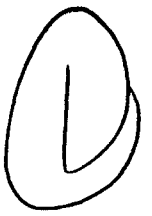
12-1993

A Method for Populating the Knowledge Base of APTAS, a Domain-Oriented Application Composition System

Raleigh A. Sandy III

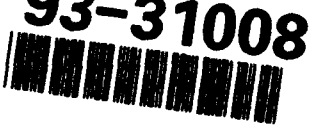


AFIT/GCE/ENG/93D-13

A Method for Populating the Knowledge Base of APTAS, a Domain-Oriented Application Composition System

THESIS
Raleigh Albert Sandy, III
Captain, USAF

AFIT/GCE/ENG/93D-13



AFIT/GCE/ENG/93D-13

A METHOD FOR POPULATING THE KNOWLEDGE BASE OF APTAS, A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

THESIS

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

Master of Science in Computer Engineering

Raleigh Albert Sandy, III, B.S.E.E.

Captain, USAF

December, 1993

Acknowledgements

This thesis is the result of more effort than just my own. I am grateful to all the support I received during this research. The quality of the entire thesis, from style to content, is the direct result of my thesis advisor, Maj David Luginbuhl. Thanks, Maj Luginbuhl, for all your time and patience. May God richly bless you in all your efforts.

Chapters II and III were really a combination of my own work and that of Capt Russell Warner. Thanks, Russ (and Marilee) for all your comments and contributions. I pray you have a healthy son who is as intelligent and helpful as both of you.

Much of the information in Chapter IV resulted from contributions from several Lockheed employees: Stan Jensen, Lori Ogata, and Joe Louie. I sincerely appreciate the quick (and patient) responses to all my questions. It was a pleasure to work with you all.

The formal support was only a small portion of the support I received in writing this thesis. My wife was the cornerstone of anything accomplished in this research. Thanks, Janet, for your patience and encouragement. My love for you has been strengthened through this experience, and I thank God for blessing me with such a perfect wife.

My parents were also strong supporters. Most of the style quality is the result of the editing skills of Pat Sandy. Mom, you are the greatest. A son could not ask for a better mother than you. Dad, thank you for your work ethic, your positive encouragement, and your presence during my most difficult time (*the defense*). Both of you have made the 18 months more endurable through your prayers and encouragement.

What would it all be for without my children? Thanks, Lee, for all the study breaks and reality checks. Someday, son, you will do wonderful things for the Lord. Thanks, Courtney, for the much needed naps. You are just as beautiful and sweet as your mother. I pray that you and Lee will both remain faithful to Christ and will be prepared when He returns.

How about the humor that kept my sanity? Thanks, Dave Mezera, for being an awesome friend (and brother). Oh yace! Thanks, Doug Looney and Tony Moyers, for breakfast relief. Thanks, Jeff Miller and Jay Cossentine, for sound effects and lighting. Dooh!

Even with all this support, something could have gone wrong. However, because of the exceeding grace of the Lord Jesus Christ, I have the assurance that no trial or tribulation will ever be more than I can bear. It was not by my talent and strength, the Lord (through the Holy Spirit) provided all the talent and strength required to accomplish this research. As Ray Boltz so appropriately sings, "No matter what tomorrow brings, or what it has in store, I know I will praise the Lord!"

Raleigh Albert Sandy, III

Table of Contents

List of Figures

Abstract

A formal process is described for populating the knowledge base of the Automatic Programming Technologies for Avionics Software (APTAS) system. This process was developed using a general knowledge base population process that applies to many knowledge-based software engineering systems. This general process is also described. The formal process for APTAS was demonstrated by storing new information from the radar tracking domain into the knowledge base of APTAS. Several procedures (some automatic and some manual) were implemented to support the formal process.

A METHOD FOR POPULATING THE KNOWLEDGE BASE OF APTAS, A DOMAIN-ORIENTED APPLICATION COMPOSITION SYSTEM

I. Research Introduction

1.1 Background

Software system development methods must take advantage of new technologies to catch up to hardware system development methods. Software engineering must become more disciplined and produce better models of software development that capture reliable development practices and formalize engineering knowledge. These new models will improve software system quality and will expedite software system production (9:258).

Producing models of software development depends on advances in many areas of technology (4:92-93). These areas include software reuse, artificial intelligence, formal methods, and object-oriented development. Expediting software system production depends on advances in software reuse. Implementation of software reuse must advance from opportunistic reuse of source code to systematic reuse of software system specifications. Capturing development practices and formalizing engineering knowledge depend on advances in artificial intelligence (particularly in expert systems, knowledge acquisition, and knowledge representation). Improving software system quality depends on advances in formal methods. Practical application of formal methods must advance from representing small-scale software systems to specifying large-scale systems using mathematical formalisms. Becoming more disciplined in software development depends on advances in object-oriented programming. Programming languages and tools that support object-oriented development are changing the way software engineers think about software system development.

Software engineers currently use several different software development methods. The inherent limitations of a particular method often offset the method's benefits. The

waterfall model is well-understood and often easy to manage. However, waterfall model development does not result in a software product (other than documentation) until development completes and rarely involves the users. Incremental development usually results in system capabilities early in development. However, incremental development depends on other development methods and is often not feasible for some software systems. Rapid prototyping also results in system capabilities early in development and definitely involves the users. However, rapid prototyping can result in a system that is not intended for production and that is difficult to maintain. The spiral model is an attempt to combine other development methods through an iterative process to manage risks in software development (24:48). However, spiral model development requires more highly trained personnel because of its complexity. Automatic code generation is also an attempt to improve the other limitations but cannot guarantee efficient code. None of these software development methods, by themselves, provide the foundation needed to catch up to current hardware development methods. However, another development method may provide this foundation; a method involving knowledge-based software engineering.

1.2 Knowledge-Based Software Engineering

Knowledge-based software engineering provides a user with the capability to specify, prototype, and generate new applications within modeled application domains. Users specify applications using familiar domain-specific terminology under the guidance of a knowledge-based software engineering system. Users rapidly prototype applications by executing an their specifications. Users incrementally modify specifications until tested behaviors meet the specific requirements. When prototypes meet the requirements, users automatically generate executable applications directly from the final specifications. Users do not have to know a particular programming language since application development occurs using familiar terminology from the application domain.

The concepts of knowledge-based software engineering are derived from a combination of other software development techniques and key areas of software engineering technology. Users develop their own applications with a language they understand and satisfy their requirements through rapidly prototyping application behaviors. Knowledge-

based software engineering can potentially realize large improvements in software quality and expedite software production.

Application composition systems, a class of knowledge-based software engineering systems, interface with users through a composition environment. This environment helps users to specify (and possibly prototype) applications within a modeled application domain. The environment accesses information stored in a knowledge base. The knowledge base captures current information about both modeled application domains and software development practices. Other environments of these systems, such as a code generation environment, also access information stored in the knowledge base.

A group of software engineers at the Air Force Institute of Technology (AFIT) has concentrated their research on advancing the technology of knowledge-based software engineering systems. The group is studying two knowledge-based software engineering systems: Architect and the Automatic Programming Technologies for Avionics Software (APTAS) system (17, 2, 32, 41).

Architect was developed using the Software Refinery^{TM} Environment to study advances in knowledge-based software engineering. Architect's capabilities range from the application of software architectures to the implementation of domain-specific languages. The system, however, has a limited code generation capability. Current research includes improving the visualization of domain information within the application composition environment (7); formalizing software architectures (10) and the services they provide (42); modeling the real-time circuit domain (38), the digital signal processing domain (39), and the application executive domain (42); and integrating the use of an object-oriented data base (6). Research also includes developing a process to populate the Architect knowledge base (39).

APTAS is a prototype knowledge-based software engineering system developed for the Air Force by Lockheed (17). Although the knowledge base for APTAS is well-structured, the system is currently limited to composing and simulating simple radar tracking applications. This thesis focuses on studying the APTAS knowledge base.

