightarrow \\
& \quad R'^{a}_{e} \Leftrightarrow \\
& \quad R'^{c.a}_{e} \Leftrightarrow \\
& \quad wp(<\text{ SET-a}, [c, e] >,\ R')
\end{aligned}
$$

□

7.4 Proof for Subprogram Conversion (σ)

The following inductive proof shows that the σ transformation of imperative subprograms whose statements also include procedure calls maintains functional equivalence.

Theorem VII.12. *The σ transformation of imperative subprograms where Σ_S is comprised of assignment, input, output, skip, selective, and iterative statements, as well as procedure calls, preserves functional equivalence.*

Proof by Induction. 1. *Base Case:* The σ transformation of a subprogram that calls a procedure whose Σ_S is comprised of assignment, input, output, skip, selective, and iterative statements preserves functional equivalence. Using Corollary VII.5, the base case is proven by showing the v_m transformation of such a call preserves functional equivalence.

$$\begin{aligned}
& wp(v_m(S, < id_S, P_{in}, P_{out} >),\ \theta_e(R)) \Leftrightarrow \\
& \quad wp(< id_S, \theta_V(P_{in}), \theta_V(P_{out}), >, \theta_e(R)) \Leftrightarrow \\
& \quad wp([\bar{x} := \theta_V(P_{in}), \theta(S1), \theta_V(P_{out}) := \bar{z}],\ \theta_e(R)) \Leftrightarrow \\
& \quad wp(\bar{x} := \theta_V(P_{in}), wp(\theta(S1), wp(\theta_V(P_{out}) := \bar{z},\ \theta_e(R)))) \Leftrightarrow \\
& \quad wp(\bar{x} := \theta_V(P_{in}), wp(\theta(S1), \theta_e(R)_{\bar{z}}^{\theta_V(P_{out})})) \Leftrightarrow \\
& \quad wp(\bar{x} := \theta_V(P_{in}), wp(\theta(S1), \theta_e(R_{\bar{z}}^{P_{out}}))) \Leftrightarrow \\
& \quad wp(\bar{x} := \theta_V(P_{in}), \theta_e(wp(S1, R_{\bar{z}}^{P_{out}}))) \Leftrightarrow \\
& \quad \theta_e(wp(S1, R_{\bar{z}}^{P_{out}}))_{\theta_V(P_{in})}^{\bar{x}} \Leftrightarrow \\
& \quad \theta_e(wp(S1, R_{\bar{z}}^{P_{out}})_{P_{in}}^{\bar{x}}) \Leftrightarrow \\
& \quad \theta_e(wp(\bar{x} := P_{in}, wp(S1, R_{\bar{z}}^{P_{out}}))) \Leftrightarrow \\
& \quad \theta_e(wp(\bar{x} := P_{in}, wp(S1, wp(P_{out} := \bar{z},\ R)))) \Leftrightarrow \\
& \quad \theta_e(wp([\bar{x} := P_{in}, S1, P_{out} := \bar{z}],\ R)) \Leftrightarrow \\
& \quad \theta_e(wp(< id_S, P_{in}, P_{out} >,\ R))
\end{aligned}$$

2. *Inductive Hypothesis:* Assume Σ_S of the called procedure also includes calls to other procedures. Since σ is used to transform these called procedures, assume that the σ transformation correctly transforms an arbitrary number, n, of nested procedure calls.

3. *Inductive Step:* Prove that adding the $n+1$ nested procedure call maintains functional equivalence. Since this new nested procedure call is the final nested call, it includes only assignment, input, output, skip, selective, or iterative statements in Σ_S. The functional equivalence is maintained for this call as shown for the base case.

□

Corollary VII.6. *The σ transformation of imperative subprograms where Σ_S includes statements with expressions that include function calls preserves functional equivalence.*

7.5 Proof for Program Slicing (T_{ps}^{main})

This section presents a proof that the T_{ps}^{main} transformation preserves functional equivalence. This transformation builds a program slice for each data item being produced from a Category 4 or Category 5 subprogram using the projection function π. The collection of program slices produced duplicate the functionality of the original subprogram. This is expressed formally as shown below. Let Σ represent the sequence of statement from the original subprogram which produces data items a and b.

$$wp(\Sigma, R) \Leftrightarrow wp([\pi(\Sigma,\ a),\ \pi(\Sigma,\ b)])$$

The T_{ps}^{main} transformation uses the T_{ps}^{calls} transformation to replace each call to a Category 4 or Category 5 subprogram with a sequence of calls to the program slices built for the subprogram. The proof of the following theorem shows that these transformations preserve functional equivalence.

Theorem VII.13. *The transformation of GIM subprograms and subprogram calls using the T_{ps}^{calls} transformation preserves functional equivalence.*

Proof.

$$\begin{aligned}
&wp(T_{ps}^{calls}([< id_S, P_{in}, [a,b] >]), R) \Leftrightarrow \\
&\quad wp([< id_{S_a}, P_{in_a}, [a] >, < id_{S_b}, P_{in_b}, [b] >],\ R) \Leftrightarrow \\
&\quad wp(< id_{S_a}, P_{in_a}, [a] >,\ wp(< id_{S_b}, P_{in_b}, [b] >,\ R)) \Leftrightarrow \\
&\quad wp([\bar{x}_a := P_{in_a}, \pi(\Sigma, a), a := z],\ wp([\bar{x}_b := P_{in_b}, \pi(\Sigma, b), b := v],\ R)) \Leftrightarrow \\
&\quad wp([\bar{x}_a := P_{in_a}, \pi(\Sigma, a), a := z, \bar{x}_b := P_{in_b}, \pi(\Sigma, b), b := v],\ R) \Leftrightarrow \\
&\quad wp([\bar{x}_a := P_{in_a}, \bar{x}_b := P_{in_b}, \pi(\Sigma, a), a := z, \pi(\Sigma, b), b := v],\ R) \Leftrightarrow \\
&\quad wp([\bar{x}_a := P_{in_a}, \bar{x}_b := P_{in_b}, \pi(\Sigma, a), \pi(\Sigma, b), a := z, b := v],\ R) \Leftrightarrow \\
&\quad wp([\bar{x}_a := P_{in_a}\ \bar{x}_b := P_{in_b}], wp([\pi(\Sigma, a), \pi(\Sigma, b)], wp([a := z, b := v],\ R))) \Leftrightarrow \\
&\quad wp([\bar{x} := P_{in}], wp(\Sigma_S, wp([a := z, b := v],\ R))) \Leftrightarrow \\
&\quad wp([\bar{x} := P_{in}, \Sigma_S, a := z, b := v],\ R) \Leftrightarrow \\
&\quad wp(< id_S, P_{in}, [a,b] >,\ R)
\end{aligned}$$

□

7.6 Proof for Imperative Design (σ_F)

This section provides a proof that the object-oriented code extracted using the PBOI methodology is a consistent implementation of the legacy imperative code. As discussed in Section 7.1, the functionality of the imperative code is embodied at the highest level in the main program, F. The functionality of the object-oriented code is embodied at the highest level in the main method, F'. The σ_F transformation is used to transform F to F', and this section provides a proof that the σ_F transformation produces an F' that is functionally equivalent to F.

Theorem VII.14. *The σ_F transformation preserves functional equivalence.*

Proof.

$$\begin{aligned}
&wp(\sigma_F(F), \theta_e(R)) \Leftrightarrow \\
&\quad wp(\sigma(T_{ps}^{main}(F)),\ \theta_e(R)) \Leftrightarrow \\
&\quad \theta_e(wp(T_{ps}^{main}(F),\ R)) \Leftrightarrow \\
&\quad \theta_e(wp(F, R))
\end{aligned}$$

□

Since this proof shows that the σ_F transformation preserves functional equivalence, the object-oriented design extracted using the PBOI methodology is functionally equivalent to the legacy imperative design.

7.7 Summary

This chapter presented several definitions, theorems, and proofs in order to show that using the PBOI methodology results in an object-oriented design which is a consistent implementation of the original imperative design. The overall functionality of the imperative design was proven to be maintained in the extracted object-oriented design. This was done by proving that the σ_F transformation preserves functional equivalence. The following chapter presents a feasibility demonstration of the PBOI methodology.

VIII. Feasibility Demonstration

8.1 Introduction

This chapter discusses the proof-of-concept prototype developed to implement the PBOI methodology. As a feasibility demonstration, a legacy imperative system consisting of over 3000 lines of FORTRAN-77 code was converted to the object-oriented paradigm. This chapter describes the transformations used to convert FORTRAN-77 code to the GIM and presents a summary of the legacy system that was converted. A description of the extracted object-oriented design is presented at the end of the chapter.

8.2 Converting FORTRAN to the GIM

The GIM has been developed in order to provide a canonical form for representing imperative programming languages. As a proof-of-concept, automatic transformations from FORTRAN-77 to the GIM have been developed. These transformations were built using the *Software Refinery*^{TM} development environment. The *Refine/FORTRAN*^{TM} reverse engineering tool, part of *Software Refinery*^{TM}, includes a domain model and grammar for FORTRAN-77. Refine/FORTRAN parses in FORTRAN source code and builds Abstract Syntax Trees (ASTs) that store information about the source code. The transformations build new GIM ASTs based on these FORTRAN-77 ASTs.

The first step in re-engineering a legacy system is to parse each subprogram using Refine/FORTRAN^{TM}. Refine/FORTRAN^{TM} has a constraint that each of the subprograms must reside in its own file, and the files to be parsed are collected into a *system.* An AST is built for the system and each AST for a subprogram is part of this AST. After parsing, the system AST is stored in one file called the *analysis file.* The analysis file is loaded into the re-engineering prototype, so the transformations can be applied to each AST.

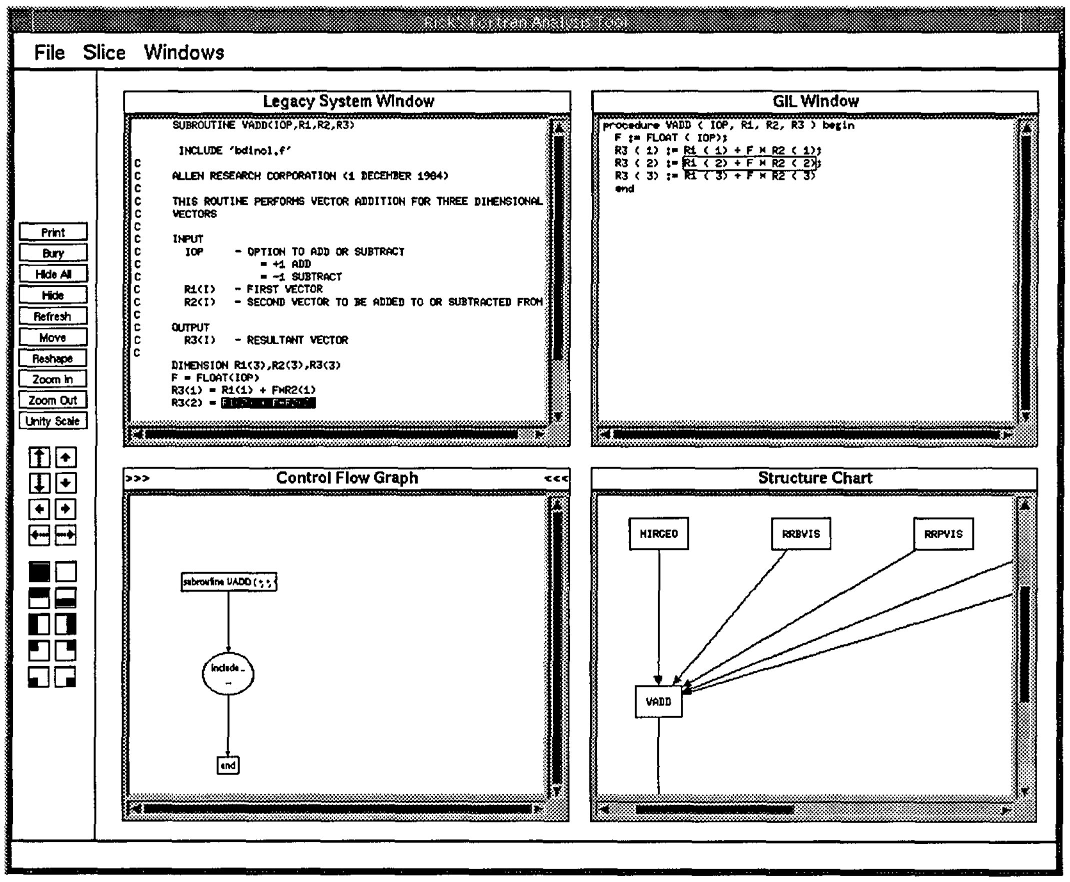
A typical transformation has the following form.

$$< \textit{FORTRAN AST} > \longrightarrow < \textit{GIM AST} >$$

The approach taken in the prototype is to generate a new GIM AST using the transformations, as opposed to transforming the existing FORTRAN ASTs. This means as FORTRAN ASTs are found that match the left-hand-side of the transformation, the GIM AST from the right-hand-side of the transformation is built. The identifiers on the left-hand-side indicate ASTs from the FORTRAN-77 domain model provided in the Refine/FORTRAN^{TM} User's Guide [55] and the identifiers on the right-hand-side indicate ASTs from the GIM domain model as defined in Chapter III. All of the transformations are applied to each node in the FORTRAN AST by using the REFINE *pre-order traversal* command. The traversal begins with the FORTRAN AST representing the overall system.

A Refine/FORTRAN^{TM} workbench has been built that displays a subprogram from the legacy system (using the FORTRAN-77 surface syntax) and the corresponding subprogram as represented in the GIM (using the GIL surface syntax). Figure 193 shows this workbench. The upper-left subwindow of the workbench shows the original FORTRAN legacy subroutine. The upper-right subwindow shows the GIM representation of this subroutine. The two subwindows are *hyper-linked* to show the connection between the legacy FORTRAN code and the representation of the code in the GIM. When the cursor is over an AST entity in the `GIL Window`, the surface syntax of the corresponding FORTRAN AST is highlighted in the `Legacy System Window`. In the figure, the cursor is over the GIM addition expression `R1 ( 2) + F * R2 ( 2)`, as indicated by the rubber-band box. The FORTRAN AST entity from which this expression was transformed is highlighted in the `Legacy System Window` using reverse video. Note that the entire FORTRAN subroutine is not visible in the upper-left subwindow, but the scroll bar can be used to view the rest of the subroutine. The workbench also shows the control flow graph and structure chart for the `VADD` subprogram. The structure chart is provided as part of the Refine/FORTRAN^{TM} workbench; the control flow graph is not.