1.3 Knowledge Bases - A Population Problem

It has already been mentioned that many knowledge-based software engineering capabilities depend on the information stored in the knowledge base. Systems like Architect and APTAS expect particular forms of information from their knowledge bases. The knowledge base organizes information about a modeled application domain into several categories. Domain-specific information might represent entity structures and their behaviors, as well as rules about the composition of applications within the domain. Other information might represent particular methods the system employs when specifying, simulating, or generating applications. All this information must either already exist in the knowledge base or some method must exist to store new information. The APTAS system has limited information stored in its knowledge base. APTAS documentation includes a brief description of a process to populate the knowledge base with new information. However, this process is too informal and too complex. The goal of this thesis is to solve the following problem:

> **There is currently no understood formal process to efficiently capture and store new domain information within the APTAS Knowledge Base.**

This thesis proposes a documented population process to store new information in the APTAS Knowledge Base, known as the Tracking Taxonomy and Coding Knowledge Base.

1.4 Research Scope

This thesis documents a formal process to populate the Tracking Taxonomy and Coding Knowledge Base. The formal population process results from a general process outline that applies to many knowledge-based software engineering system. For instance, Warner (39) has used this general process outline to study the same problem with the Architect system. The intention is to suggest important characteristics of the population process applying to all knowledge-based software engineering systems with a specific focus on developing the formal knowledge base population process for APTAS.

This thesis then demonstrates the process by storing new information from the radar tracking domain in the knowledge base. The population process requires expert knowledge

from an application domain. Domain information normally originates from an expert through a formal knowledge acquisition process. Because information was not acquired using a formal knowledge acquisition process, only enough new information was added to sufficiently demonstrate the formal population process.

Finally, this research results in several procedures that automate some of the population process activities. These procedures simply demonstrate the formal nature of the process and the impact of the process to software system development. Their use is not intended for anything other than knowledge base population research.

1.5 Research Approach

The APTAS knowledge base population problem is solved in several steps. After some of the research related to this problem is analyzed, this thesis outlines a general knowledge base population process. Based on documentation about the APTAS system and the general process outline, this thesis develops a formal population process for the Tracking Taxonomy and Coding Knowledge Base. This thesis concludes with a process demonstration and several suggestions for further research.

Chapters II and III were co-written with Warner (39) because of the similarity between the two research efforts. Chapter II describes published literature that relates to knowledge-based software engineering and the knowledge base population problem. Subjects studied included domain analysis, domain modeling, and systematic reuse. Other general areas of interest include knowledge acquisition, automatic code generation, and formal methods. Some of the fundamental concepts for the general process outline were derived from several articles about software reuse and domain analysis.

Chapter III describes a general population process that applies to a specific class of knowledge-based software engineering systems. The process consists of five activities ranging from domain-specific, system-independent domain analysis activities to system-dependent implementation activities. The chapter concludes with a discussion of several constraints that are involved when applying the process to specific systems.

Chapter IV analyzes the APTAS system and its knowledge base prior to instantiating a formal process. I examine the relationship between the various system components to the knowledge base. I also define the specific categories of information stored in the Tracking Taxonomy and Coding Knowledge Base. The formal population process uses definitions for the structures of the application domain and the knowledge base. Procedures for the population activities result from the application of the general process outline.

Chapter V demonstrates the formal population process by adding new information to the knowledge base. I develop a simple model of the radar tracking domain and use it to populate the APTAS Knowledge Base. The procedures that automatically transform the radar tracking domain into the knowledge base representation show the potential acceleration in software production. The procedures that evaluate each population activity show the potential improvement in software quality.

Chapter VI summarizes the results of this thesis. I suggest several possible topics for future research. These future research topics include populating knowledge bases in general and studying the APTAS system.

II. Literature Review[1]

2.1 Introduction

Many researchers are studying methods to encapsulate knowledge needed for software engineering into reusable models. They have proposed ideas to improve knowledge-based software engineering and shorten the gap between software and hardware system development. Some of this research is directly related to our knowledge base population problem.

The technology involved in the effective modeling of application domains is very important to the success of knowledge-based software engineering. Software engineers must develop formal knowledge acquisition processes that solve the knowledge base population problem. Section 2.2 reviews ideas we found useful in our research from current literature in the areas of domain analysis and domain modeling. Both domain analysis and domain modeling focus on the effective modeling of application domains. "There is a strong relationship between [domain analysis] and knowledge acquisition. Building a knowledge base and defining heuristics for an expert system are basically the same problems as [domain analysis]" (11).

Systematic reuse is another topic that supports knowledge-based software engineering. Arango defines systematic reuse as an activity "in which information is systematically acquired and reused in software construction under the control of some management guidelines and costing models" (3:84). The knowledge base must support the code generation component of a knowledge-based software engineering system by providing a library of reusable software specifications or implementations. Section 2.3 reviews ideas we examined in the area of systematic reuse.

There is a class of knowledge-based software engineering systems known as application composition systems that employ techniques like automatic code generation and formal methods to transform specifications into executable code. Because both Architect and APTAS fall into this class of systems, we studied the characteristics of application composition systems. Our study focused on the application composition process and the

[1]This chapter was co-written with Russell Warner and also appears in (39).

knowledge base structures. Section 2.4 describes the composition process and knowledge base structures of several application composition systems.

Several terms require definition before we begin our review of the related research. We have already used the term **application domain** that we define as "a coherent set of systems that exhibits common features and functionality across existing and proposed instances" (28). Information occurring within the scope of an application domain, such as functional behaviors and parameters, is considered **domain knowledge**. The term **domain analysis** was first introduced by Neighbors as "the activity of identifying objects and operations of a class of similar systems in a particular problem domain" (26). Prieto-Díaz later defined domain analysis as the process where "information used in developing software systems is identified, captured, and organized with the purpose of making it reusable" (31:47).

2.2 Domain Analysis

Domain analysis is the first, and probably most important, step in adding new information to a knowledge base. Domain analysis was originally adopted as a process to automate several aspects of software development including specification analysis, verification, and application generation (3:82). Early research into domain analysis uncovered the importance of organizing domain knowledge into reusable components. Researchers learned that identifying specific knowledge to reuse through domain analysis was no easy task. Neighbors discovered that "the key to reusable software is captured in domain analysis in that it stresses the reusability of analysis and design, not code" (27) and later proposed a domain analysis method called DRACO. Other researchers have also proposed domain analysis approaches, and we summarize some of these in the following paragraphs.

Prieto-Díaz (30) proposed the data flow model shown in Figure 2.1 that represents his domain analysis approach. In his model, the domain expert (a knowledgeable person in that particular field) and domain analyst (a person with training and experience in analyzing domains) identify and select the domain knowledge. Possible sources for domain knowledge include expert advice, customer surveys, technical literature, and existing implementations, as well as current and future requirements. The domain analyst then assists the domain

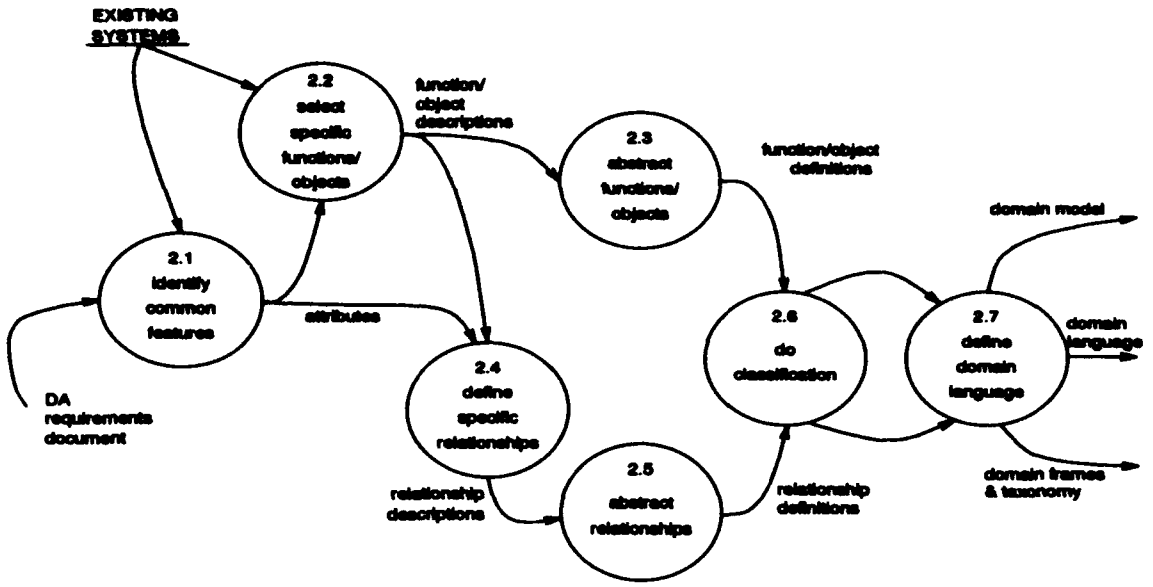


Figure 2.1 Domain Analysis Approach Proposed by Prieto-Díaz (30:67).

expert in abstracting and encapsulating the collected domain knowledge into a subset of the expected outputs (i.e., domain model, domain language, domain taxonomy), as well as domain standards and reusable components. The entire approach is implicitly iterative.