Because of the limited documentation provided with Refine/FORTRAN^{TM}, it was difficult to determine the attributes of an AST node included in the FORTRAN-77 domain model without parsing a statement that builds the node. For this reason, only the transformations that were needed to convert the selected proof-of-concept legacy system were developed in the prototype.



```
      SUBROUTINE VADD(IOP,R1,R2,R3)

       INCLUDE 'bdinol.f'
C
C     ALLEN RESEARCH CORPORATION (1 DECEMBER 1984)
C
C     THIS ROUTINE PERFORMS VECTOR ADDITION FOR THREE DIMENSIONAL
C     VECTORS
C
C     INPUT
C       IOP      - OPTION TO ADD OR SUBTRACT
C                    = +1 ADD
C                    = -1 SUBTRACT
C       R1(I)    - FIRST VECTOR
C       R2(I)    - SECOND VECTOR TO BE ADDED TO OR SUBTRACTED FROM
C
C     OUTPUT
C       R3(I)    - RESULTANT VECTOR
C
      DIMENSION R1(3),R2(3),R3(3)
      F = FLOAT(IOP)
      R3(1) = R1(1) + F*R2(1)
      R3(2) = R1(2) + F*R2(2)
```

```
procedure VADD ( IOP, R1, R2, R3 ) begin
  F := FLOAT ( IOP);
  R3 ( 1) := R1 ( 1) + F * R2 ( 1);
  R3 ( 2) := R1 ( 2) + F * R2 ( 2);
  R3 ( 3) := R1 ( 3) + F * R2 ( 3)
  end
```

Figure 193   FORTRAN Workbench

Program level objects
EXECUTABLE-PROGRAM
PROGRAM-UNIT
END-STATEMENT
HEADER-STATEMENT
PROGRAM-STATEMENT
SUBROUTINE-STATEMENT
PARAMETER
Variables object
INDEXABLE-NAME
IDENTIFIER
IMPLICIT-STATEMENT
TYPE-STATEMENT
TYPE-BYTE
TYPE-CHARACTER
TYPE-DOUBLE-PRECISION
TYPE-INTEGER
TYPE-LOGICAL
TYPE-REAL
NAME-DECLARATOR
POSSIBLY-INITIALIZED-NAME
DIMENSION-DECLARATOR
SIMPLE-LENGTH-DECL
COMMON-DECLARATOR
COMMON-STATEMENT
EQUIVALENCE-DECLARATOR
EQUIVALENCE-STATEMENT
DIMENSION-STATEMENT
RECORD-VAR-DECL
STRUCT-REFERENCE
STRUCT-ACCESS

Table 2 Fortran ASTs Used

Expressions objects
ADD-EXPRESSION
AND-EXPRESSION
CONCATENATE-EXPRESSION
DIVIDE-EXPRESSION
EQ-EXPRESSION
EQV-EXPRESSION
EXPONENTIATE-EXPRESSION
GE-EXPRESSION
GT-EXPRESSION
LE-EXPRESSION
LT-EXPRESSION
MULTIPLY-EXPRESSION
NE-EXPRESSION
NEQV-EXPRESSION
OR-EXPRESSION
SUBTRACT-EXPRESSION
LITERAL-FALSE
LITERAL-TRUE
LITERAL-INTEGER
LITERAL-RIGHT-TEXT
LITERAL-DP
LITERAL-REAL
LITERAL-CHARSTRING
LITERAL-HOLLERITH
IDENTITY-EXPRESSION
NEGATE-EXPRESSION
NOT-EXPRESSION
NULL-EX
Assignment objects
ASSIGNMENT-STATEMENT
PARAMETER-STATEMENT
Control flow statement objects
CONTINUE-STATEMENT
RETURN-STATEMENT
INCLUDE-STATEMENT
LABEL-DEFINITION
LABEL-USE
UNCONDITIONAL-GOTO
CALL-STATEMENT
BLOCK-IF-STATEMENT
LOGICAL-IF-STATEMENT
IF-THEN-ELSE-STATEMENT
DO-STATEMENT

Table 3 Fortran ASTs Used (cont)

I/O statements
OPEN-STATEMENT
READ-STATEMENT
WRITE-STATEMENT
PRINT-STATEMENT
FORMAT-STATEMENT
FORMAT-COMMA-SEPARATOR
FORMAT-CLOSER
UNIT-IDENTIFIER
FORMAT-IDENTIFIER
IO-FULL-SPECIFIER
LIT-STRING-FORMAT
FORMAT-SLASH-SEPARATOR
FORMAT-DOUBLE-SLASH
X-FORMAT
I-FORMAT
F-FORMAT
E-FORMAT
O-FORMAT
Z-FORMAT
R-FORMAT
A-FORMAT

Table 4 Fortran ASTs Used (cont)

Table 2, Table 3, and Table 4 show the domain model ASTs that are included in the prototype. The other ASTs shown in Chapter 14 of the Refine/FORTRAN^{TM} User's Guide [54] were not used. A separate process was used to evaluate the completeness of the transformation after the pre-order traversal process completed execution. This process is described in detail in Section 8.3.

The following sections describe the transformations that have been developed and included in the prototype. These transformations convert FORTRAN statements, variables, and expressions to the GIM.

8.2.1 FORTRAN Assignment Statement Transformation. Variables in FORTRAN are assigned values using the assignment statement and the data statement. The transformation of these two statements is shown in Figure 194.

$$\begin{array}{rcl} \text{assignment-statement} & \longrightarrow & \text{imperative-assignment} \\ \text{data-statement} & \longrightarrow & \text{imperative-assignment} \end{array}$$

Figure 194 Assignment Transformations

The following FORTRAN assignment statement is represented using the FORTRAN `assignment-statement` AST.

```
RM = VMAG(R)
```

The variable on the left-hand-side of this assignment is converted to a GIM variable and stored in the left-hand-side attribute of the GIM `imperative-assignment` AST. The expression on the right-hand-side of this assignment is converted to a GIM expression and stored in the right-hand-side attribute of the GIM AST. The GIM AST built from this transformation is shown below using GIL syntax.

```
RM := VMAG ( R);
```

The FORTRAN AST that represents the following `DATA` statement is transformed to the GIM by building two `imperative-assignment` ASTs.

```
DATA TWOPI/6.2831853072D0/ ,XMU/398601.2D0/
```

Each data item being initialized is converted to a GIM variable and stored in the left-hand-side attribute of a GIM `imperative-assignment` AST. The corresponding value for each data item is converted to a GIM expression and stored as the right-hand-side of the assignment statement. The two GIM assignment statement ASTs generated for this `data` statement are shown below using GIL syntax.

```
TWOPI := 6.2831853072D0;
XMU := 398601.2D0;
```

Because of the syntactic complexity allowed for data statements, a restriction is placed on the data statements that are transformed using the prototype. Specifically, the number of the data *elements* being assigned values must match the number of the data *constants* provided.

8.2.2 FORTRAN Sequential Control Flow Transformation. Sequential control flow in the FORTRAN AST is represented by storing statements in sequences. Sequences are also used in the GIM to store the ASTs built from the transformation of these statements. Specifically, statements in a sequence are transformed one after another (sequentially), and the corresponding GIM AST built from the transformation is appended to the corresponding sequence in the GIM. Sequences are used in FORTRAN ASTs to represent the statements in subprograms, if-then-else statements, and looping statements.

8.2.3 FORTRAN Selective Control Flow Transformation. Selective control flow in the FORTRAN programming language is implemented by the `if-then-else` statement, the `if-then` statement, and the `logical-if` statement. Since an `if-then` statement is an `if-then-else` statement without an `else` part, both statements are represented using the REFINE/FORTRAN `block-if-statement` AST.

Figure 195 shows the transformations that have been built in the prototype to convert FORTRAN selective control flow.

For example, consider the following FORTRAN `if-then-else` statement.

block-if-statement $\longrightarrow$ imperative-selection

logical-if-statement $\longrightarrow$ imperative-selection

Figure 195 Selective Control Flow Transformations

```
IF(ITYPE.LT.0) THEN
CALL CSP(RT,RBO,TFBOT,XMU,VBOPP)
ELSE
CALL CSP(RBO,RT,TFBOT,XMU,VBO)
END IF
```

The attributes of the FORTRAN `block-if-statement` AST that represents this statement are used to build the GIM `imperative-selection` AST shown below (using GIL syntax).

```
if ITYPE < 0 then
  CSP ( RT, RBO, TFBOT, XMU, VBOPP);
else
  CSP ( RBO, RT, TFBOT, XMU, VBO)
endif
```

The FORTRAN expression is converted to a GIM expression and stored as the expression of the `imperative-selection` AST. Each statement in the `then` part is converted and stored in the `then` part of the `imperative-selection` AST. Each statement in the `else` part is also converted and stored in the `else` part of the `imperative-selection` AST.

The transformation of the FORTRAN `if-then` statement is only slightly different. The following statement is represented using the FORTRAN `block-if-statement` AST.

```
IF(THETA.GT.PI) THEN
ITYPE = -ABS(ITYPE)
THETA = 2.DO*PI - THETA
END IF
```

In this case, the `else` part of statement is empty, which is represented in the FORTRAN AST as an empty sequence. The expression from this AST is converted and stored as before, and the statements from the `then` part are converted and stored. The empty `else` part of this statement is also represented as an empty sequence in the GIM AST. The GIM AST is shown below using GIL syntax.

```
if THETA > PI  then
  ITYPE := -ABS ( ITYPE);
  THETA := 2.0d0 * PI - THETA
```

```
else
endif;
```

The GIL syntax for the `if-then` statement is not like the FORTRAN syntax because the GIL syntax includes the `else` keyword even if there are no statements in the `else` part of an `if-then-else` statement.

The FORTRAN `logical-if` statement provides a short-hand syntax for the `if-then` statement. The following statement is represented using the FORTRAN `logical-if` AST.

```
        IF(ALT.LT.0.) ALT = 0
```

As with the `if-then` statement, the expression is converted to a GIM expression and stored. The `logical-if` always includes a single statement, so this statement is converted and stored in a sequence in the `then` part of the GIM AST. An empty sequence is stored in the `else` part of the AST as was done with the `if-then` statement. The GIM AST is shown below using GIL syntax.

```
if ALT < 0 then
  ALT := 0
else
endif;
```

The FORTRAN programming language also provides a way to *nest* `if-then-else` statements together into one statement. For example, the following FORTRAN statement is represented using a REFINE/FORTRAN `block-if-statement` AST.

```
IF(ITYP.EQ.1) THEN
ALT =  -5.71D0
ELSE IF(ITYP.EQ.2) THEN
ALT = -1.57D0
ELSE IF(ITYP.EQ.3) THEN
ALT = -9.46D0
END IF
```

Note that there is only one `END IF` keyword even though there are multiple `IF` clauses in this statement. In the AST, the `ELSE IF` branches of the statement are collected into a sequence of `block-if-statements`. Since not all imperative languages provide such a way to nest `if-then-else` statements, this statement is transformed into an equivalent collection of *embedded* `if-then-else` statements. The transformed statement is shown below.

```
if ITYP = 1 then
  ALT := -5.71d0;
else
  if ITYP = 2 then
    ALT := -1.57d0;
  else
    if ITYP = 3 then
      ALT := -9.46d0;
    else
    endif
  endif
endif;
```

At the top level, this is a single `if-then-else` statement with a second `if-then-else` statement embedded in the `else` clause. A third `if-then-else` statement is embedded in the `else` clause of the second `if-then-else` statement.

Appendix C shows a proof that the transformation of nested `if-then-else` statements to embedded `if-then-else` statements maintains the functionality of the original FORTRAN statement.

8.2.4 FORTRAN Iterative Control Flow Transformation. Structured iterative control flow in the FORTRAN language is implemented using the `do` statement and the proper combination of the `if-then` statement and the `goto` statement. Let $\xrightarrow{goto}$ represent the transformation that only builds a GIM `imperative-iteration` AST if this proper combination of statements is used. The transformations built in the prototype for converting FORTRAN iterative control flow constructs are shown in Figure 196.

$$\begin{array}{rcl} \text{do-statement} & \longrightarrow & \text{imperative-iteration} \\ \text{block-if-statement} & \xrightarrow{goto} & \text{imperative-iteration} \end{array}$$

Figure 196 Iterative Control Flow Transformations

The following `do` statement is represented using the FORTRAN `do-statement` AST.

```
      DO 100 I=1,3
      U(I) = R(I)/RM
  100 CONTINUE
```

The attributes of this AST are used to build the GIM `imperative-iteration` AST as follows. The label `100` is used to indicate syntactically which statements will be iterated. The FORTRAN AST has an attribute that stores this sequence of statements, so each statement is converted to the GIM and stored in the `imperative-iteration` AST. The loop variable syntax `I=1,3` is represented by an attribute for the loop variable, the initial value and the final value. A GIM `imperative-assignment` AST is generated to assign the initial value to the loop variable and is returned as part of the `imperative-iteration` AST transformation. This, in effect, inserts an assignment statement into the sequence of GIM statements right before the `imperative-iteration` statement. The expression built for the GIM `imperative-iteration` AST is always a less-than-or-equal expression that tests whether the loop variable is less than or equal to the final value. The transformed GIM `imperative-iteration` AST is shown below using the GIL syntax.

```
I := 1;
while I <= 3 do
  begin
    U ( I) := R ( I) / RM;
    I := I + 1
  end
```

The only `if-then` statements transformed into `imperative-iteration` ASTs have a single `goto` statement as the last statement of the `then` clause. This `goto` statement must cause program control to return to the line that includes the `if-then` statement. For example, the following `if-then` statement has such a `goto` statement.

```
150 IF(C2.GT.1.0D0) THEN
    E = E - DE
    DE = DE/2.
    E = E + DE
    C1 = 1.0D0 - E**2
    C2 = C1/C2GAM
    GO TO 150
    END IF
```

This `if-then` statement is transformed into the following GIM iterative statement.

```
while C2 > 1.0d0 do begin
  E := E - DE;
  DE := DE / 2;
  E := E + DE;
  C1 := 1.0d0 - E^2;
  C2 := C1 / C2GAM
  end;
```

8.2.5 FORTRAN Subprogram Transformation. Imperative subprograms are implemented in the FORTRAN language by the **`subroutine`** statement and the **`function`** statement. Subroutines are invoked using the FORTRAN `call` statement. Functions are invoked by using the function's identifier in an expression. Let $\xrightarrow{call}$ represent the transformation that only builds a GIM **`imp-function-call`** AST if the identifier refers to a function begin invoked. The transformations that have been built in the prototype to convert these entities are shown in Figure 197.

$$\begin{aligned} \text{subroutine-statement} &\longrightarrow \text{imperative-procedure} \\ \text{function-statement} &\longrightarrow \text{imperative-function} \\ \text{call-statement} &\longrightarrow \text{imp-procedure-call} \\ \text{identifier} &\xrightarrow{call} \text{imp-function-call} \end{aligned}$$