Arango based his view of domain analysis on "the *systematic and incremental approximation* to a definition of an ontology and semantics for a problem domain" (3:83). He proposed an operational definition of domain analysis focused on a reuse-based task (3:83):

Given:

1. a partial, nonformal description of a problem domain
2. a model of a reuser as a learning system

Find: a systematic method to

1. *identify* information in the problem domain which, if available to the reuser in appropriate form, would allow it to attain a specified level of performance,
2. *capture* the information identified as relevant, and
3. *evolve* the acquired information to enhance or maintain the performance of a reuser.

The domain analysis results in a model of the application domain. As with the approach proposed by Prieto-Díaz, this approach identifies and collects reusable domain knowledge.

However, Arango's approach also compares the performance of the reuse-based task to a desired performance level. The domain analysis works to improve the reuse-based task until it reaches the desired performance. Therefore, Arango modeled the reuser as a learning system where improvements to performance correspond to subsequent iterations of domain analysis.

McCain (25) proposed a domain analysis approach consisting of two separate tasks. The first task, application domain analysis, identifies a hierarchy of components and their associations. Application domain analysis is basically the same as other domain analysis approaches studied. This task has three activities: define reusable entities, define reusable abstractions, and perform classification of reusable abstractions. The second task, component domain analysis, defines the individual component behaviors and requirements. This task has four activities to define component interfaces, constraints, algorithms, and customization requirements. McCain's approach is different from other domain analysis approaches by his explicit inclusion of a component domain analysis task.

Kang and others (18) proposed a domain analysis approach called Feature-Oriented Domain Analysis (FODA). The approach identifies prominent features (similarities) and distinctive features (differences) of software systems within an application domain. The features also define mandatory, optional, and alternative characteristics of software systems in the domain. Unlike the other domain analysis approaches we have summarized, the researchers described FODA in sufficient detail to use on large domain analysis projects (ones with several domain analysts). However, this depth of detail can restrict the applicability of the approach.

The Domain Analysis Working Group Report (37) described two domain analysis approaches. The first approach lists the common steps found in other domain analysis approaches:

1. Define Domain Analysis
2. Identify and scope the domain
3. Select a representative set of systems to study
4. Gather inputs for the domain analysis
5. Perform feature analysis at the requirements level

6. Analyze separability, selectability, and trade-offs of features
7. Select an implementation technology
8. Implement and validate products in phases
9. Deliver products of domain analysis

However, they give no real explanation of how a domain analyst accomplishes these steps. They give more detail for the second approach:

1. Acquire knowledge
2. Perform high-level functional analysis (top-down)
3. Identify objects and operators (bottom-up)
4. Define domain models and architecture

A different view of domain analysis was proposed by Iscoe (15). He focused on the results of the domain analysis rather than a specific approach or the inputs to the domain analysis. He suggested the problem was "to create a model for domain knowledge that is general enough to be instantiated in several domains" (14:299). His approach involved developing levels of "meta-models" that a domain analyst uses to capture the information of a particular application domain. Models consist of objects and their attributes, along with the operations performed upon those objects. This approach had two distinct characteristics: (1) attributes and operations are defined in terms of their underlying scales and (2) object classes use multiple inheritance. Iscoe's approach to domain analysis is known as domain modeling.

Domain modeling is a subset of domain analysis. It is a formal approach to capturing domain knowledge into a specific form that results in a defined knowledge structure called a domain model. We define **domain modeling** as the process of organizing and encapsulating information within an application domain into a predefined knowledge structure. The structure for a domain model depends on the specific application domain being modeled as well as the domain analysis approach applied.

Prieto-Díaz suggested that the structure of domain models "range in level of complexity and expressive power from a simple domain taxonomy to functional models to domain languages" (31:51-52). He defines a domain language as "a collection of rules that relate objects and functions and which can be made explicit and encapsulated in a formal

language and further used as a specification language for the construction of systems in that domain" (30:66). Arango suggested that a domain language documents a "shared paradigm [that] is a precondition for domain analysis" (3:82).

Neighbors developed a domain modeling approach with his DRACO system (27), one of the first systems to specifically employ domain languages and domain models. His approach involved a hierarchy of domains consisting of different levels of abstraction. Domains at the highest level of abstraction are called application domains. Domains at the lowest level of abstraction model conventional programming languages and are called executable domains. Those domains in between are called modeling domains. Application domains span several modeling domains. A domain language defines the external syntax of an application domain. The domain language semantics are written in Backus-Naur form and augmented with control mechanisms.

The domain analysis approaches we have described above are a sample of those approaches published. The software engineer has many options for using or modifying an existing approach. Wartik and Prieto-Díaz (40) presented a strategy for comparing different domain analysis approaches that included the following criteria:

- definition of "domain"
- determination of problems in the domain
- permanence of domain analysis results
- relation to the software development process
- focus of analysis
- paradigm of problem space models
- purpose and nature of domain models
- approach to reuse
- primary product of domain development

Software developers can use these criteria to choose a domain analysis approach that meets their objectives and is within their current resources.

2.3 *Systematic Reuse*

Wartik and Prieto-Díaz also described three categories of reuse: ad hoc, opportunistic, and systematic (39). Ad hoc reuse is reuse without any formal reuse method. Opportunistic reuse is a software development process with methods to identify the types of reusable components, when to use them, and where they might be found. Systematic reuse is a software development process with methods to define and construct reusable components. They suggested that a software development process could not realize systematic reuse without including the role of domain analysis.

The success of knowledge-based software engineering systems, such as application composition systems, depends on the practice of systematic reuse. Prieto-Díaz suggested that domain analysis could realize systematic reuse by successfully "deriving common architectures, generic models or specialized languages that substantially leverage the software development process in a specific problem area" (31:47). He provided an example of how domain analysis might fit into the software development process (shown in Figure 2.2). Prieto-Díaz claimed that this concept could support several methods of software

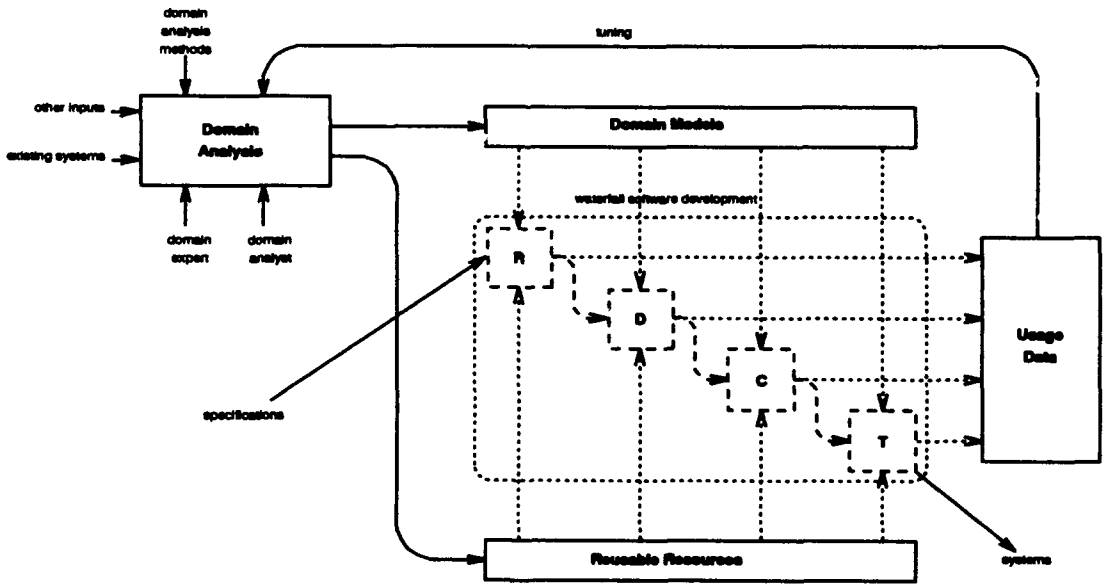


Figure 2.2 Domain Analysis and Software Development (31:52).