Figure 197 Subprogram Transformations

```
      SUBROUTINE VADD(IOP,R1,R2,R3)

       INCLUDE 'bdincl.f'
C
C      THIS ROUTINE PERFORMS VECTOR ADDITION FOR THREE DIMENSIONAL
C       VECTORS
C
      DIMENSION R1(3),R2(3),R3(3)
      F = FLOAT(IOP)
      R3(1) = R1(1) + F*R2(1)
      R3(2) = R1(2) + F*R2(2)
      R3(3) = R1(3) + F*R2(3)
      RETURN
      END
```

Figure 198 FORTRAN Subroutine VADD

For example, Figure 198 shows the FORTRAN subroutine VADD. The attributes of the FORTRAN AST built for VADD are used to build an imperative-procedure AST in the GIM as follows. The subroutine identifier is converted into a GIM variable and stored as the identifier of the imperative-procedure AST. The formal parameters IOP, R1, R2, R3 are converted into GIM variables and stored (in the same order) in the sequence of formal parameters for the GIM AST. Each statement from the subroutine is converted into a GIM statement and stored (in the same order) in the sequence of statements for the imperative-procedure AST. The GIM AST built from this transformation is shown below using GIL syntax.

```
procedure VADD ( IOP, R1, R2, R3 )
  begin
  F := FLOAT ( IOP);
  R3 ( 1) := R1 ( 1) + F * R2 ( 1);
  R3 ( 2) := R1 ( 2) + F * R2 ( 2);
  R3 ( 3) := R1 ( 3) + F * R2 ( 3)
  end
```

Note that the comments from the FORTRAN subroutine are not modeled in the GIM AST. There is a attribute in the REFINE/FORTRAN AST that stores the comments from a subroutine, but no attempt is made in this research to carry the comments forward. The FORTRAN call statement that invokes VADD is shown below.

```
CALL VADD(-1,RB,R(1,J,K),RRB)
```

The call identifier from the FORTRAN AST is converted to a GIM variable and stored as the identifier of the imp-procedure-call AST. The sequence of actual parameters is converted to a sequence of GIM variables (with the same order) and stored as the actual parameters in the GIM AST. Because of Restriction III.3, an extra step is performed in this transformation to ensure all actual parameters are variables. Specifically, the -1 parameter in the call to VADD is replaced by a temporary variable and an assignment statement is inserted before the call. The two GIM ASTs built from this transformation are shown below using GIL syntax.

```
TEMP-29 := -1;
VADD ( TEMP-29, RB, R ( 1, J, K), RRB);
```

8.2.6 FORTRAN Variable Transformation. Each FORTRAN variable in the legacy system must be converted to a GIM variable. The representation of variables in the GIM consists of two parts. An `imperative-variable` AST is built for each variable and stored in the Imperative Symbol Table (IST). All of the information needed to build an `imperative-variable` is taken from the FORTRAN symbol table generated during compilation. The symbol table includes the variable's identifier, scope, data type, and a constant value (if any). If the FORTRAN variable is an array, the indices are not stored with the variable since the entire array is considered one variable.

References to a FORTRAN variable are modeled using the `imperative-name` AST shown in Figure 199.

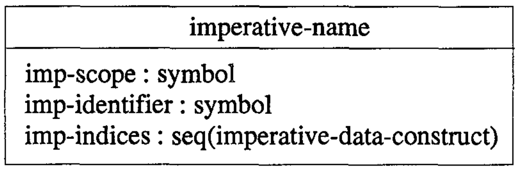


Figure 199 Imperative Name Class

A new instance of this class is built for each reference to a FORTRAN variable. If the variable is an array, the array indices are converted to GIM expressions and stored in the `imperative-name` object. When a FORTRAN variable is transformed, the IST is checked to see if an `imperative-variable` AST has already been built to represent the FORTRAN variable. If not, an `imperative-variable` is built and stored in the IST.

Using the `imperative-name` AST was meant to simplify the process of representing variables. However, in hind-sight, it is just as easy to represent each FORTRAN variable using an instance of the `imperative-variable` class. The only difference between an `imperative-name` AST and an `imperative-variable` AST is that an `imperative-name` doesn't store the data type and an `imperative-variable` doesn't store the indices of an array variable. These two ASTs should be combined into one.

Because of the information provided by the FORTRAN symbol table, declarations of variables are not transformed or modeled in the GIM. This means FORTRAN `common` statements and `dimension` statements have no corresponding construct in the GIM. Since there is no modeling of variable declarations in the GIM, a system built to convert GIM ASTs into an imperative language such as C would have to automatically generate the appropriate declarations for each variable as required by the language. There is no surface syntax provided for a GIM `imperative-variable` ASTs, only `imperative-name` ASTs.

8.2.7 FORTRAN Data Types Transformation. The transformations built to convert FORTRAN data types (except implicit types and built-in types) are shown in Figure 200.

type-integer	⟶	imperative-integer
type-byte	⟶	imperative-integer
type-real	⟶	imperative-real
type-double-precision	⟶	imperative-real
type-logical	⟶	imperative-boolean
type-character	⟶	imperative-string
array-type	⟶	imperative-array

Figure 200 Data Type Transformations

For implicit types, if the `type-expr-name` attribute of the `type-expr-type` attribute of an `implicit-type` FORTRAN AST equals the symbol `real`, then an `imperative-real` GIM AST is built. If the attribute equals the symbol `integer`, an `imperative-integer` GIM AST is built. Similarly, if the `type-expr-name` attribute of a `built-in-type` FORTRAN AST equals the symbol `real`, then an `imperative-real` GIM AST is built. If the attribute equals the symbol `integer`, an `imperative-integer` GIM AST is built.

8.2.8 FORTRAN Expression Transformation. A FORTRAN expression can be either a variable, a binary expression, a unary expression, or a literal constant value. This section describes each of the transformations for these kinds of expressions. The transformation of a variable is described in Section 8.2.6. The transformations included in the prototype for FORTRAN binary expressions are shown in Figure 201. The transformations built to convert FORTRAN unary expressions are shown in Figure 202. The transformations built to convert FORTRAN literal expressions are shown in Figure 203.

add-expression	$\longrightarrow$	imperative-addition
and-expression	$\longrightarrow$	imperative-and
concatenate-expression	$\longrightarrow$	imperative-concat
divide-expression	$\longrightarrow$	imperative-division
eq-expression	$\longrightarrow$	imperative-equal
eqv-expression	$\longrightarrow$	imperative-equal
exponentiate-expression	$\longrightarrow$	imperative-exponent
ge-expression	$\longrightarrow$	imperative-greater-than-or-equal
gt-expression	$\longrightarrow$	imperative-greater-than
le-expression	$\longrightarrow$	imperative-less-than-or-equal
lt-expression	$\longrightarrow$	imperative-less-than
multiply-expression	$\longrightarrow$	imperative-multiplication
ne-expression	$\longrightarrow$	imperative-not-equal
neqv-expression	$\longrightarrow$	imperative-not-equal
or-expression	$\longrightarrow$	imperative-or
subtract-expression	$\longrightarrow$	imperative-subtraction

Figure 201 Binary Expression Transformations

negate-expression	$\longrightarrow$	imperative-negate
not-expression	$\longrightarrow$	imperative-not
null-ex	$\longrightarrow$	imperative-null

Figure 202 Unary Expression Transformations

8.2.9 FORTRAN Input/Output Transformation. Input is implemented in the FORTRAN language by the `read` statement. Output is implemented by the `write` and `format` statements. The transformations built in the prototype to convert FORTRAN input and output statements are shown in Figure 204.

8.3 Completeness of the Transformation

Since transformations were not built for the entire domain model provided by REFINE/FORTRAN, a process was defined to assess the completeness of the transformation. Specifically, as the FORTRAN AST was being transformed, each transformation set a flag

literal-false	⟶	imperative-literal-boolean
literal-true	⟶	imperative-literal-boolean
literal-hex	⟶	imperative-literal-integer
literal-integer	⟶	imperative-literal-integer
literal-octal	⟶	imperative-literal-integer
literal-real	⟶	imperative-literal-real
literal-quad	⟶	imperative-literal-real
literal-double	⟶	imperative-literal-real
literal-charstring	⟶	imperative-literal-charstring
literal-hollerith	⟶	imperative-literal-charstring
literal-right-text	⟶	imperative-literal-charstring

Figure 203 Literal Expression Transformations

read-statement	⟶	imperative-input
write-statement	⟶	imperative-output
format-statement	⟶	imperative-output

Figure 204 Input and Output Transformations

on the AST node being transformed. A separate traversal of the FORTRAN AST examines each node and report any nodes not transformed. This process proved to be quite effective and uncovered problems with the overall transformation that were easily corrected. An excerpt from the transcript generated by the conversion of the `ANG` subprogram is shown below.

```
Converting "ANG" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 60
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file...
Saving surface syntax in file...
```

This transcript reports the conversion of ANG is 100% complete. There are 60 tree nodes in the ANG FORTRAN AST, and the new AST is saved to the Persistent Object Base (POB). The complete transcript from the conversion of the 3000 line FORTRAN-77 legacy system is shown in Appendix E.

8.4 The Ballistic Missile Defense Simulation System

The legacy imperative system that was selected for the feasibility demonstration is the Ballistic Missile Defense Simulation (BMDSIM) system [36]. This system consists of 53 subprograms including the main program, BMDSIM1. This system was developed for use as a *real* production system and is not a toy system developed just for this research.

Table 5 and Table 6 show a summary of these subprograms. The type of subprogram (function or procedure), the subprogram identifier, the category[1] into which this subprogram is classified, and the set of data items produced are presented in the table.

Some of the subprograms in BMDSIM were re-structured to eliminate unstructured uses of GOTO statements. There were no global variables in the system and none of the parameters were both input and output parameters. The RAND function had an output parameter, so the function was converted into a procedure with two output parameters.

After these changes, the total number of lines of code was 3109.

8.5 Eliminating Category 4 and Category 5 subprograms

As described in Chapter VI, the first step in converting a legacy system is to eliminate the Category 4 and Category 5 subprograms by using program slicing. Figure 205 shows how the original 53 subprograms are classified according to the taxonomy of subprograms. After program slicing, the BMDSIM system consisted of 105 subprograms including the main program. Figure 205 also shows how each of these 105 subprograms are classified.

Tables 7, 8, and 9 show a summary of the new subprograms generated by program slicing. The tables show the name of each subprogram in BMDSIM. For any Category 4 or

[1]One of the six categories from the taxonomy of imperative subprograms presented in Chapter V.

Type	Subprogram	Cat	Data Items Produced
Function	ANG	2	
Function	ANGLE	3	
Procedure	ASSIGN	5	DWELLA, IENG, IZTAB, LLAS, LMIRS, NBPCR, RBENG, TLAS, TMIRS
Main Program	BMDSIM1	1	
Procedure	BOOSTR	4	R, V
Procedure	BOSTIT	5	AR, ITYPE, RBO, RL, RT, TBO, TFBOT, VBO, VI
Function	BOUNCE	3	
Function	CAPTURE	2	
Procedure	CROSS	2	
Procedure	CSP	3	
Function	CUV	2	
Procedure	DASET	4	DALASM, DWELLT, IASGN
Function	DOT	2	
Procedure	KEP	4	R, V
Procedure	LASP	4	BFLU, BFLUX, TD
Procedure	LNKCAL	5	DALASM, DWELLT, IASGN, PHASE
Procedure	LNKCK	4	MIRF, RANGE
Procedure	LNKORD	4	DWELLA, DWELLT, LIASGN
Procedure	MAT	2	
Procedure	MAXA	4	AMAX, NMAX
Procedure	MIRGEO	3	
Procedure	MIRVIS	5	MIRF, MIRR
Procedure	MTM2	3	
Procedure	MTM3	3	
Procedure	MTPD	2	
Procedure	MTRT	2	
Procedure	ORBEL	4	A, E

Table 5 Subprograms from BMDSIM

Type	Subprogram	Cat	Data Items Produced
Function	PKILL	2	
Procedure	POSVEC	2	
Procedure	POSVECS	2	
Function	PRDIV	2	
Function	RADIUS	2	
Procedure	RAND	4	RANDVAL, RSEED
Procedure	RELAY	5	AREA, DALAS, EBRITE, EFFNCY, PHASE
Function	RHO	2	
Procedure	RRBVIS	5	RIANG, RRBM, SLANG
Procedure	RRPVIS	5	RPANG, RRPM
Function	RTAN	2	
Procedure	SBMIT	5	DETAO, DRAO, ETAO, ORATE, RAO, RSEED
Procedure	SBMLOC	2	
Procedure	SBMPOS	5	ETA, R, RA
Procedure	SELECL	4	IDX, ILS, J1S, J2S, K1S, K2S, TNGAGE
Function	SGN	2	
Function	SUV	2	
Procedure	TFBT	2	
Procedure	TPANG	5	ALP, THE, XINC
Procedure	TRAJ	5	R, V
Procedure	UNIT	3	
Function	UPLREQ	2	
Function	UPTRNS	2	
Procedure	VADD	2	
Procedure	VISCK	5	MCK, RRM
Function	VMAG	2	

Table 6 More Subprograms from BMDSIM

Category	Before Slicing	After Slicing
0	0	0
1	1	1
2	22	52
3	8	52
4	9	0
5	13	0
Total	53	105

Figure 205 Subprograms Classified

Category 5 subprogram, the names of the new subprograms built from it and the category of the new subprogram are shown. The final column indicates on which of the slices the process of *masking* had to be used.

8.6 Implementing the PBOI

The next step in the conversion process is to transform the subprograms represented in the GIM into classes and methods represented in the GOM. This is done using the PBOI methodology described in Chapter V. Each of the PBOI transformations was implemented in the prototype using the REFINE programming language. Every attempt was made to match the names of the REFINE functions with the transformations they implement. The mappings from GIM AST entities to GOM AST entities are shown in Appendix F.

A Refine^{TM} workbench has been built that displays both the imperative subprogram (shown using GIL syntax) and the object-oriented class and method (shown using GOL syntax) extracted from this subprogram. Figure 206 shows an example of this workbench. The upper-left subwindow of the workbench shows the GIL representation of the GIM subprogram being transformed. The subprogram is transformed manually using the **sigma** option (not shown) from the **Transformations** menu. Once the subprogram is transformed, the upper-right subwindow displays the extracted class and method. The lower-left subwindow displays the object-oriented design (that has been extracted to this point) using class diagrams. The class diagram shows a class as a box displaying the name of the class, the name of its methods, and the name and type of its attributes. Finally, the lower-right subwindow displays the AST that represents the overall design. The GOM

Subprogram	Cat	Slices	Cat	Masked
ANG	2			
ANGLE	3			
ASSIGN	5	ASSIGN-DWELLA	3	X
		ASSIGN-IENG	3	X
		ASSIGN-IZTAB	3	X
		ASSIGN-LLAS	3	X
		ASSIGN-LMIRS	3	X
		ASSIGN-NBPCR	3	X
		ASSIGN-RBENG	3	X
		ASSIGN-TLAS	3	X
		ASSIGN-TMIRS	3	X
BMDSIM1	1			
BOOSTR	4	BOOSTR-R	2	
		BOOSTR-V	2	
BOSTIT	5	BOSTIT-AR	3	X
		BOSTIT-ITYPE	3	X
		BOSTIT-RBO	3	X
		BOSTIT-RL	3	
		BOSTIT-RT	3	X
		BOSTIT-TBO	2	
		BOSTIT-TFBOT	3	X
		BOSTIT-VBO	3	X
		BOSTIT-VI	2	
BOUNCE	3			
CAPTURE	2			
CROSS	2			
CSP	3			
CUV	2			
DASET	4	DASET-DALASM	2	
		DASET-DWELLT	2	
		DASET-IASGN	2	
DOT	2			
KEP	4	KEP-R	2	
		KEP-V	2	
LASP	4	LASP-BFLU	2	
		LASP-BFLUX	2	
		LASP-TD	2	X

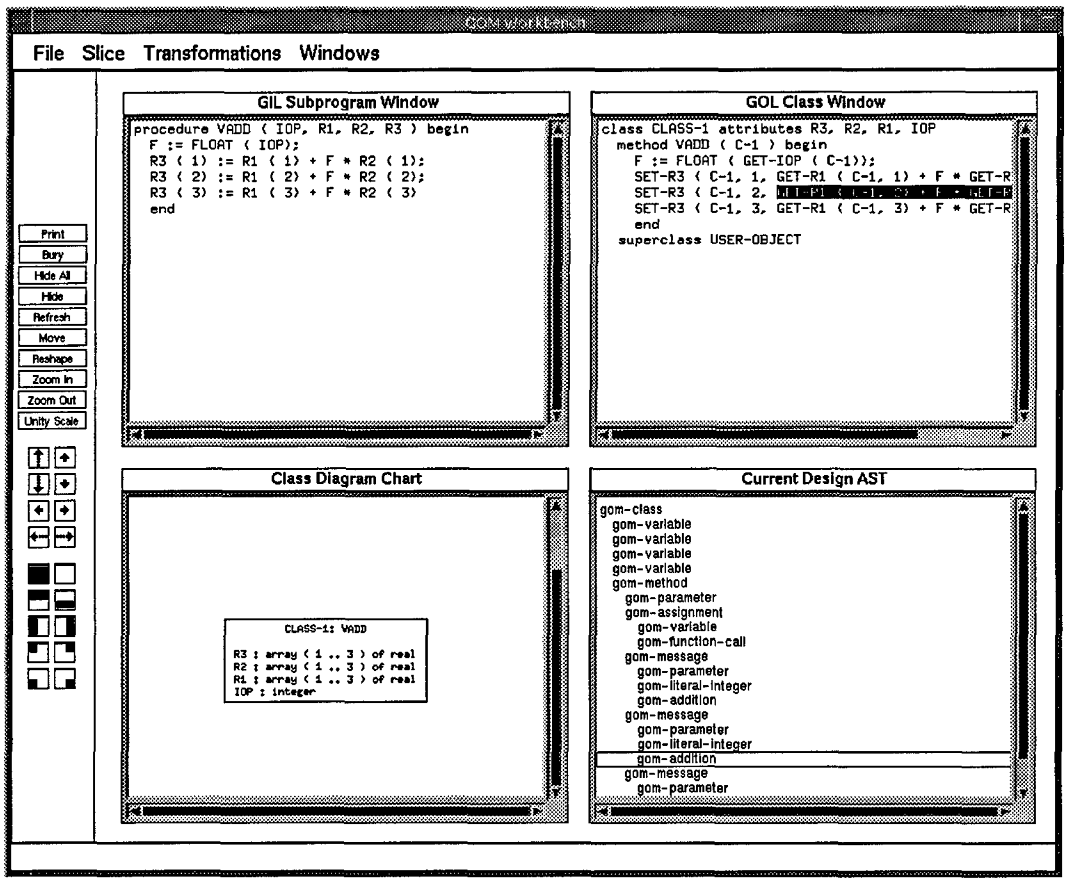
Table 7 Slices for Category 4 and Category 5 Subprograms

Subprogram	Cat	Slices	Cat	Masked
LNKCAL	5	LNKCAL-DALASM	3	X
		LNKCAL-DWELLT	3	
		LNKCAL-IASGN	3	X
		LNKCAL-PHASE	3	
LNKCK	4	LNKCK-MIRF	2	
		LNKCK-RANGE	?	
LNKORD	4	LNKORD-DWELLA	2	X
		LNKORD-DWELLT	2	
		LNKORD-LIASGN	2	X
MAT	2			
MAXA	4	MAXA-AMAX	2	
		MAXA-NMAX	2	X
MIRGEO	3			
MIRVIS	5	MIRVIS-MIRF	?	
		MIRVIS-MIRR	3	
MTM2	3			
MTM3	3			
MTPD	2			
MTRT	2			
ORBEL	4	ORBEL-A	2	X
		ORBEL-E	2	X
PKILL	2			
POSVEC	2			
POSVECS	2			
PRDIV	2			
RADIUS	2			
RAND	4	RAND-RANDVAL	2	X
		RAND-RSEED	2	
RELAY	5	RELAY-AREA	3	
		RELAY-DALAS	3	X
		RELAY-EBRITE	3	
		RELAY-EFFNCY	3	
		RELAY-PHASE	3	
RHO	2			

Table 8 More Slices for Category 4 and Category 5 Subprograms

Subprogram	Cat	Slices	Cat	Masked
RRBVIS	5	RRBVIS-RIANG	3	
		RRBVIS-RRBM	3	
		RRBVIS-SLANG	3	
RRPVIS	5	RRPVIS-RPANG	3	
		RRPVIS-RRPM	3	
RTAN	2			
SBMIT	5	SBMIT-DETAO	3	X
		SBMIT-DRAO	3	X
		SBMIT-ETAO	3	X
		SBMIT-ORATE	2	
		SBMIT-RAO	3	X
		SBMIT-RSEED	3	
SBMLOC	2			
SBMPOS	5	SBMPOS-ETA	3	
		SBMPOS-R	3	
		SBMPOS-RA	3	
SELECL	4	SELECL-IDX	2	
		SELECL-ILS	2	
		SELECL-J1S	2	
		SELECL-J2S	2	
		SELECL-K1S	2	
		SELECL-K2S	2	
		SELECL-TNGAGE	2	
SGN	2			
SUV	2			
TFBT	2			
TPANG	5	TPANG-ALP	3	
		TPANG-THE	3	X
		TPANG-XINC	3	
TRAJ	5	TRAJ-R	3	
		TRAJ-V	3	
UNIT	3			
UPLREQ	2			
UPTRNS	2			
VADD	2			
VISCK	5	VISCK-MCK	3	
		VISCK-RRM	?	
VMAG	2			

Table 9 More Slices for Category 4 and Category 5 Subprograms



```
procedure VADD ( IOP, R1, R2, R3 ) begin
  F := FLOAT ( IOP);
  R3 ( 1) := R1 ( 1) + F * R2 ( 1);
  R3 ( 2) := R1 ( 2) + F * R2 ( 2);
  R3 ( 3) := R1 ( 3) + F * R2 ( 3)
  end
```

```
class CLASS-1 attributes R3, R2, R1, IOP
  method VADD ( C-1 ) begin
    F := FLOAT ( GET-IOP ( C-1));
    SET-R3 ( C-1, 1, GET-R1 ( C-1, 1) + F * GET-R
    SET-R3 ( C-1, 2, GET-R1 ( C-1, 2) + F * GET-R
    SET-R3 ( C-1, 3, GET-R1 ( C-1, 3) + F * GET-R
    end
  superclass USER-OBJECT
```

```
CLASS-1: VADD

R3 : array ( 1 .. 3 ) of real
R2 : array ( 1 .. 3 ) of real
R1 : array ( 1 .. 3 ) of real
IOP : integer
```

```
gom-class
  gom-variable
  gom-variable
  gom-variable
  gom-variable
  gom-method
    gom-parameter
    gom-assignment
      gom-variable
      gom-function-call
    gom-message
      gom-parameter
      gom-literal-integer
      gom-addition
    gom-message
      gom-parameter
      gom-literal-integer
      gom-addition
    gom-message
      gom-parameter
```

Figure 206 GOM Workbench

ASTs shown in this subwindow are hyper-linked with the GOL surface syntax shown in the upper-right subwindow.

Each of the 105 Category 2 and Category 3 subprograms from the BMDSIM system was converted to the object-oriented paradigm using the prototype. This resulted in an object-oriented design with 104 classes and 104 methods. The main program was also converted which added another class and method to the design.

During the conversion of the main program, 770 create messages were built for 511 distinct object instances. After removing *duplicate* objects, 109 create messages remained for 109 distinct object instances. After removing *overlapping* classes, the design consisted of 43 classes. Of these classes, 33 classes had only one method, 6 classes had between 2 and 11 methods, and 4 classes had between 29 and 47 methods.

In order to address the *reasonableness* of the objects being extracted, an informal semantic analysis was done. The comments in the BMDSIM system were examined to develop a rudimentary domain model. The extracted objects were compared against this domain model in order to find a correlation between the objects and a semantic entity from the domain. This analysis found that the extracted objects tended to contain attributes of a single domain entity, as opposed to multiple domain entities. The number of attributes in the extracted objects tended to be less than the number of attributes in the domain entity. That is, the extracted objects described pieces of the domain entities. This could be due to the rudimentary nature of the domain model developed. The entire methodology described to this point has been fully automated. To add semantics to the extracted objects may require a cooperative effort involving the analyst. Further analysis of the semantic nature of the extracted objects is left as future research.

8.7 Analysis

The prototype was built on a SUN SPARC 5 workstation with 128MB of memory. Using the prototype, the conversion of BMDSIM to the GIM takes less than 20 minutes and the elimination of Category 4 and Category 5 subprograms takes approximately one hour. The prototype converts GIM subprograms to the GOM using the PBOI methodology (except for merging overlapping classes) in approximately 24 hours. Merging overlapping classes using the prototype requires an additional 12 hours. Because of its inefficiency, using the prototype to convert millions of lines of imperative code is not feasible.

8.8 Summary

This chapter has presented the results of the feasibility demonstration. A summary of the BMDSIM system and its subprograms was provided along with the subprograms resulting from program slicing. The specific transformations used to transform FORTRAN to the GIM were described in this chapter. The object-oriented design extracted from BMDSIM was also described. This transformation of BMDSIM using the prototype implementation demonstrates the conversion of a FORTRAN system to an object-oriented design is technically feasible.

IX. Contributions

9.1 Introduction

This chapter lists the contributions this research makes to the field of re-engineering.

9.2 Major Contributions

This research makes the following major contributions.

1. The Generic Imperative Model (GIM).
2. The Generic Object-Oriented Design Model (GOM).
3. The Parameter-Based Object Identification (PBOI) methodology.
4. The transformations that formalize the PBOI methodology.
5. A proof that the object-oriented design is a *consistent* implementation of the legacy code.

The GIM is significant because it provides a canonical form for imperative programming languages. This allows legacy code that does the same to look the same. There are several advantages to using the GIM in this research or other research involving imperative programming languages. First, the GIM is programming language independent, which means the representation is not tied to one specific programming language. By using the GIM in the prototype, legacy code in another language can be re-engineered once the transformations from that language to the GIM are developed. In this way, the GIM allows the prototype to be easily extended to other languages. Second, the GIM is program construct independent. This allows the GIM to represent imperative control flow structures in easily understood canonical forms. For example, there are many programming language implementations of the *selective* control flow construct from imperative languages, but the GIM is able to represent if-then statements, if-then-else statements, nested and embedded if-then-else statements, and case statements using a single control flow construct. This greatly simplifies the task of building formal transformations and prototypes based on the GIM constructs. Third, the GIM provides constructs for representing simulated control flow constructs so they can be transformed into a canonical form. For example, imperative iteration can be simulated with an if-then statement and a goto statement. This simulated iteration is converted into a GIM iteration construct, which simplifies the representation of the imperative code. From these three advantages, it is clear that any research based on the GIM is simplified because the number of imperative programming language constructs to consider has been greatly reduced. Finally, the GIM provides a formal definition of each of the imperative programming language constructs based on the weakest precondition notation.

The GOM is an important contribution because it provides a canonical form for object-oriented programming languages. Much like the GIM, the GOM is programming language independent, program construct independent, and recognizes simulated control flow constructs. The GOM has been built primarily as a tool for forward engineering, so it provides a model from which an object-oriented program can be produced. By building transformations from the GOM to a specific object-oriented language, the programming constructs that best implement the canonical forms from the GOM can be used to generate the object-oriented program. As in the GIM, any research based on the GOM is simplified because the number of object-oriented constructs to consider is reduced. The GOM also provides formal definitions for object-oriented constructs including classes, objects, methods, and messages. Defining the formal semantics of these constructs using the weakest precondition notation is original research in this field.

The PBOI methodology is the most important contribution of this research because it provides a new way to extract objects from legacy imperative code. There have been other object identification methods presented in the literature (see Chapter I), but the PBOI methodology results in an object-oriented design that is a *consistent* implementation of the legacy imperative system. Other researchers have developed systems to extract functionally equivalent objects, but these researchers do not provide a proof of this claim. The object-oriented design extracted using the PBOI methodology is valuable because it produces the same output as the legacy system given the same input. The PBOI methodology is similar to the GBOI, TBOI, and RBOI methodologies because it extracts objects from legacy code. The PBOI methodology is significantly better because it extracts functionally equivalent objects, it has been formalized using mathematical transformations, a proof-of-concept prototype implementation has been developed, and it has been demonstrated on a (real world) 3000-line FORTRAN legacy system.

Using formal transformations to define the PBOI methodology is a significant contribution because it casts the methodology in a consistent, unambiguous, and provably correct form. The transformations demonstrate how an entire methodology for converting imperative subprograms can be expressed using formal methods. The transformations are based on the formal definitions of imperative subprograms, subprogram calls, and programming statements, which are minor research contributes as discussed in Section 9.3. Other researchers can use this work as an exemplar of how to express the transformation of imperative subprograms using the power of formal methods. These transformations were fully automated using the Refine^{TM} programming language, as described in Chapter VIII.