development other than the waterfall model. He called this concept a **reuse infrastructure** and stated:

development other than the waterfall model. He called this concept a **reuse infrastructure** and stated:

> Domain models, in a variety of forms, support (i.e., control) the different phases of software development. Reusable resources are selected and integrated in the new system. Reuse data is then collected and feedback to domain analysis for refining the models and for updating the library. As developed systems become existing systems they are also used to refine the reuse infrastructure (31:52).

Neighbor's DRACO system generates software systems from abstract specifications using its hierarchy of domains. A specification begins in an application domain and gets refined by the system through modeling domains until it can be implemented using an execution domain containing reusable components.

Reusable components, like those in DRACO's execution domains, are very important to systematic reuse. The reusable components must be constructed using some consistent structure called a software architecture. Software architectures define a consistent component structure and also define how to compose applications using a domain's components. Researchers at the Software Engineering Institute have studied systematic reuse and have developed a software architecture called the Object Connection Update (OCU) model (19). Figure 2.3 shows a subsystem in the OCU architecture. Applications are composed of at

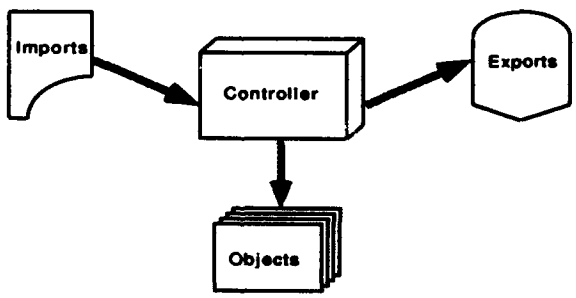


Figure 2.3 An OCU Subsystem (19:18).

least one subsystem under the control of an application executive. Subsystems consist of imports, exports, a controller, and objects. Objects consist of inputs, outputs, and update functions. Gool (10) summarizes the OCU model as well as several other documented software architectures.

2.4 Application Composition Systems

Both domain analysis and systematic reuse play important roles in knowledge-based software engineering systems, especially in the class of application composition systems. There are several application composition systems in use today. Anderson (2), Randour (32), and Weide (40) developed the initial version of Architect, which is an application composition system that implements the OCU software architecture. In this system, domain information is captured in reusable objects at the specification level. Along with these reusable objects is information specifying a domain-specific language (DSL) and visualization specification language (VSL). An Application Specialist (the person creating the software system, called an application) composes applications by either entering a textual specification using the DSL or by visually manipulating icons specified by the VSL. The Architect system is undergoing further study at the AFIT (see research by Gool (10), Warner (38), Cossentine (7), Welgan (41), and Waggoner (37)).

The Lockheed Software Technology Center, under contract with the United States Air Force, prototyped an application composition system (17). This system, called APTAS, "automatically synthesizes executable code from high-level tracking system specifications" (17:1). APTAS generates applications through the support of its Tracking Taxonomy and Coding Knowledge Base. The system uses a software architecture enforced by the knowledge base structure; however, there is no method defined to store information (on existing or new domains) into its knowledge base.

Several other application composition systems exist. Some of these include the Kestrel Interactive Development System (KIDS) developed at Kestral Institute (35), the Khoros system developed at the University of New Mexico (33), and the Intelligent Design Aid (IDeA system) developed at the University of Illinois (23).

2.5 Summary

Software reuse has come a long way from ad hoc reuse of low-level code. The reuse of high-level abstractions capturing domain knowledge has become a reality through technologies like domain analysis and domain modeling. Software architectures, combined

with domain analysis, have made it possible for researchers to build software development systems that practice systematic reuse of both low-level code and high-level abstractions. Application composition systems, like Architect and APTAS, have shown that the users themselves can develop their own software systems in a familiar language and environment. Ongoing research in domain analysis and systematic reuse will provide more insight in the development of more operational application composition systems and modeling more application domains. These advances promise to improve the software development process drastically.

III. Knowledge Base Population Methodology[1]

3.1 Introduction

Many researchers have envisioned software development evolving from the art of hand-writing code to the engineering discipline of combining and specializing reusable components. One such researcher, Michael Lowry (22:630), envisioned Knowledge-Based Software Engineering environments that automate software reuse using domain knowledge captured through domain analysis. The KBSE research group at AFIT is developing formal methods to implement the automated reuse that Lowry envisioned in the above paragraph. The group's work is based on several of Lowry's premises. This thesis is part of the group's work, and it is focused primarily on how to capture the reusable components that Lowry described into an automated software development system (i.e., how to populate the system's knowledge base).

This chapter proposes a general process that can be tailored to populate the knowledge bases of a particular class of software development systems. Section 3.2 defines the particular class of software development systems. Section 3.3 presents our view of domain models and their corresponding knowledge base representations. Section 3.4 describes the domain analysis methods that we found helpful in developing our process. We use these methods in Section 3.5, along with our system definition and domain model view, to develop a general process to populate a system's knowledge base. Finally, in Section 3.6, we support the development of a "general" process and define several constraints to its implementation.

3.2 Generic Domain-Oriented Application Composition System

We developed a knowledge base population process for a specific class of software development systems. There are several characteristics that distinguish this class of systems. Each system has a knowledge base and a process to compose applications. The knowledge base stores reusable components. Applications are specified using these components. Users can modify, save, and maintain applications through the composition process. Also

[1]This chapter was co-written with Russell Warner and also appears in (38).

through the composition process, users can simulate the execution of application (before code is created), translate them to some external form (i.e., synthesize code), and execute them outside the system. These characteristics, including the ability to synthesize executable code, describe the class of application composition systems. However, our class of systems has a knowledge base that must be organized into application domains in an object-oriented fashion. Therefore, we refer to this class of systems as **Domain-Oriented Application Composition Systems** (DOACS).

There are several advantages to this type of system. The most important advantage is systematic reuse. Reuse is not limited to the code level, but occurs at all levels, primarily at the specification level. Maintenance also occurs at the specification level instead of at the more difficult code level. The application composition process, with its ability to simulate specification behaviors, provides an ideal environment to develop rapid prototypes. This type of system can also provide the powerful capability of creating systems within a graphical environment. The user works with the components and does not have to possess expert knowledge of the domain (e.g., specific algorithms). A user does not need traditional programming knowledge to create new applications, nor to maintain existing applications, because the system automatically generates applications from their specifications. Users can possibly choose between different hardware platforms and programming languages when generating code. Finally, these systems could automate the "housekeeping" chores (e.g., configuration management) so users can concentrate on the more important task of application specification.

It is important to make the distinction between domain-oriented and domain-specific application composition systems. A domain-specific system can be used to compose applications in only one domain and new domains cannot be added. A domain-oriented system can be used to compose applications in any domain implemented in that system and more domains can be added. While there are similarities between these two types of systems, the fact that domain-specific systems contain only one domain greatly simplifies the creation of the system and, of course, nullifies the problem of knowledge base population since the domain information is integrated into the system when it is built. Although domain-specific systems are limited to one domain, the creators of the system are able to take advantage

of the features of that particular domain when designing the system (e.g., the system can be custom tailored around the architecture that best fits that particular domain). Because of this, domain-specific systems have an advantage over domain-oriented systems in ease of composition and capabilities in that particular domain. However, it is not practical to build domain-specific systems for every application domain. Due to their modularity, it is easier to update domain-oriented systems with new software engineering techniques. Also, only one system has to be updated to take advantage of any new technique for many domains as opposed to updating several domain-specific systems (it would be a significant effort to update each system separately).

Figure 3.1 contains the major characteristics of our generic DOACS (G-DOACS). We

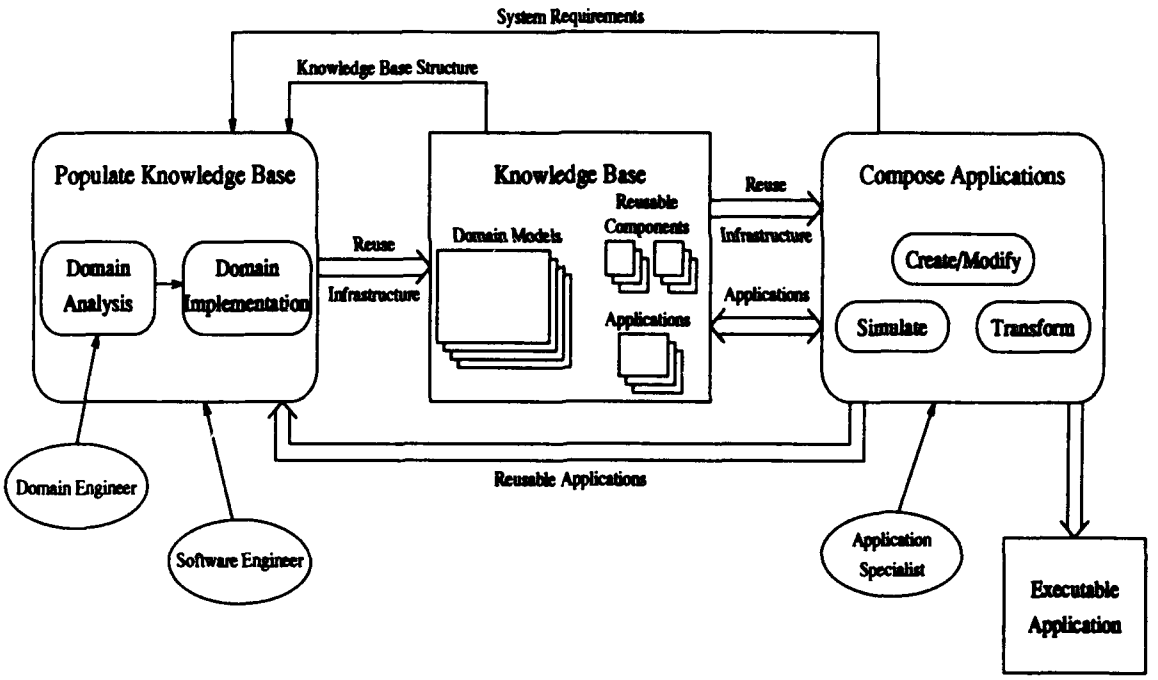


Figure 3.1 Generic Domain-Oriented Application Composition System (G-DOACS)

have used rounded boxes to represent processes and subprocesses (e.g., **Compose Applications** and **Create**), regular boxes to represent physical structures (e.g., **Knowledge Base**), and ovals to represent the roles of the people involved. Notice that we have also included another important characteristic to the G-DOACS definition, the addition of a **Populate Knowledge Base** process that performs the actual knowledge base population.

3.2.1 Compose Applications. An **Application Specialist** uses the Compose Applications process to create, modify, and validate (through simulation) software applications. The application can then be transformed into executable code. The specific methods to accomplish this process may vary from one DOACS to another.

In general, the Application Specialist creates an application by choosing components from a domain-specific component library, specifies how those components connect together, and declares any necessary processing information. The Application Specialist does not add any new functionality (i.e., no new code), but does specify a component's particular functionality by setting various component attributes. The ability to specify components from several domains within a single application is not a requirement for our class of systems, but this is a desired capability.

Rapid prototyping can be easily accomplished through the simulation capability. The Application Specialist can quickly compose an application and simulate its execution. If the behavior meets the requirements, then the Application Specialist can continue to refine the application; if not, then the Application Specialist can modify the application or throw it away and start over.

We use the term "simulating" rather than "running" because no code has been generated. The system uses the current application specification and any selected reusable components to simulate the behavior that would be expected if code had been generated and executed. Through simulation, the Application Specialist can validate the application's behavior and modify the specification until it meets the desired behavior.

After the application is validated, the Application Specialist can transform it into an executable form (i.e., synthesize code) for a particular target platform. At any stage of application development, the Application Specialist can save the current application specification. This environment also supports application maintenance by allowing the Application Specialist to load and then modify the application.

3.2.2 Knowledge Base. Although specific knowledge bases vary, every DOACS knowledge base contains at least three distinct types of information: applications, reusable components, and domain model representations.

Applications are compositions of the reusable components, along with composition information (e.g., how they are connected, execution order). Therefore, applications contain either links to the reusable components they employ or copies of each reusable component. If links are maintained, then the attribute values that have been changed by the Application Specialist are also saved.

Reusable components (we will often refer to them as just components) are the objects that are connected together to build applications. Reusable components are either primitives or reusable applications. **Primitives** are independent objects that capture the behavior and attributes of objects and classes specified during domain analysis and identified in the domain model. **Reusable applications** are those applications (or parts of applications) identified for potential reuse within future applications. They are somehow processed to make them available to the Application Specialist for composition into applications just like the primitives.

Domain model representations are formal structures that organize the reusable components and other domain-specific information (such as data types, semantic rules, and perhaps even specific architecture information) within an application domain. The types of information in a particular representation depend primarily on the specific application domain and on the chosen approach to domain analysis. We discuss our view of domain models and their representations in more detail in Section 3.3.

3.2.3 Populate Knowledge Base. The **Populate Knowledge Base** process is the focus of our research and is the topic of the rest of this chapter. Briefly, knowledge base population is a process in which the **Domain Engineer** captures domain information as high-level abstractions, and the **Software Engineer** represents these abstractions in a form that is stored directly in a particular system's knowledge base.

In our general process, population begins by selecting an object-oriented **Domain Analysis** approach. The Domain Engineer models a particular application domain using the domain analysis approach chosen. The result of the domain analysis is a domain model and individual component abstractions (i.e., component behavior definitions).

The Software Engineer uses the domain model and the individual component abstractions to create the formal structure of that domain in the knowledge base. We call this construction of a domain model representation the **Domain Implementation**. The domain model representation is a particular instantiation of a domain model and the individual component abstractions for a specific DOACS knowledge base. Once instantiated, the Software Engineer adds the domain model representation to the knowledge base. The Application Specialist can then access any new information when composing applications in that domain.

We borrowed the term **reuse infrastructure** from Arango (3) and Prieto-Díaz (31) to refer to a domain model representation. We developed our process to keep the domain model as independent of the particular DOACS and its knowledge base structure as possible. This independence delays (for as long as possible) the addition of any particular system constraints to the analysis process. The view of a domain model being different from its reuse infrastructure is important to our development of a general knowledge base population process.

3.3 Domain Models and Reuse Infrastructures

The terms domain model and reuse infrastructure are central to our research and, in our opinion, are ill-defined in the current literature where they take on many different meanings. In this section, we define the meanings of these two terms with respect to our research.

Many researchers have viewed the results of a domain analysis as a set of reusable software components and composition rules that capture and implement the semantics of applications within the domain. Given this interpretation, however, Domain Engineers must know the particular knowledge base structure before completing their domain analysis. Domain analysis becomes a task of finding, identifying, organizing, and implementing reusable components. Domain Engineers become the people responsible for populating the knowledge base and collecting the results of their domain analysis within the knowledge base structure itself.

This approach can lead to quick and efficient domain implementations, but can also lead to several problems because of inherent limitations in any knowledge base. Because all domain information cannot be stored in a particular knowledge base, it is difficult to reuse the domain analysis results to populate the knowledge base of another DOACS. Also, problems can occur when identifying or changing the design of the domain model or making other changes as new information is discovered (e.g., better domain implementation methods), because some domain information may have been lost through design decisions when populating the knowledge base. In addition, if the Domain Engineer views a domain through the structure of a particular knowledge base, it will influence the interpretation of domain knowledge and may result in missed or incorrect domain encapsulation. This problem is similar to the difficulty in identifying seemingly simple solutions to a problem when viewing it through the wrong paradigm.