The proof that the PBOI methodology extracts an object-oriented design that is a *consistent* implementation of the legacy imperative code is a significant contribution in the area of re-engineering because it is the first such proof. Without defining *any* of the specific input or output values from the legacy imperative code, it has been proven that the extracted object-oriented design produces the same output as the legacy imperative code given the same input. This means it is not necessary to implement the design to prove it is consistent with the legacy code. The proofs of the individual theorems also provide exemplars of how to prove the consistency of formal transformations using weakest precondition arguments.

These major contributions combine to provide a significant advance in the area of re-engineering, viz. the definition, formalization, proof, and automation of a novel methodology for extracting a functionally equivalent object-oriented design from legacy imperative code. The minor contributions of this research are discussed in the following section.

9.3 Minor Contributions

This research also makes the following minor contributions.

1. A taxonomy of imperative subprograms.
2. The formal definitions of imperative subprograms, object-oriented classes, object-oriented methods, and object-oriented messages.
3. The Generic Imperative Language (GIL).
4. The Generic Object-Oriented Language (GOL).

5. The μ relation that links actual parameters to formal parameters.
6. The proof-of-concept prototype built for the FORTRAN language.

The taxonomy of imperative subprograms provides a meaningful way to classify any imperative subprogram. It focuses the job of re-engineering to six well defined categories. Other research on imperative subprograms can also use the taxonomy in the same way. The attributes that define the taxonomy could be modified to reflect different aspects of imperative subprograms.

The formal definitions of imperative subprograms, subprogram calls, statements, and designs presented in Chapter III are a consistent and unambiguous representation of imperative programming language constructs. They provide a foundation for formal transformations of imperative constructs that can be used to develop new transformations in other research. The formal definitions of object-oriented classes, methods, messages, statements, and designs provide a consistent and unambiguous representation of object-oriented programming language constructs. In this research, these definitions provide the representation of the target of the formal transformations. The definitions of imperative and object-oriented constructs are used extensively in the proof of functional equivalence. These representations simplify the proof because of their formal nature.

The GIL and the GOL provide a surface syntax for the GIM and the GOM, respectively. They have been implemented using a grammar that defines the surface syntax of GIM and GOM constructs. The GIL and the GOL demonstrate the use of a surface syntax as a convenient user interface to Abstract Syntax Trees. The surface syntax defined for the GIM and the GOM provide a user-friendly interface to the entities being transformed. In this way, the GIL and GOM are intended to be used to view an AST that has already been built. Since the GIL and the GOL have been built using a grammar, they can also be used to *build* ASTs using the syntax of the grammar. Implementing and testing this aspect of these languages is beyond the scope of this research, but the idea of defining a *generic* imperative language and a *generic* object-oriented language is appealing. Building the GIL and the GOL is a minor contribution that can be used as a stepping stone to future research.

The μ relation proved to an invaluable aid in the formalization of subprogram signatures. The transitive closure of μ is an effective way to determine which actual parameter is linked to a formal parameter that has been converted to an attribute of a class. It may be possible to apply this relation to other applications that update signatures of subprograms.

Finally, the proof-of-concept prototype demonstrates that legacy systems written in FORTRAN-77 can be converted to the GIM. It also shows that the entire PBOI methodology can be automated. The transformations that define the PBOI were easily implemented using the REFINE^{TM} programming language, which demonstrates the utility of such formal definitions and languages such as REFINE.

These minor contributions aided this research in achieving the overall goal of extracting functionally equivalent object-oriented designs from imperative legacy code. The items identified for future research are discussed in the following section.

9.4 Summary

This chapter has explained why the GIM, the GOM, the PBOI methodology, and the formal transformations are significant contributions to the field of re-engineering. The following chapter discusses future research and overall conclusions.

X. Future Research and Conclusions

10.1 Introduction

This chapter summarizes several ideas for future research that could easily extend this research. Concluding remarks are included at the end of this chapter.

10.2 Future Research

The following sections discuss items that should be done as future research.

10.2.1 Data dependencies. The PBOI methodology may be too aggressive when determining which parameters should be attributes of classes. Specifically, PBOI Case 3 should be scrutinized and possibly extended as follows. It may be the case that a local variable is directly *data dependent* on an input parameter of a Category 3 subprogram. When this local variable is passed as a parameter to another subprogram, it will not remain as an attribute of a class. This is because it is a local variable.

However, it may be possible to extract more semantically meaningful objects by allowing the data item linked to this local variable to remain an attribute of a class. This is based on the local variable's data dependency to the formal parameter. There are three dependency cases to consider.

direct One or multiple assignment statements link the local variable *back* to the formal parameter.

indirect The local variable is *produced* from a subprogram call that includes the formal parameter as an input parameter.

hybrid A combination of both direct and indirect data dependencies.

An issue remains as to which data item should be built as the attribute of the class. Is the formal parameter of the called subprogram the attribute or a different view of the attribute? Is the formal parameter of the calling subprogram the attribute? Where should the code that changes the formal parameter into the local variable be placed as a method?

10.2.2 Arrays. Arrays present an interesting challenge. There are several things that have been left as future research. Specifically, a common method for storing data in the imperative paradigm is to build ***parallel arrays*** of data. It has been proposed that the attributes of a candidate object can be derived from the arrays because the values of the attributes of a single instance are stored at the same index in the arrays. Future research should convert these parallel arrays of attributes into a single array of instances with the appropriate attributes.

Another popular method for storing homogeneous attributes of candidate objects is to use a single array to represent all the attributes. A second array or another dimension of the array is used to store the instances of the objects. These arrays of attributes could be converted into parallel arrays of attributes and then converted into a single array of objects with attributes.

10.2.3 Modifying the Design. The process of merging overlapping classes based on the overlap of a single data item may be too aggressive. As described in the Feasibility Demonstration (see Chapter VIII), the design extracted for the BMDSIM legacy system included classes with large numbers of attributes and methods. Future research should explore different heuristics for the updates done to the design when the main program is being transformed.

One possibility is to merge two classes only when the attributes of one class are a subset of the attributes of the other class. This is a more restrictive heuristic for merging that may avoid building classes with such large numbers of attributes and methods. This may introduce duplication of system state variables, however, so a method for combining these classes without duplicating the imperative state variables must be developed. One possibility is to change attributes not in the intersection into parameters in the methods of the effected classes.

Another issue is the lack of analysis that is done on the subprograms that are called by the main program. The formal parameters of these subprograms are built as attributes of classes, but there is no *filtering* of these data items to see if they should remain as attributes.

It may be the case that changing one or more of these attributes into a parameter helps to avoid the problem of building classes with large numbers of attributes and methods.

During the conversion of the BMDSIM legacy system, several classes in the extracted design included no attributes. Future research could examine these classes and possibly merge them into one *utility* class. There appears to be a difference between a class where the attributes have been converted solely into attributes of other classes and a class where the attributes have been converted solely into parameters.

Currently, object instances are built only when the main program is being converted. It is possible to recognize that some collection of local variables in a subprogram actually represent an object and the object can be instantiated at the correct point in the subprogram. This might improve the prospects of identifying more "meaningful" objects from the GIM.

Finally, in order to add semantics to the extracted objects, some interaction with the user may be required. An extension to the prototype could include this interaction to determine if extracted objects should be combined to more closely resemble domain entities. If domain entities are expressed formally, it may be possible for the computer to compare the domain entities to the extracted objects.

10.2.4 Program Slices. The program slicing done to eliminate Category 4 and Category 5 subprograms produces inefficient slices. Future research could explore ways to optimize these slices and produce less conservative slices. Similarly, some of the program slices that are built during this process are repeated in whole as part of another program slice. Such slices could be built as object methods and called by the encompassing program slice. Finally, the same imperative output statement may appear in multiple slices which may cause spurious output when the design is finally implemented and executed. A more detailed analysis of the variables being output could be done as future research in order to ensure the same output statement does *not* appear in multiple slices.

10.2.5 The Generic Unstructured Model. It may be possible to add a front-end to the reverse engineering methodology that automatically re-structures imperative code.

Unstructured legacy code could be transformed into a canonical form, (termed the Generic Unstructured Model (GUM)) that models the unstructured use of `goto` statements. The transformation of GUM entities to GIM entities would provide a canonical means for restructuring legacy imperative systems.

10.2.6 Extensions. Currently, the prototype transforms only FORTRAN-77 to the GIM. Future research should explore the conversion of other imperative languages to the GIM. Reasoning Systems^{TM} currently provides language tools only for FORTRAN, C, Cobol, and Ada, which restricts the extension of the prototype to these languages until parsers for other languages can be built. Some prototype transformations have been done for Ada 83. Specifically, four different forms of iteration in Ada have been canonicalized into one representation in the GIM. To extend the GIM further, other imperative entities such as heterogeneous data types and pointers could be added to the GIM. If larger systems with millions of lines of imperative code are to be transformed, the efficiency of the prototype must be improved.

10.3 Conclusions

It is clear that this research has been completed successfully. The objectives defined in the Problem Statement (see Chapter II) have been met. The GIM provides the desired canonical form for re-engineering legacy imperative code, as presented in Chapter III. The GOM provides the desired canonical form for the object-oriented target system, as presented in Chapter IV. Several in-depth examples of how the PBOI methodology extracts objects from GIM subprograms were presented in Chapter V. The PBOI formal transformations presented in Chapter VI define the PBOI methodology in an unambiguous, consistent, and provably correct manner. Using the PBOI methodology is desirable because the extracted object-oriented design is functionally equivalent to the legacy imperative code. The proof of this claim was presented in Chapter VII. It is feasible to automate this methodology as demonstrated in Chapter VIII. It should be clear to the reader that this research makes significant contributions to the field of re-engineering.

Appendix A. The Generic Imperative Language

A.1 Introduction

This appendix includes the surface syntax developed for the Generic Imperative Language (GIL[1]). The surface syntax provides an easy way to view the GIM representation of an imperative program. The GIL is not intended to be used as a forward-engineering programming language.

The notation used to present the surface syntax is taken from the Software Refinery^{TM} Dialect tool [53]. Each production for producing surface syntax is presented using the following format:

```
<GIM object> ::= [ <surface syntax constructs> ]
```

The left-hand-side of this production gives the GIM object for which this surface syntax is defined, and the right-hand-side shows the sequence of constructs used to define the surface syntax. This sequence can include attributes of the object or literal syntax tokens. Optional constructs are enclosed in braces, i.e. `{ }`. Sequences of constructs are indicated by listing an attribute of the object followed by a star token (*) or plus token (+) followed by the delimiter token. The star token indicates zero or more members of the sequence and plus token indicates one or more members. For example, the surface syntax for GIM `imperative-name` objects is shown below.

```
imperative-name ::= [imp-identifier !! imp-has-indices
     "(" imp-indices + "," ")"]
  builds imperative-name,
```

This production means the surface syntax for an `imperative-name` is the value of the `imp-identifier` attribute followed by any indices in the `imp-indices` attribute separated by commas. The `imp-has-indices` attribute is a boolean attribute used to determine if the `imperative-name` is an array being accessed.

A.2 The Generic Imperative Language

The productions that define the GIL grammar are presented below.

[1]Pronounced "Jill", as in Jack and Jill.

```
!! in-package("RU")
!! in-grammar('syntax)

grammar Generic-Imperative-Language

  file-classes imperative-subprogram

  productions

%-------------------------------------------------------------------------------
% imperative design
%-------------------------------------------------------------------------------

    imperative-design ::= [imperative-programs + ""]
      builds imperative-design,

%-------------------------------------------------------------------------------
% data types
%-------------------------------------------------------------------------------

    imperative-integer ::= ["integer"]
      builds imperative-integer,

    imperative-real ::= ["real"]
      builds imperative-real,

    imperative-boolean ::= ["boolean"]
      builds imperative-boolean,

    imperative-character ::= ["character"]
      builds imperative-character,

    imperative-string ::= ["string"]
      builds imperative-string,

    imperative-array ::= ["array" "(" imp-array-dimensions * "," ")"
                          "of" imp-array-element-type ]
      builds imperative-array,

    imperative-index-type ::= [imp-index-lower-bound ".." imp-index-upper-bound]
      builds imperative-index-type,

%-------------------------------------------------------------------------------
% variables and names
%-------------------------------------------------------------------------------

    imperative-name ::= [imp-identifier ~!! imp-has-indices]
      builds imperative-name,

    imperative-name ::= [imp-identifier !! imp-has-indices
         "(" imp-indices + "," ")"]
      builds imperative-name,

%-------------------------------------------------------------------------------
% binary expressions
%-------------------------------------------------------------------------------
```

```
    imperative-addition ::= [imp-bin-exp-seq ++ "+"]
      builds imperative-addition,

    imperative-and ::= [imp-bin-exp-seq ++ "and"]
      builds imperative-and,

    imperative-concat ::= [imp-bin-exp-seq ++ "&"]
      builds imperative-concat,

    imperative-division ::= [imp-bin-exp-seq ++ "/"]
      builds imperative-division,

    imperative-equal ::= [imp-bin-exp-operand-1 "=" imp-bin-exp-operand-2]
      builds imperative-equal,

    imperative-exponent ::= [imp-bin-exp-operand-1 "^" imp-bin-exp-operand-2]
      builds imperative-exponent,

    imperative-greater-than-or-equal ::= [imp-bin-exp-operand-1 ">=" imp-bin-exp-operand-2]
      builds imperative-greater-than-or-equal,

    imperative-greater-than ::= [imp-bin-exp-operand-1 ">" imp-bin-exp-operand-2]
      builds imperative-greater-than,

    imperative-less-than-or-equal ::= [imp-bin-exp-operand-1 "<=" imp-bin-exp-operand-2]
      builds imperative-less-than-or-equal,

    imperative-less-than ::= [imp-bin-exp-operand-1 "<" imp-bin-exp-operand-2]
      builds imperative-less-than,

    imperative-multiplication ::= [imp-bin-exp-seq ++ "*"]
      builds imperative-multiplication,

    imperative-not-equal ::= [imp-bin-exp-operand-1 "/=" imp-bin-exp-operand-2]
      builds imperative-not-equal,

    imperative-or ::= [imp-bin-exp-seq ++ "or"]
      builds imperative-or,

    imperative-subtraction ::= [imp-bin-exp-seq ++ "-"]
      builds imperative-subtraction,

%-----------------------------------------------------------------------------
% unary expressions
%-----------------------------------------------------------------------------

    imperative-negate ::= ["-" imp-unary-operand]
      builds imperative-negate,

    imperative-not ::= ["not" imp-unary-operand]
      builds imperative-not,

    imperative-null ::= [" "]
      builds imperative-null,
%-----------------------------------------------------------------------------
% literals
%-----------------------------------------------------------------------------

    imperative-literal-true ::= "true"
      builds imperative-literal-true,

    imperative-literal-false ::= "false"
      builds imperative-literal-false,

    imperative-literal-integer ::= imperative-literal-value
      builds imperative-literal-integer,

    imperative-literal-real ::= imperative-literal-value
      builds imperative-literal-real,

    imperative-literal-charstring ::= imperative-literal-value
      builds imperative-literal-charstring,

    imperative-literal-newline ::= "~%"
      builds imperative-literal-newline,

%-----------------------------------------------------------------------------
% assignment
%-----------------------------------------------------------------------------

    imperative-assignment ::= [imp-assign-lhs ":=" imp-assign-rhs]
      builds imperative-assignment,

%-----------------------------------------------------------------------------
% selection
%-----------------------------------------------------------------------------

    imperative-selection ::= ["if" imperative-exp "then"
                                 imperative-then-part + ";"
                              "else"
                                 imperative-else-part + ";"
                              "endif"]
      builds imperative-selection,

%-----------------------------------------------------------------------------
% iteration
%-----------------------------------------------------------------------------

    imperative-iteration ::= ["while" iter-exp "do"
                                ["begin"
                                   iter-body + ";"
                                 "end"]
                             ]
      builds imperative-iteration,

%-----------------------------------------------------------------------------
% imperative subprograms
%-----------------------------------------------------------------------------

    imperative-procedure ::= ["procedure" imp-proc-identifier
```

```
                                       "(" imp-proc-formals * "," ")"
                                     ["begin"
                                        imp-proc-statements + ";"
                                      "end"]
                                  ]
      builds imperative-procedure,

    imp-procedure-call ::= [ imp-proc-call-identifier
                                       "(" imp-proc-call-actuals * "," ")" ]
      builds imp-procedure-call,

    imperative-function ::= [imp-function-return-type "function" imp-func-identifier
                                       "(" imp-func-formals * "," ")"
                                     ["begin"
                                        imp-func-statements + ";"
                                      "end"]
                                  ]
      builds imperative-function,

    imperative-function-call ::= [ imp-fun-call-identifier
                                       "(" imp-fun-call-actuals * "," ")" ]
      builds imperative-function-call,


%--------------------------------------------------------------------------------
% output statements
%--------------------------------------------------------------------------------

    imperative-input ::= ["read" "(" imp-in-logical-file ","
                                     imp-input-list + "," ")"]
      builds imperative-input,

    imperative-output ::= ["write" "(" imp-out-logical-file ","
                                       imp-output-list + "," ")"]
      builds imperative-output,

    imperative-format-item ::= [imp-fmt-item]
      builds imperative-format-item,

    imperative-format-item ::= [imp-fmt-item imp-fmt-format]
      builds imperative-format-item,

    imperative-format-integer ::= ["~i" imp-format-width ]
      builds imperative-format-integer,

    imperative-format-integer ::= ["~I" imp-format-width ]
      print-only,

    imperative-format-decimal ::= ["~d" imp-format-width "." imp-decimal-part-width]
      builds imperative-format-decimal,

    imperative-format-decimal ::= ["~D" imp-format-width "." imp-decimal-part-width]
      print-only,

    imperative-format-scientific ::= ["~e" imp-format-width ]
      builds imperative-format-scientific,
    imperative-format-scientific ::= ["~E" imp-format-width ]
      print-only,

    imperative-format-string ::= ["~s" imp-format-width ]
      builds imperative-format-string,

    imperative-format-string ::= ["~S" imp-format-width ]
      print-only

  start-classes imperative-ast, imperative-subprogram

  no-patterns

  precedence

      for imperative-expression brackets "(" matching ")"

         (same-level "or" associativity left),
         (same-level "and" associativity left),
         (same-level "not" associativity right),

         (same-level "<", "<=", "=", ">=", ">", "/=" associativity none),

         (same-level "+", "-" associativity left),
         (same-level "*", "/" associativity left),
         (same-level "^" associativity right)

end
```