For these reasons, among others, we make a distinction between the results of the domain analysis (domain model and component abstractions) and the implementation of the domain (or reuse infrastructure) in the knowledge base. Therefore, although many researchers have assumed the domain model and its reuse infrastructure (domain model representation) are one and the same, we agree with Arango:

> Models of domains and reuse infrastructure should be treated as separate entities, conceptually and practically. Models of domains capture the results of the learning process in domain analysis and support the application of learning techniques. Reuse infrastructures are specified to support the efficient reuse of the information from the model in particular environments (3:88).

This division between domain analysis and domain implementation that we are proposing is similar to a division in compilers. Compiler theory makes a distinction between the intermediate code generated by the front end (analogous to our Domain Analysis) and the machine code generated by the back end (analogous to our Domain Implementation). The front end of the compiler includes those portions of the compiler "that depend primarily on the source language and are largely independent of the target machine ... [while] the back end includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language" (1:20). The front end can be created once for a language, and then different back

ends can be combined with it to create compilers for different machines. In our generic methodology, we propose that domain analysis and domain implementation are analogous to the front and back ends of a compiler. The results of one domain analysis can be used to populate different DOACSs.

3.3.1 Domain Model. The software engineering community has many different views on what constitutes a domain model. In G-DOACS, a domain model is a structure that captures an application domain's individual components (including their attributes and operations), relationships between components, and other related information (such as shared data types, global operations, composition rules, and architecture information). Prieto-Díaz suggested that the the purest form for a domain model would be a domain language (31:52) that captures all the information about a domain listed above. The syntax of such a language would capture the types of components (with their attributes) and the ways they can be combined, while the semantics would capture the the behaviors of component combinations.

For our process, we chose not to include the individual component behaviors (semantics) as part of the domain model. We separated the component behaviors from the domain model because defining behaviors is often the most difficult task during domain analysis. Also, users can compose applications with a well-defined domain model implementation but with only partially defined (and implemented) component behaviors. This allows a Domain Engineer to quickly capture a small "core" of domain knowledge (the domain model plus a few of the component behaviors) that, once implemented, provides the Application Specialist the capability to compose simple applications before many of the component behaviors have been defined. Under our definition, the domain model contains only the information that fully describes the syntax of an application domain; individual component behaviors are defined and implemented separately.

In this chapter and those that follow, we describe two instances of the domain model. The first is the domain model created during the domain analysis process (as described in the preceding paragraph); the second is the implementation of the domain model in the

knowledge base. The instance we are identifying with the term domain model should be clear from the context.

3.3.2 Reuse Infrastructure. As stated previously, the results of the domain analysis should be independent of any particular DOACS. So ideally the domain analysis outputs a domain model and component abstractions without constraints to their usefulness to any particular DOACS. This approach follows the established software engineering practice of pushing design decisions down to the lowest possible level. If the domain analysis is done correctly, the domain model and component abstractions can be evolved over time without the need to reanalyze the whole domain. The domain analysis results can also be used to populate the knowledge base of any DOACS (i.e., the results are reusable[2]).

Since the domain analysis results are derived independent of the knowledge base structure, the system cannot use them to generate applications. Therefore, the Software Engineer must organize and implement the domain model and component abstractions into the correct knowledge base representation for some particular DOACS. We call this instantiation of the domain analysis results the reuse infrastructure. The reuse infrastructure implements the information captured in the domain model and component abstractions in the form required by the structure of a particular knowledge base. Traditional software engineering methods apply to the development of any reuse infrastructure.

Separating the reuse infrastructure from the domain analysis results allows us to develop our knowledge base population process without introducing constraints too early in the process. Before presenting our process, we will discuss several theories that were helpful in our research.

3.4 Domain Analysis Research

In Chapter II, we summarized the current literature in the field of domain analysis and discussed the relation between this field and our research in knowledge base population. This section summarizes some of the contributions of two recognized researchers in the

[2]Methods to capture domain information in an object-oriented database feeding the knowledge bases of several DOACS are addressed by Cecil and Fullenkamp (6).

field: Prieto-Díaz and Arango. Prieto-Díaz (30) proposed a functional model of a domain analysis process while Arango (3) explored the domain analysis process in a formal software reuse system. Our process is built upon many of their contributions.

3.4.1 Prieto-Díaz's Research. According to Prieto-Díaz (30), domain analysis captures the "essential functionality" of components, which assists the application developer. He proposed the data flow diagram in Figure 3.2 with three activities that are involved with domain analysis: prepare domain information, analyze domain, and produce reusable workproducts. These three activities comprise the task of knowledge base population.

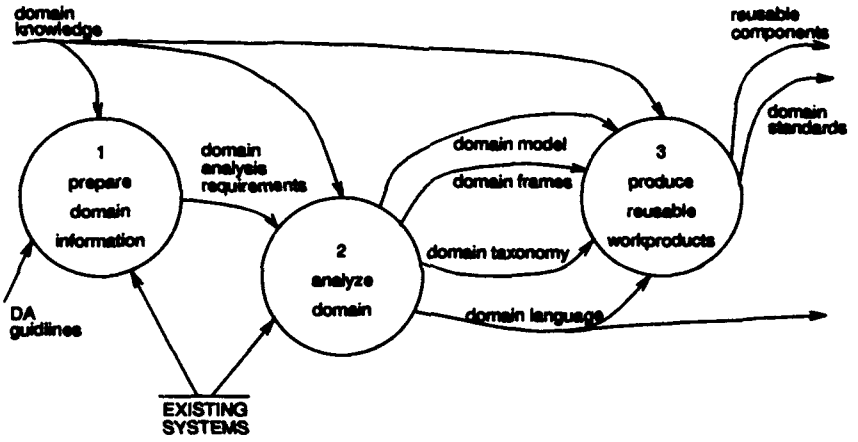


Figure 3.2 Producing Reusable Workproducts (30:67).

The prepare domain information activity produces the requirements of the domain analysis. This activity includes bounding the application domain, identifying the sources for domain knowledge, selecting a specific domain analysis approach, and defining the expected results.

The analyze domain activity uses these requirements to produce collections of domain abstractions including the domain model, domain frames, a domain taxonomy, and a domain language. This activity is the domain analysis theory proposed by Prieto-Díaz – we previously presented the details of the analyze domain activity with the data flow diagram in Figure 2.1. The domain abstractions capture the behavior of objects within

the domain, identify their relationships, and model the structure of these relationships. Prieto-Díaz suggested that the ideal result of domain analysis is the domain language.

The produce reusable workproducts activity takes the domain abstractions and produces a set of reusable components. The components implement the objects and relationships identified in the domain model and used in the domain language.

Prieto-Díaz proposed a functional model that successfully identifies the relationship between domain analysis and the production of a reusable infrastructure. It also successfully defines several outputs involved in the process; however, it does not sufficiently describe the requirements of each activity. For instance, there is nothing that constrains the structure of the reusable components. Systematic reuse cannot be realized without such constraints. His model does not explicitly capture the importance of feedback and iteration in the domain analysis process, nor does it identify the role of the knowledge base in separating the reuse infrastructure from the domain model.

3.4.2 Arango's Research. Arango (3) outlined a "domain engineering framework" based on the concepts of software reuse. His framework serves as a structure for synthesizing a tailored domain analysis process. Arango suggested his framework has general application because reusers are modeled as learning systems.

Figure 3.3 describes the learning component using boxes from the Structured Analysis and Design Technique. The component consists of three activities and is defined by a set of state variables (3:84):

- **Exp**: expertise in the domain
- **ReuseLog**: feedback from the reuse task
- **RI**: reuse infrastructure
- **TL**: technologies to support the representation and evolution of the domain model
- **MoD**: domain model
- **Lc**: method to increase coverage
- **Le**: method to improve efficiency

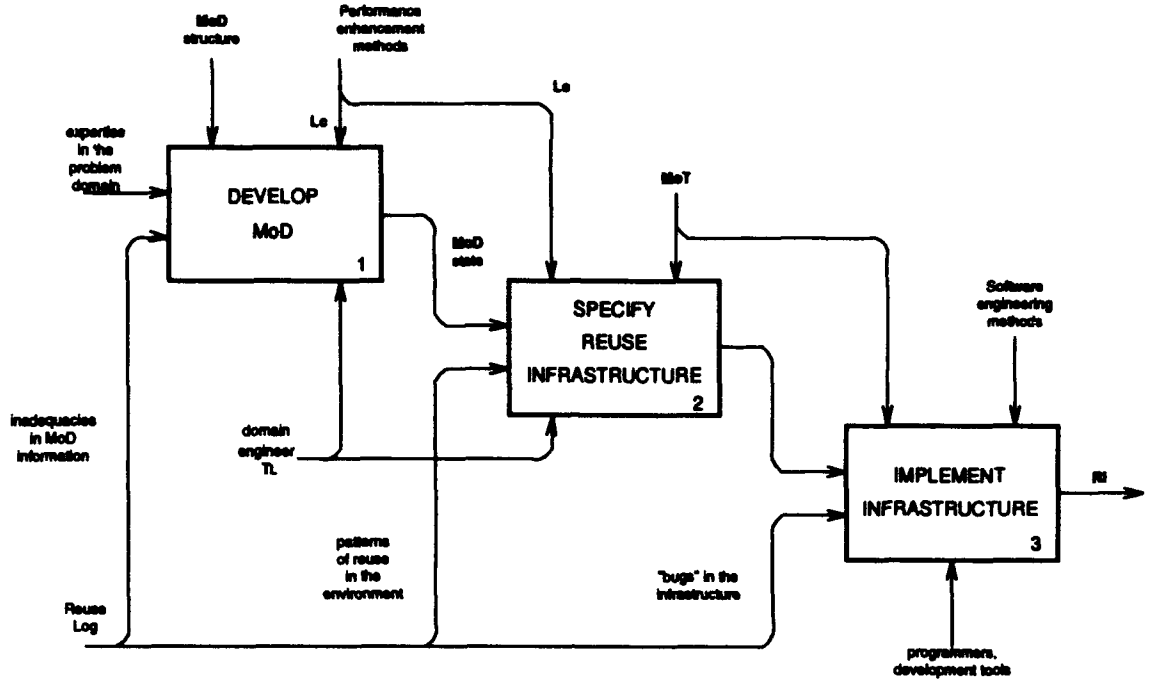


Figure 3.3 Develop Reuse Infrastructure (3:85).

- **MoT**: model of the reuse task

The develop MoD activity represents the actual domain analysis and results in a domain model. Arango made a clear distinction between the domain model and the reuse infrastructure. He stated that the distinction is analogous to the distinction between "representations for systems specifications and programming languages for systems implementations" (3:82). This distinction allows the domain model to be independent from the **MoT**.

The other two activities use the current state of the domain model to organize and implement a reuse infrastructure. The particular organization and implementation depend heavily upon the particular **MoT**. Traditional software development procedures are applicable during these two activities.

Arango's learning component is very similar to the functional model proposed by Prieto-Díaz. It separates the creation of a domain model from the development of the reuse infrastructure. Arango successfully identified the various traits involved in the development

of a reuse infrastructure. He also explicitly indicated the roles of feedback and iteration. However, his state variables were too ambiguous. He did not sufficiently describe the model of the reuse task or the role of the knowledge base, which are essential in developing the reuse infrastructure. There was also no clear distinction between the results of infrastructure specification and those of infrastructure implementation. Prieto-Díaz had combined these two activities into his produce reusable workproducts, but Arango seemed to think there should be some distinction. Our research combines these two processes into a single knowledge base population process. We attempt to clarify all traits involved in knowledge base population without adding additional constraints to the process.

3.5 Knowledge Base Population Process

We propose a five-step process to knowledge base population that explicitly identifies the roles of the Domain and Software Engineers, incorporates feedback, and iteratively captures a domain (in stages). Each step has defined inputs, outputs, methods, and constraints. Our process is shown in Figure 3.4, using boxes from the Structured Analysis and Design Technique. The methods (bottom arrows) driving each activity or step must be selected prior to performing the activity. The constraints (top arrows) show the requirements that drive each activity. The inputs and outputs (left and right arrows, respectively) show the domain information as it is captured and passed from activity to activity.

Activities one and two compose our domain analysis process and result in a domain model and component abstractions. As discussed in previous sections, this domain analysis is independent of the particular DOACS. Activities three and four comprise our domain implementation process that results in a reuse infrastructure and, as can be seen from the constraints, that is very dependent on the particular DOACS. This division between domain analysis and domain implementation is such that one domain analysis can feed multiple domain implementations for different DOACS. Activity five provides an evaluation and feedback mechanism to iteratively improve the domain model, component abstractions, and the reuse infrastructure. The remainder of this section defines each activity and all its associated traits.

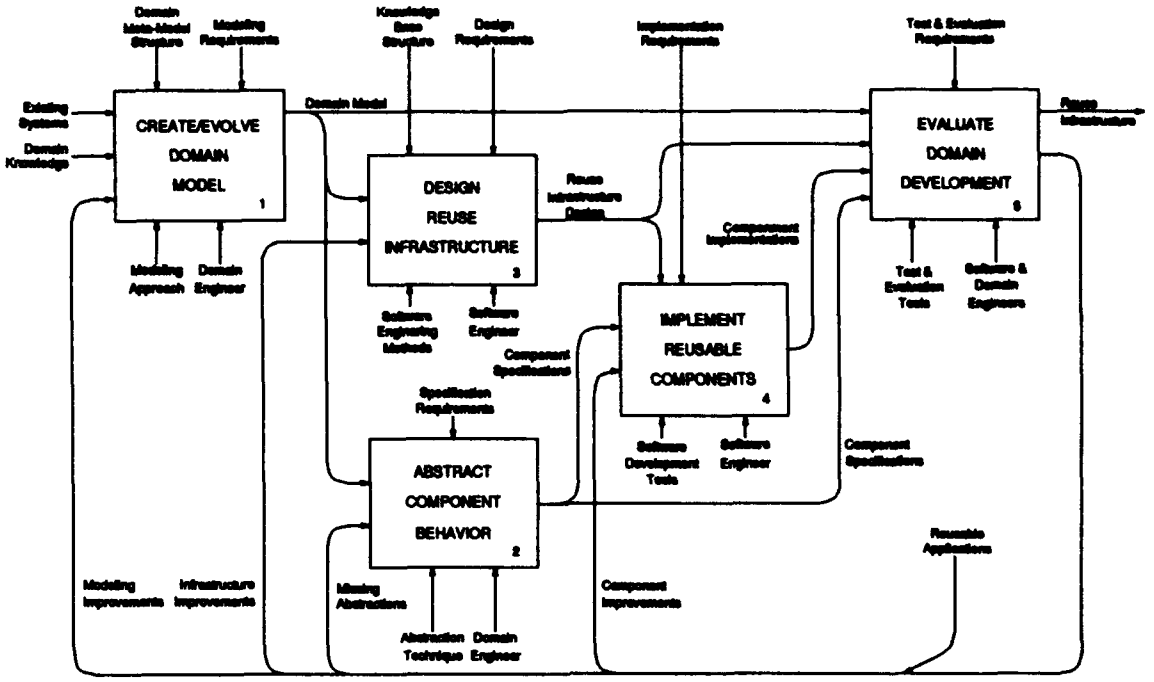


Figure 3.4 Knowledge Base Population

3.5.1 Create/Evolve Domain Model. The first step in populating the knowledge base of a DOACS with a particular domain is generating a domain model. A domain model is created during the initial domain modeling[3] activity and periodically updated through subsequent iterations. Before performing the initial iteration of this process, the methods and constraints must be defined, including the specific domain modeling approach and the formal modeling requirements.