Appendix B. The Generic Object-Oriented Language

B.1 Introduction

This appendix includes the surface syntax defined for the Generic Object-Oriented Language (GOL[1]). This surface syntax is not intended for use as a forward-engineering programming language, but as a user-friendly interface to the Generic Object-Oriented Model (GOM) ASTs. Instead of printing each GOM AST and comparing the values of the attributes of the AST, the GOL provides a short-hand view that is used to quickly and easily view the attributes of a GOM AST.

The notation used to present the surface syntax is taken from the Software Refinery^{TM} Dialect tool [53]. Each production for producing surface syntax is presented using the following format:

```
<GOM object> ::= [ <surface syntax constructs> ]
```

The left-hand-side of this production gives the GOM object for which this surface syntax is defined, and the right-hand-side shows the sequence of constructs used to define the surface syntax. This sequence can include attributes of the object or literal syntax tokens. Optional constructs are enclosed in braces, i.e. `{ }`. Sequences of constructs are indicated by listing an attribute of the object followed by a star token (*) or plus token (+) followed by the delimiter token. The star token indicates zero or more members of the sequence and plus token indicates one or more members. For example, the surface syntax for GOM `GOM-Variable` objects is shown below.

```
GOM-Variable ::= [gom-name !! gom-has-indices
      "(" gom-indices + "," ")" ]
  builds GOM-Variable,
```

This production means the surface syntax for an `GOM-Variable` is the value of the `gom-name` attribute followed by any indices in the `gom-indices` attribute separated by commas. The `gom-has-indices` attribute indicates whether or not this variable access includes indices into an array.

[1]Pronounced "*gôl*" as in "golf".

B.2 The Generic Object-Oriented Language

The productions that define the GOL grammar are presented below.

```
!! in-package("RU")
!! in-grammar('syntax)

grammar Generic-OO-Language

  file-classes GOM-Design

  productions

%-----------------------------------------------------------------------------
% overall design class
%-----------------------------------------------------------------------------

    GOM-Design ::= [GOM-classes + ""]
      builds GOM-design,

%-----------------------------------------------------------------------------
% OO objects
%-----------------------------------------------------------------------------

    GOM-Class ::= ["class" gom-name
                  ["attributes" gom-attrs * ","]
                  [gom-opers * ""]
                  ["superclass" gom-super]]
      builds GOM-Class,

    GOM-Instantiate ::= ["new" "(" gom-inst-class ")"]
      builds GOM-Instantiate,

    GOM-Method ::= ["method" gom-name "(" [gom-params * ","] ")"
                      ~!! gom-rtns-val
                      "begin"
                         [gom-stmts * ";"]
                      "end"]
      builds GOM-Method,

    GOM-Method ::= ["method" gom-name "(" [gom-params * ","] ")"
                      !! gom-rtns-val ":" gom-return-type
                      "begin"
                         [gom-stmts * ";"]
                      "end"]
      builds GOM-Method,

    GOM-Message ::= [ gom-call "(" gom-actuals * "," ")" ]
      builds GOM-Message,

%-----------------------------------------------------------------------------
% variables, attributes, and parameters
%-----------------------------------------------------------------------------
```

```
    GOM-Variable ::= [gom-name ~!! gom-has-indices]
      builds GOM-Variable,

    GOM-Variable ::= [gom-name !! gom-has-indices
          "(" gom-indices + "," ")" ]
      builds GOM-Variable,

    GOM-Attribute ::= [gom-name ~!! gom-has-indices]
      builds GOM-Attribute,

    GOM-Attribute ::= [gom-name !! gom-has-indices
          "(" gom-indices + "," ")" ]
      builds GOM-Attribute,

    GOM-Attr-Access ::= [ gom-tar-object "." gom-attrib ]
      builds GOM-Attr-Access,

    GOM-Parameter ::= [gom-name ~!! gom-has-indices]
      builds GOM-Parameter,

    GOM-Parameter ::= [gom-name !! gom-has-indices
          "(" gom-indices + "," ")" ]
      builds GOM-Parameter,

%--------------------------------------------------------------------------------
% data types
%--------------------------------------------------------------------------------

    GOM-integer ::= ["integer"]
      builds GOM-integer,

    GOM-real ::= ["real"]
      builds GOM-real,

    GOM-boolean ::= ["boolean"]
      builds GOM-boolean,

    GOM-character ::= ["character"]
      builds GOM-character,

    GOM-string ::= ["string"]
      builds GOM-string,

% arrays

    GOM-array ::= ["array" "(" gom-array-dimensions * "," ")"
                        "of" gom-array-element-type ]
      builds GOM-array,

    GOM-index-type ::= [gom-index-lower-bound ".." gom-index-upper-bound]
      builds GOM-index-type,

% instances

    GOM-instance ::= [ "a" gom-instance-of ]
      builds GOM-instance,
%--------------------------------------------------------------------------------
% binary expressions
%--------------------------------------------------------------------------------

    GOM-addition ::= [GOM-bin-exp-seq ++ "+"]
      builds GOM-addition,

    GOM-and ::= [GOM-bin-exp-seq ++ "and"]
      builds GOM-and,

    GOM-concat ::= [GOM-bin-exp-seq ++ "&"]
      builds GOM-concat,

    GOM-division ::= [GOM-bin-exp-seq ++ "/"]
      builds GOM-division,

    GOM-equal ::= [GOM-bin-exp-operand-1 "=" GOM-bin-exp-operand-2]
      builds GOM-equal,

    GOM-exponent ::= [GOM-bin-exp-operand-1 "^" GOM-bin-exp-operand-2]
      builds GOM-exponent,

    GOM-greater-than-or-equal ::= [GOM-bin-exp-operand-1 ">="
                                   GOM-bin-exp-operand-2]
      builds GOM-greater-than-or-equal,

    GOM-greater-than ::= [GOM-bin-exp-operand-1 ">" GOM-bin-exp-operand-2]
      builds GOM-greater-than,

    GOM-less-than-or-equal ::= [GOM-bin-exp-operand-1 "<="
                                GOM-bin-exp-operand-2]
      builds GOM-less-than-or-equal,

    GOM-less-than ::= [GOM-bin-exp-operand-1 "<" GOM-bin-exp-operand-2]
      builds GOM-less-than,

    GOM-multiplication ::= [GOM-bin-exp-seq ++ "*"]
      builds GOM-multiplication,

    GOM-not-equal ::= [GOM-bin-exp-operand-1 "/=" GOM-bin-exp-operand-2]
      builds GOM-not-equal,

    GOM-or ::= [GOM-bin-exp-seq ++ "or"]
      builds GOM-or,

    GOM-subtraction ::= [GOM-bin-exp-seq ++ "-"]
      builds GOM-subtraction,

%--------------------------------------------------------------------------------
% unary expressions
%--------------------------------------------------------------------------------

    GOM-negate ::= ["-" GOM-unary-operand]
      builds GOM-negate,
    GOM-not ::= ["not" GOM-unary-operand]
      builds GOM-not,

    GOM-null ::= [" "]
      builds GOM-null,

%-----------------------------------------------------------------------------
% literals
%-----------------------------------------------------------------------------

    GOM-literal-true ::= "true"
      builds GOM-literal-true,

    GOM-literal-false ::= "false"
      builds GOM-literal-false,

    GOM-literal-integer ::= GOM-literal-value
      builds GOM-literal-integer,

    GOM-literal-real ::= GOM-literal-value
      builds GOM-literal-real,

    GOM-literal-charstring ::= GOM-literal-value
      builds GOM-literal-charstring,

    GOM-literal-newline ::= "~%"
      builds GOM-literal-newline,

%-----------------------------------------------------------------------------
% assignment
%-----------------------------------------------------------------------------

    GOM-assignment ::= [gom-assign-lhs ":=" gom-assign-rhs]
      builds GOM-assignment,

%-----------------------------------------------------------------------------
% selection
%-----------------------------------------------------------------------------

    GOM-selection ::= ["if" gom-exp "then"
                               gom-then-part + ";"
                            "else"
                               gom-else-part + ";"
                            "endif"]
      builds GOM-selection,

%-----------------------------------------------------------------------------
% iteration
%-----------------------------------------------------------------------------

    GOM-iteration ::= ["while" gom-iter-exp "do"
                              ["begin"
                                 gom-iter-body + ";"
                               "end"]
                            ]
      builds GOM-iteration,
%------------------------------------------------------------------------------
% non-user-defined subprogram calls
%------------------------------------------------------------------------------

    GOM-function-call ::= [ gom-fun-call-identifier
                                "(" gom-fun-call-actuals * "," ")" ]
      builds GOM-function-call,

    GOM-procedure-call ::= [ gom-proc-call-identifier
                                "(" gom-proc-call-actuals * "," ")" ]
      builds GOM-procedure-call,

%------------------------------------------------------------------------------
% output statements
%------------------------------------------------------------------------------

    GOM-input ::= ["read" "(" gom-in-logical-file ","
                                gom-input-list + "," ")"]
      builds GOM-input,

    GOM-output ::= ["write" "(" gom-out-logical-file ","
                                gom-output-list + "," ")"]
      builds GOM-output,

    GOM-format-item ::= [gom-fmt-item]
      builds GOM-format-item,

    GOM-format-item ::= [gom-fmt-item gom-fmt-format]
      builds GOM-format-item,

    GOM-format-integer ::= ["~i" gom-format-width ]
      builds GOM-format-integer,

    GOM-format-integer ::= ["~I" gom-format-width ]
      print-only,

    GOM-format-decimal ::= ["~d" gom-format-width "." gom-decimal-part-width]
      builds GOM-format-decimal,

    GOM-format-decimal ::= ["~D" gom-format-width "." gom-decimal-part-width]
      print-only,

    GOM-format-scientific ::= ["~e" gom-format-width ]
      builds GOM-format-scientific,

    GOM-format-scientific ::= ["~E" gom-format-width ]
      print-only,

    GOM-format-string ::= ["~s" gom-format-width ]
      builds GOM-format-string,

    GOM-format-string ::= ["~S" gom-format-width ]
      print-only

  start-classes GOM-Design, GOM-Class
  no-patterns

  precedence

      for GOM-expression brackets "(" matching ")"

         (same-level "or" associativity left),
         (same-level "and" associativity left),
         (same-level "not" associativity right),
         (same-level "&" associativity right),
         (same-level "<", "<=", "=", ">=", ">", "/=" associativity none),

         (same-level "+", "-" associativity left),
         (same-level "*", "/" associativity left),
         (same-level "^" associativity left)

end
```

Prove:

$$\mathrm{wp}(\text{if } B_1 \text{ then } S_1 \text{ elsif } B_2 \text{ then } S_2 \text{ else } S_3, \mathrm{R}) \Leftrightarrow$$
$$\mathrm{wp}(\text{if } B_1 \text{ then } S_1 \text{ else } (\text{if } B_2 \text{ then } S_2 \text{ else } S_3), \mathrm{R})$$