The Domain Engineer should select the domain modeling approach based on the strengths and weaknesses of the approach and on the characteristics of the particular domain to be analyzed. We recognize that factors such as the Domain Engineer's familiarity with certain modeling approaches, management's overall goals, and the availability of resources will also play an important role in choosing a specific approach. Any well-defined object-oriented modeling approach should be sufficient since our methodology requires

[3]In keeping with our definitions presented in Chapter II, we considered domain modeling to be a type of domain analysis that results in a domain model with some formalized structure.

consistency and completeness[4] of the resulting domain model, but puts no restrictions on the form of this model (other than object-oriented). The Domain Engineer is the one who will have to use the approach to create and update the domain model. Therefore, the approach must be thoroughly understood. Also, the approach should take advantage of any available automated domain analysis/modeling tools to aid in capturing the domain. Such tools could improve performance and could aid in maintaining the domain model's consistency and completeness (information on one such tool can be found in Crowley's research (8)). We expect that these tools will be tied together after they (and DOACS) become more established (effectively automating much of the role of the Software Engineer).

The modeling requirements come from both the domain modeling approach chosen as well as the fact that domain analysis must be accomplished using object-oriented techniques. The minimum modeling requirement consists of a formal structure that encapsulates all the information represented in the domain model. The structure is similar to a meta-model structure (proposed by Iscoe (15)) and should be in some object-oriented form.

Once the domain modeling approach and modeling requirements have been defined, the Domain Engineer can begin creating the domain model. The domain model is transformed from a simple hierarchy of abstraction identifiers to a formal structure of captured domain semantics (which ideally leads to a domain-specific language). As the domain model evolves, the Software Engineer can begin applying the constraints of a knowledge base for a particular DOACS to construct a reuse infrastructure (i.e., instantiate the domain model). However, before discussing the domain implementation, we must address the task of defining individual component abstractions.

3.5.2 Abstract Component Behavior. After a structure has been created containing components in the domain, the behavior of the individual components must be defined. McCain refers to this process as "component domain analysis" (25:73). The abstraction

[4]By completeness, we do not mean that the domain model has to capture the whole domain, but rather that the parts of the domain that it does capture are completely captured within the scope of the domain analysis approach.

technique chosen by the Domain Engineer must be consistent with the modeling approach used in the Create/Evolve Domain Model activity (ideally, they both should be part of an integrated domain analysis approach). The results of the abstraction technique should be in a form that the Software Engineer understands and can implement. For example, it is possible to create *Z* schemas that cannot be implemented on a computer system. The distinct features of the domain under analysis should also play a role in which abstraction technique the Domain Engineer chooses. However, although different abstraction techniques may be used for different domains, using the same technique within a domain is required.

One of the most important aspects of the component abstraction is the interface specification. Although object interfaces could have been partially defined during the previous activity (if not completely defined, depending on the modeling approach chosen), each component abstraction must have interfaces that support consistent relationships between other component abstractions. The Domain Engineer must keep in mind that this whole process is based on an object-oriented methodology and that the final result will be objects that can be connected together. Incorrect or inconsistent component interfaces will cause severe problems when the Software Engineer tries to implement the abstractions and also when the Application Specialist attempts to compose applications.

Another aspect of each component abstraction is the effect of certain component attribute values. Attributes contain data that represents characteristics of an object (19:20). These characteristics may have a large impact on the behavior of a particular component. The Domain Engineer must identify these impacts during this activity. For example, a component that converts a real number to an integer may have an attribute that specifies whether that component will do the conversion by rounding to the nearest integer, truncating, or rounding up to the next integer. One useful feature of a DOACS is the ability for the Application Specialist to change these component attributes and "tailor" the component for a specific application. However, components can have attributes that should not be changed by the Application Specialist (for example, attributes that reflect the internal state of the component). These internal attributes should also be identified during this activity.

It is expected that the Domain Engineer may identify better ways to model the domain at this stage of the process. For example, at this point, the Domain Engineer may choose to combine objects or classes (generalization), or split an object or class (specialization). These improvements are acceptable (and even desirable). As stated before, our process is iterative. Component abstractions are evaluated (by the Evaluate Domain Development activity), and the Domain Engineer is notified of any problems. Our study of domain analysis revealed that the best domain model is the one that has evolved over time (i.e., it is difficult to come up with the best model on the first iteration).

We recognize that the Domain and Software Engineers may choose to merge this activity with the Implement Reusable Components activity by using a DOACS-specific method to define and implement individual component behaviors. This could save time and effort during the domain analysis, but it has its drawbacks. When system requirements and implementation requirements constrain the definition of individual component behaviors, biases could easily enter into the component definitions (e.g., implementation details often "muddy the water" of a "pure" domain analysis). Any bias introduced at this early stage may cause difficulties as the domain implementation evolves. Also, abstracting component behaviors in this way may make it difficult (or impossible) to reuse the abstracted components to populate another DOACS.

3.5.3 Design Reuse Infrastructure. The domain model is a structure (much like the syntax and semantics of a grammar) that captures the individual components, relationships between components, and other related information in an application domain. The Software Engineer must now organize the information captured in the domain model into a form that can be stored in the knowledge base of the particular DOACS. The organized abstractions become the domain's reuse infrastructure design. The infrastructure design is similar to Arango's reuse infrastructure specification that acts as "an architecture for reusable information"(3:82). In designing the reuse infrastructure, current software engineering methods should be used; however, these methods must fit into the object-oriented paradigm. Also, the Software Engineer must understand and follow any other design requirements (e.g., organization-specific design standards).

The implementation of this process is very system-specific; however, the results of this activity will usually form, as a minimum, an abstract syntax tree with each node representing some abstraction and each leaf representing a component implementation. More sophisticated DOACS will require more sophisticated results like implemented domain-specific languages, domain semantics (e.g., a domain rule that component A must follow component B), domain-specific software architectures, and methods for defining how the components are presented to the Application Specialist during the composition process.

3.5.4 Implement Reusable Components. Now that the reuse infrastructure has been designed and the component behaviors have been defined, the individual components must be implemented (transformed into the required "executable" form). Component implementation is a translation of each component from its abstract definition into a form representable in the knowledge base and executable by the DOACS.

The Software Engineer implements each abstraction organized in the domain infrastructure design. The implemented abstractions become part of the reuse infrastructure implementation and, when all of them are done, domain implementation is complete. Before implementing the components, the specific software development tools and the implementation requirements must be identified.

The selection of software development tools will depend on the software implementation languages accepted by the DOACS. The Software Engineer should use whatever tools are available to implement the components. Commercial compilers and software development environments can provide the necessary utilities to perform reuse infrastructure implementation.

The implementation requirements constraint indicates the constraints imposed by the DOACS, most of which result from the requirements of its knowledge base. These constraints must include a formal description of the software architectures supported by the DOACS and the specific procedures for representing component implementations within the knowledge base. The implementation requirements should also identify different methods to model the component abstractions. For example, a stack data type can be implemented as an array or list with various advantages to each of these implementations. Part

of the implementation requirements should detail the procedures for selecting the desired implementation method for different abstractions.

A priority list of the order to implement the components may also be included in the implementation requirements. The Application Specialists may require some components immediately, while other components may not be needed until well in the future. Using priorities, the Software Engineer could implement components in order of their importance to the Application Specialist. This allows the DOACS to support application composition before complete domain implementation is finished.

3.5.5 Evaluate Domain Development. In this activity, the Software and Domain Engineers evaluate the outputs of the previous activities. This activity covers a broader scope than the others because, while the others are focused on one area of domain analysis/implementation, this activity has relevance to the entire knowledge base process from start to finish.

The results of each of the other four activities should be evaluated as they are generated, both individually and along with results from previously completed activities. Also, when all the results are completed, they should again be evaluated to ensure consistency and completeness. This activity should trigger one or more of the previous activities when errors or better ways to model/implement the domain are discovered. Each activity could include their own evaluations as part of their normal execution, but we have made the evaluation a separate activity to provide a way to evaluate the results of all the activities in reference to one another.

Although we have placed this activity at the end of our process, it is involved at each step of the knowledge base population process. As the domain model and knowledge base evolve, this activity becomes more important in maintaining the integrity of the applications they generate.

An input to this activity that is not specifically shown on Figure 3.4, but is worthy of mention, is from the Application Specialist. As the Application Specialist composes applications, several