Proof:

$$\mathrm{wp}(\text{if } B_1 \text{ then } S_1 \text{ elsif } B_2 \text{ then } S_2 \text{ else } S_3, \mathrm{R}) \Leftrightarrow$$

$$\begin{aligned}&(B_1 \implies \mathrm{wp}(S_1, \mathrm{R})) \wedge\\ &\quad((\neg B_1 \wedge B_2) \implies \mathrm{wp}(S_2, \mathrm{R})) \wedge\\ &\qquad((\neg B_1 \wedge \neg B_2) \implies \mathrm{wp}(S_3, \mathrm{R})) \Leftrightarrow\end{aligned}$$

$$\begin{aligned}&(B_1 \implies \mathrm{wp}(S_1, \mathrm{R})) \wedge\\ &\quad(\neg (B_1 \vee \neg B_2) \implies \mathrm{wp}(S_2, \mathrm{R})) \wedge\\ &\qquad(\neg (B_1 \vee B_2) \implies \mathrm{wp}(S_3, \mathrm{R})) \Leftrightarrow\end{aligned}$$

$$\begin{aligned}&(B_1 \implies \mathrm{wp}(S_1, \mathrm{R})) \wedge\\ &\quad((B_1 \vee \neg B_2) \vee \mathrm{wp}(S_2, \mathrm{R})) \wedge\\ &\qquad((B_1 \vee B_2) \vee \mathrm{wp}(S_3, \mathrm{R})) \Leftrightarrow\end{aligned}$$

$$\begin{aligned}&(B_1 \implies \mathrm{wp}(S_1, \mathrm{R})) \wedge\\ &\quad(B_1 \vee (\neg B_2 \vee \mathrm{wp}(S_2, \mathrm{R}))) \wedge\\ &\qquad(B_1 \vee (B_2 \vee \mathrm{wp}(S_3, \mathrm{R}))) \Leftrightarrow\end{aligned}$$

$$\begin{aligned}&(B_1 \implies \mathrm{wp}(S_1, \mathrm{R})) \wedge\\ &\quad(B_1 \vee ((\neg B_2 \vee \mathrm{wp}(S_2, \mathrm{R})) \wedge (B_2 \vee \mathrm{wp}(S_3, \mathrm{R})))) \Leftrightarrow\end{aligned}$$

$$\begin{aligned}&(B_1 \implies \mathrm{wp}(S_1, \mathrm{R})) \wedge\\ &\quad(B_1 \vee ((B_2 \implies \mathrm{wp}(S_2, \mathrm{R})) \wedge (\neg B_2 \implies \mathrm{wp}(S_3, \mathrm{R})))) \Leftrightarrow\end{aligned}$$

$$\begin{aligned}&(B_1 \implies \mathrm{wp}(S_1, \mathrm{R})) \wedge\\ &\quad(\neg B_1 \implies ((B_2 \implies \mathrm{wp}(S_2, \mathrm{R})) \wedge (\neg B_2 \implies \mathrm{wp}(S_3, \mathrm{R})))) \Leftrightarrow\end{aligned}$$

$$\begin{aligned}&(B_1 \implies \mathrm{wp}(S_1, \mathrm{R})) \wedge\\ &\quad(\neg B_1 \implies \mathrm{wp}(\text{if } B_2 \text{ then } S_2 \text{ else } S_3, \mathrm{R})) \Leftrightarrow\end{aligned}$$

$$\mathrm{wp}(\text{if } B_1 \text{ then } S_1 \text{ else } (\text{if } B_2 \text{ then } S_2 \text{ else } S_3), \mathrm{R}) \Leftrightarrow$$

Q.E.D.

Appendix D. Parameter Transformation Proof

D.1 Introduction

This appendix explains how to convert a procedure that has at least one parameter that is both an input parameter and an output parameter into a procedure with parameters that are either input only or output only.

D.2 New Procedure Definition

Because of the importance of the distinction between input and output parameters when converting the GIM to the GOM, parameters that are both input and output parameters require special processing. A new vector of output parameters is used to represent the "output" aspect of the input/output parameters. The original parameters are assumed to be input only and the new output parameters hold the new values returned from a procedure call. These new values are stored in the original data items by using an assignment statement after the call to the procedure.

Let $\bar{y}_{out}$ represent the vector of output parameters needed to represent the "output" aspect of the input/output parameters in $\bar{y}$. Let $P^{\bar{y}}_{\bar{y}_{out}}$ represent the precondition P with all occurrences of the parameters in $\bar{y}$ replaced by the parameters in $\bar{y}_{out}$. Let $body^{\bar{y}}_{\bar{y}_{out}}$ represent the body of procedure p, with all occurrences of the parameters in $\bar{y}$ replaced by the parameters in $\bar{y}_{out}$. Using these definitions, a new procedure p' is built that accepts the new output parameter $\bar{y}_{out}$.

```
procedure $p'(\bar{x}, \bar{y}, \bar{y}_{out}, \bar{z})$
    $\{P\}$
    $\bar{y}_{out} := \bar{y};$
    $\{P^{\bar{y}}_{\bar{y}_{out}}\}$
    $body^{\bar{y}}_{\bar{y}_{out}}$
    $\{R\}$
```

The body of p' includes as its first statement an assignment statement to assign all the output parameters in $\bar{y}_{out}$ the initial values of $\bar{y}$. All references to $\bar{y}$ in the original body of procedure p (and the precondition P) have been replaced by references to $\bar{y}_{out}$ to create the rest of the body of the procedure p'.

If the precondition and the postcondition for p' are the same as the precondition and the postcondition for p, then the semantics of p' are the same as that of p, i.e. the two procedure definitions are equivalent. The proof of this is shown below.

Proof.

1. The postcondition R is given as the postcondition for both p and p'.
2. All references to $\bar{y}$ are replaced by references to $\bar{y}_{out}$ in both *body* and P for p', thus $P^{\bar{y}}_{\bar{y}_{out}}$ is the precondition for $body^{\bar{y}}_{\bar{y}_{out}}$.
3. $wp(\bar{y}_{out} := \bar{y},\ P^{\bar{y}}_{\bar{y}_{out}}) \ = \ (P^{\bar{y}}_{\bar{y}_{out}})^{\bar{y}_{out}}_{\bar{y}} \ = \ P$, thus the precondition for p' is P.
4. Since the precondition and postcondition are the same for p and p', the semantics for these two procedure definitions are the same.

$\square$

D.3 New Procedure Call

Let $\bar{b}_{out}$ represent a vector of output parameters that represent the "output" aspect of the input/output parameters in $\bar{b}$. The invocation of the new procedure p' has the form

$$\begin{aligned} &p'(\bar{a}, \bar{b}, \bar{b}_{out}, \bar{c}); \\ &\bar{b} := \bar{b}_{out} \end{aligned}$$

The original $\bar{b}$ parameters are used as input only parameters. The new $\bar{b}_{out}$ vector holds the output parameters and these new values are stored back into the $\bar{b}$ parameters after the call to p' using an assignment statement. Each call to p' is now followed by such an assignment statement.

Let $body'$ represent the body of statements from procedure p'. The execution of a call to procedure p' is equivalent to

$$\begin{array}{rcl}\bar{x} & := & \bar{a};\\ \bar{y} & := & \bar{b};\\ & body'; & \\ \bar{b}_{out} & := & \bar{y}_{out};\\ \bar{c} & := & \bar{z};\end{array}$$

This sequence clearly shows how the new output parameters in $\bar{b}_{out}$ are set to the values in the parameters $\bar{y}_{out}$ returned from the procedure p'.

Let β' represent the execution of p'. The semantics for this procedure call are defined as

$$wp(p'(\bar{a}, \bar{b}, \bar{b}_{out}, \bar{c}), R) \;=\; wp(\beta', R)$$

The assignment statement following the procedure call is required in order to maintain the semantics of the original procedure call. Thus, it is more proper to define the semantics for the new form of procedure call as

$$wp(p'(\bar{a}, \bar{b}, \bar{b}_{out}, \bar{c}); \bar{b} := \bar{b}_{out}, R) \;=\; wp(\beta', wp(\bar{b} := \bar{b}_{out}, R))$$

In order to prove that the addition of the new vectors $\bar{y}_{out}$ and $\bar{b}_{out}$ is correct, it must be proven that the new form of the procedure call maintains the semantics of the original procedure call. Specifically, it must be proven that the following is true.

$$wp(p(\bar{a}, \bar{b}, \bar{c}), R) \;\Leftrightarrow\; wp(p'(\bar{a}, \bar{b}, \bar{b}_{out}, \bar{c});\; \bar{b} := \bar{b}_{out}, R)$$

The proof is shown below.

Proof.

$$
\begin{aligned}
&wp(p(\bar{a},\bar{b},\bar{c}),R)\\
&\quad\Leftrightarrow\ wp(\bar{x} := \bar{a};\ \bar{y} := \bar{b};\ body;\ \bar{b} := \bar{y};\ \bar{c} := \bar{z},\ R)\\
&\quad\Leftrightarrow\ wp(wp(\bar{x} := \bar{a},\ wp(\bar{y} := \bar{b},\ wp(body,\ wp(\bar{b} := \bar{y},\ wp(\bar{c} := \bar{z},\ R))))))\\
&\quad\Leftrightarrow\ wp(\bar{x} := \bar{a};\ \bar{y} := \bar{b};\ \bar{y}_{out} := \bar{y};\ body^{\bar{y}}_{\bar{y}_{out}};\ \bar{b}_{out} := \bar{y}_{out};\ \bar{b} := \bar{b}_{out};\ \bar{c} := \bar{z},\ R)\\
&\quad\Leftrightarrow\ wp(\bar{x} := \bar{a};\ \bar{y} := \bar{b};\ body';\ \bar{b}_{out} := \bar{y}_{out};\ \bar{c} := \bar{z},\ R^{\bar{b}}_{\bar{b}_{out}})\\
&\quad\Leftrightarrow\ wp(\bar{x} := \bar{a};\ \bar{y} := \bar{b};\ body';\ \bar{b}_{out} := \bar{y}_{out};\ \bar{c} := \bar{z},\ wp(\bar{b} := \bar{b}_{out},\ R))\\
&\quad\Leftrightarrow\ wp(p'(\bar{a},\bar{b},\bar{b}_{out},\bar{c});\ \bar{b} := \bar{b}_{out},R)
\end{aligned}
$$

□

Appendix E. BMDSIM Conversion Transcript

This appendix shows the complete transcript of the conversion of the 53 BMDSIM subprograms from FORTRAN to the GIM.

```
.> (rfu::test-conversion)
Not re-loading analysis...

Converting "ANG" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 60
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "ANGLE" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 92
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "ASSIGN" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 609
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "BMDSIM1" to the GIM...

Warning: Make sure STOP statement is
last statement in main program

Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 2084
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "BOOSTR" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 338
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "BOSTIT" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 559
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "BOUNCE" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 71
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file
```

```
Converting "CAPTURE" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 82
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "CROSS" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 86
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "CSP" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 529
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "CUV" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 85
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "DASET" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 169
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "DOT" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 48
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "KEP" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 102
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "LASP" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 64
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file
```

```
Converting "LNKCAL" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 703
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "LNKCK" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 178
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "LNKORD" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 316
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "MAT" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 149
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "MAXA" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 62
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "MIRGEO" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 348
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "MIRVIS" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 357
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "MTM2" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 65
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file
```

```
Converting "MTM3" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 75
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "MTPD" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 84
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "MTRT" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 71
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "ORBEL" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 377
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "PKILL" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 63
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "POSVEC" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 85
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "POSVECS" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 85
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "PRDIV" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 58
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file
```

```
Converting "RADIUS" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 84
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "RAND" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 55
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "RELAY" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 459
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "RHO" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 22
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "RRBVIS" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 419
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "RRPVIS" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 201
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "RTAN" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 41
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "SBMIT" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 195
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file
```

```
Converting "SBMLOC" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 77
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "SBMPOS" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 167
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "SELECL" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 325
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "SGN" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 19
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "SUV" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 80
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "TFBT" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 94
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "TPANG" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 162
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "TRAJ" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 217
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file
```

```
Converting "UNIT" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 47
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "UPLREQ" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 67
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "UPTRNS" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 384
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "VADD" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 81
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "VISCK" to the GIM...
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 164
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Converting "VMAG" to the GIM...
No side effects in this function.
Checking transformation completeness...

The transformation was complete.
Total number of tree nodes: 37
Number of untransformed nodes: 0
Percentage of AST transformed: 100.0%
Saving to pob...
Saving AST and Symbol Table in file
Saving surface syntax in file

Overall number of nodes in analysis: 11451.0
Overall number of untransformed nodes: 0.0
Percentage of analysis transformed: 100.0%
```

Appendix F. Mapping GIM Entities to the GOM

Figures 207 through 214 show the mappings between GIM entities and GOM entities. The transformations are shown in an abbreviated format where the GIM entity on the left-hand-side of the transformation is converted into the GOM entity on the right-hand-side. These transformations are straightforward given the description of the GIM in Chapter III and the description of the GOM in Chapter IV.

imperative-assignment $\longrightarrow$	gom-assignment
imperative-selection $\longrightarrow$	gom-selection
imperative-iteration $\longrightarrow$	gom-iteration

Figure 207 Control Flow Constructs

imp-subprogram-call $\longrightarrow$	gom-procedure-call
imperative-function-call $\longrightarrow$	gom-function-call
imp-subprogram-call $\xrightarrow{u}$	gom-message
imperative-function-call $\xrightarrow{u}$	gom-message

Figure 208 Subprogram Calls

imperative-variable $\longrightarrow$	gom-variable
imperative-name $\longrightarrow$	gom-variable

Figure 209 Data Storage Constructs

imperative-integer	⟶	gom-integer
imperative-real	⟶	gom-real
imperative-boolean	⟶	gom-boolean
imperative-character	⟶	gom-character
imperative-string	⟶	gom-string
imperative-array	⟶	gom-array
imperative-index-type	⟶	gom-index-type

Figure 210 Data Type Classes

imperative-literal-boolean	⟶	gom-literal-boolean
imperative-literal-integer	⟶	gom-literal-integer
imperative-literal-real	⟶	gom-literal-real
imperative-literal-charstring	⟶	gom-literal-charstring
imperative-literal-newline	⟶	gom-literal-newline

Figure 211 Literals

imperative-addition ⟶	gom-addition
imperative-and ⟶	gom-and
imperative-concat ⟶	gom-concat
imperative-division ⟶	gom-division
imperative-equal ⟶	gom-equal
imperative-exponent ⟶	gom-exponent
imperative-greater-than-or-equal ⟶	gom-greater-than-or-equal
imperative-greater-than ⟶	gom-greater-than
imperative-less-than-or-equal ⟶	gom-less-than-or-equal
imperative-less-than ⟶	gom-less-than
imperative-multiplication ⟶	gom-multiplication
imperative-not-equal ⟶	gom-not-equal
imperative-or ⟶	gom-or
imperative-subtraction ⟶	gom-subtraction

Figure 212 Binary Expressions

imperative-negate ⟶	gom-negate
imperative-not ⟶	gom-not
imperative-null ⟶	gom-null

Figure 213 Unary Expressions

imperative-file ⟶	gom-file
imperative-input ⟶	gom-input
imperative-output ⟶	gom-output
imperative-format ⟶	gom-format

Figure 214 Input and Output

Bibliography

1. Achee, B.L. and Doris L. Carver. "A Greedy Approach to Object Indentification in Imperative Code." *3rd Workshop on Program Comprehension.* 4–11. November 1994. 7 May 97.

2. Aho, Alfred V., et al. *Compilers: Principles, Techniques, and Tools* (2nd Edition). Reading, MA: Addison Wesley, 1988.

3. Barnes, J. G. P. *Programming in Ada.* Wokingham, England: Addison-Wesley, 1994.

4. Biggerstaff, T. J., et al. "Program Understanding and the Concept Assignment Problem," *Communications of the ACM*, *37*(5):72–82 (May 1994).

5. Biggerstaff, Ted J. "Design Recovery for Maintenance and Reuse," *IEEE Computer*, 36–49 (Jul 1989).

6. Biggerstaff, Ted J., et al. "The Concept Assignment Problem in Program Understanding." *Proceedings of the 15th International Conference on Software Engineering*, edited by Edna Straub. 483–498. Los Alamitos, CA: IEEE Computer Society Press, May 17-21 1993.

7. Blaha, Michael and William Premerlani. "A Catalog of Object Model Transformations." *Proceedings of the Third Working Conference on Reverse Engineering.* 87–96. Nov 1996.

8. Boehm, Barry W. *Software Engineering Economics.* Prentice Hall, 1981.

9. Booch, Grady. *Object-Oriented Analysis and Design* (2nd Edition). Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1994.

10. Booch, Grady and James Rumbaugh. *Unified Method* (0.8 Edition). Rational Software Corporation, 280 San Tomas Expressway, 1995.

11. Budd, Timothy. *Object-Oriented Programming.* Reading, MA: Addison-Wesley, 1991.

12. Byrne, Eric J. "A conceptual foundation for software re-engineering." *Proceedings of the International Conference on Software Maintenance.* 216–235. IEEE Computer Society Press, Nov 1992.

13. Chikofsky, Elliot and James Cross. "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, *7*(1):13–17 (Jan 1990).

14. Choi, Song C. and Walt Scacchi. "Extracting and Restructuring the Design of Large Systems," *IEEE Software*, *7*(1):66–71 (Jan 1990).

15. DeLoach, Scott. *Appendix A: Generic OMT Abstract Syntax Tree.* PhD dissertation, Air Force Institute of Technology, Dayton, OH, Jun 1995.

16. Dershem, Herbert L. and Michael J Jipping. *Programming Languages: Structures and Models.* Boston, MA: PWS Publishing Co, 1993.

17. Detienne, F. and E. Soloway. "An Empirically-Derive Control Struture for the Process of Program Understanding," *International Journal of Man-Machine Studies, 33*(3):323–342 (1990).

18. Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, 1976.

19. Dromey, Geoff. *Program Derivation: The Development of Programs from Specifications.* Sydney, Australia: Addison Wesley, 1989.

20. Ferrante, J., et al. "The Program Dependence Graph and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems, 9*(3):319–349 (Jul 1987).

21. Gall, Harald and René Klösch. "Finding Objects in Procedural Programs: An Alternative Approach." In Wills et al. [78], 208–216.

22. Gallagher, Keith B. and James R. Lyle. "Using Program Slicing in Software Maintenance," *Transactions on Software Engineering, 17*(8):751–761 (Aug 1991).

23. Ghezzi, Carlo and Mehdi Jazayeri. *Programming Language Concepts.* New York: John Wiley and Sons, 1982.

24. Grassman, Winfried Karl and Jean-Paul Tremblay. *Logic and Discrete Mathematics.* Upper Saddle River, New Jersey: Prentice Hall, 1996.

25. Gries, David. *The Science of Programming.* Springer-Verlag, 1981.

26. Harandi, Mehdi T. and Jim Q. Ning. "Knowledge-Based Program Analysis," *IEEE Software, 7*(1):74–81 (Jan 1990).

27. Harris, David R., et al. "Recognizers for Extracting Architectural Features from Source Code." In Wills et al. [78], 252–261.

28. Hartman, John. *Automatic Control Understanding for Natural Programs.* PhD dissertation, University of Texas at Austin, Technical Report AI 91-161, 1991.

29. Hausler, Philip A. and Mark G. Pleszkoch. "Using Function Abstraction to Understand Program Behavior," *IEEE Software, 7*(1):55–63 (Jan 1990).

30. Horwitz, S., et al. "Interprocedural Slicing Using Dependence Graphs." *Proceedings of the ACM SIGPLAN 88, Conference on Programming Language Design and Implementation.* 35–46. Jun 1988.

31. Hutchens, D. H. and V. R. Basili. "System Structure Analysis: Clustering with Data Bindings," *IEEE Transactions on Software Engineering, SE-11*(8):749–757 (Aug 1985).

32. Jacobson, Ivar. *Object-Oriented Software Engineering.* Wokingham, England: Addison-Wesley, 1992.

33. Jacobson, Ivar and Fredrik Lindström. "Re-engineering Old Systems to an Object-Oriented Architecture." *OOPSLA Proceedings.* 340–350. 1991.

34. Johnson, W. Lewis and Ali Erdem. "Interactive Explanation of Software Systems." *Proceedings of the 10th Knowledge-Based Software Engineering Conference.* 1995.

35. Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

36. Kirkpatrick, Wally. *A Method for Improving Technology Research and Development Decisions Regarding BMD and ASAT*. DESE Research and Engineering, Inc., Huntsville, AL, July 1985. 14 July 97.

37. Knuth, Donald E. "Structured Programming with go to Statements," *Computing Surveys*, *6*(4):262–301 (Dec 1974).

38. Korson, Tim and John D. McGregor. "Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, *33*(9):40–60 (Sep 1990).

39. Kozaczynski, W., et al. "Program Concept Recognition and Transformation," *Transactions of Software Engineering*, *18*(12):1065–1075 (Dec 1992).

40. Letovsky, S. and E. Soloway. "Delocalized Plans and Program Comprehension," *IEEE Software*, *3*(3):41–48 (May 1986).

41. Letovsky, Stanley. *Plan analysis of programs*. PhD dissertation, Yale University, Dec 1988.

42. Liu, S. S. and N. Wilde. "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery." *Proceedings of the Conference on Software Maintenance*. 266–271. Nov 1990.

43. Livadas, Panos E. and Theodore Johnson. *A New Approach to Finding Objects in Programs*. Technical Report SERC-TR-63-F, University of Florida, Jun 1993.

44. Lowry, Michael R. and Robert D. McCartney, editors. *Automating Software Design*, *1*. Menlo Park, CA: AAAI Press / The MIT Press, 1991.

45. Lutsky, Patricia. "Automating Testing by Reverse Engineering of Software Documentation." In Wills et al. [78], 8–12.

46. MacLane, Saunders and Garrett Birkhoff. *Algebra* (Third Edition). New York, NY: Chelsea Publishing Company, 1993.

47. Newcomb, Phillip. "Re-engineering Procedural into Object-Oriented Systems." In Wills et al. [78], 237–251.

48. Ning, J. Q., et al. "Automated Support for Legacy Code Understanding," *Communications of the ACM*, *37*(5):50–57 (May 1994).

49. Olsem, Michael R. and Chris Sittenauer. *Reengineering Technology Report*. Technical Report Vol 1, Hill AFB, UT: Software Technology Support Center, Aug 1993. 17 Jul 97.

50. Ong, C. L. and W. T. Tsai. "Class and Object Extraction from Imperative Code," *Journal of Object-Oriented Programming*, *6*(1):58–68 (Mar 1993).

51. Quilici, Alex. "A Memory-Based Approach to Recognizing Programming Plans," *Communications of the ACM*, *37*(5):84–93 (May 1994).

52. Quilici, Alex and David N. Chin. "DECODE: A Cooperative Environment for Reverse-Engineering Legacy Software." In Wills et al. [78], 156–165.

53. Reasoning Systems Inc, Palo Alto, CA. *DIALECT User's Guide*, July 1989.

54. Reasoning Systems Inc, Palo Alto, CA. *REFINE User's Guide*, May 1990.

55. Reasoning Systems Inc, Palo Alto, CA. *REFINE/FORTRAN User's Guide*, March 1994.

56. Reubenstein, Howard B. and Richard C. Waters. "The Requirements Apprentice: Automated Assistance for Requirements Acquisition," *IEEE Trans on Software Engineering*, *17*(3):226–240 (Mar 1991).

57. Rich, Charles. "A Formal Representation for Plans in the Programmer's Apprentice." *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, edited by Michael L. Brodie, et al. 1044–1052. New York: Springer-Verlag, Aug 1981.

58. Rich, Charles and Yishai A. Feldman. "Seven Layers of Knowledge Representation and Reasoning in Support of Software Development," *IEEE Trans on Software Engineering*, *18*(6):451–469 (Jun 1992).

59. Rich, Charles and Richard Waters. "The Programmer's Apprentice: A Research Overview," *IEEE Computer*, 10–25 (Nov 1988).

60. Rich, Charles and Linda M. Wills. "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, *7*(1):82–89 (Jan 1990).

61. Rugaber, Spencer. *White Paper on Reverse Engineering*. Technical Report, Atlanta, GA: Georgia Institute of Technology, Mar 1994.

62. Rumbaugh, James and Michael Blaha. *Object-Oriented Modeling and Design*. New Jersey: Prentice-Hall, Inc., 1991.

63. Shlaer, S. and S. J. Mellor. *Object Lifecycles - Modeling the World in States*. Englewood, Cliffs: Yourdon Press, 1992.

64. Shooman, Martin L. *Software Engineering*. McGraw Hill Book Company, 1983.

65. Smith, Douglas R. "KIDS: A Semiautomatic Program Development System," *IEEE Transactions on Software Engineering*, *16*(9):1024–1043 (Sep 1990).

66. Smith, Douglas R. *Classification Approach to Design*. Technical Report KES.U.93.4, 3260 Hillview Ave: Kestrel Institute, Nov 1993.

67. Sneed, H. "Migration of Procedurally Oriented COBOL Programs in an Object-Oriented Architecture." *Proceedings of the Conference on Software Maintenance*. 105–116. Nov 1992.

68. Sneed, Harry M. and Erika Nyáry. "Extracting Object-Oriented Specifications from Procedurally Oriented Programs." In Wills et al. [78], 217–226.

69. Soloway, E. and K. Erdlich. "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, *10*(5):595–609 (1984).

70. Soloway, Elliot and W. Lewis Johnson. "PROUST: Knowledge-Based Program Understanding," *IEEE Transactions on Software Engineering*, *SE-11*(3):267–275 (Mar 1985).

71. Stroustrup, Bjarne. *The C++ Programming Language*. ATT Bell Labs, New Jersey, Jul 1987.

72. Tennent, R. D. *Principles of Programming Languages*. New York: Prentice-Hall, 1981.

73. Tiemens, Tim. *Cognitive Models of Program Comprehension*. Technical Report, Software Engineering Research Center, Georgia Tech, Dec 1989.

74. Waters, Richard C. "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Transaction on Software Engineering*, *11*(11):1296–1320 (Nov 1981).

75. Waters, Richard C. "Program Translation via Abstraction and Reimplementation," *IEEE Transactions on Software Engineering*, *14*(8):1207–1228 (Aug 1988).

76. Weiser, M. "Program Slicing," *IEEE Transactions on Software Engineering*, *SE-10*(4):352–357 (Jul 1984).

77. Wilde, N., et al. "Dependency Analysis Tools: Reusable Components for Software Maintenance." *Proceedings of the Conference on Software Maintenance*. 126–131. Oct 1989.

78. Wills, Linda, et al., editors. *Second Working Conference on Reverse Engineering*, Los Alamitos, CA: IEEE Computer Society Press, Jul 1995.

79. Yeh, Alexander S., et al. "Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language." In Wills et al. [78], 227–236.

Vita

Major Ricky E Sward [redacted] graduated as the Salutatorian of Lynnville-Sully High School in May 1981. In May 1985 he received a Bachelor of Science degree in Computer Science from Iowa State University. Major Sward was a Distinguished Graduate of the Iowa State University ROTC program. His first assignment was to the Missile Warning Directorate at HQ Strategic Air Command in Omaha, NE. While in Omaha, [redacted] Major Sward was then selected to attend an AFIT/CI degree program at the University of Colorado, Boulder. He received a Master of Science degree in Information Systems and a Master of Science degree in Computer Science in Jul 1991. Major Sward was assigned to the USAF Academy as an Instructor in the Computer Science department. In May 1994 Major Sward received the Outstanding Academy Educator award, which is given to the top instructor in each department. In Sep 1994, Major Sward reported to Wright-Patterson AFB to enter the PhD program. Upon graduation, Major Sward will return to the USAF Academy as an Assistant Professor in the Computer Science department. [redacted]

[redacted]

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 1997	**3. REPORT TYPE AND DATES COVERED** Doctoral Dissertation

4. TITLE AND SUBTITLE Extracting Functionally Equivalent Object-Oriented Designs from Legacy Imperative Code.	**5. FUNDING NUMBERS**
6. AUTHOR(S) Ricky E. Sward, Major, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology 2750 P St WPAFB, OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/DS/ENG/97-04
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Mr Douglas A. White Rome Lab/C3CB 525 Brooks St Rome, NY 13441-4505	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

This research defines a methodology for automatically extracting functionally equivalent object-oriented designs from legacy imperative programs. The Parameter-Based Object Identification (PBOI) methodology is based on fundamental ideas that relate programs written in imperative languages such as C or COBOL to objects and classes written in object-oriented languages such as Ada 95 or C++. Transformations have been developed that formalize the PBOI methodology and a formal proof is provided showing the extracted object-oriented design is functionally equivalent to the legacy imperative system. To focus the task of re-engineering, generic models of imperative programming languages and object-oriented programming languages have been developed. The formal transformations convert imperative subprograms represented in the Generic Imperative Model (GIM) into classes and objects represented in the Generic Object-Oriented Design Model (GOM). A taxonomy of imperative subprograms has also been developed which classifies all imperative subprograms into one of six categories. A proof-of-concept prototype has been developed and a 3000-line FORTRAN-77 system has been converted to an object-oriented design as a feasibility demonstration.

14. SUBJECT TERMS reverse engineering, re-engineering, formal transformations, formal proofs, object-oriented design, imperative programming languages, object-oriented programming languages, software engineering			**15. NUMBER OF PAGES** 345
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL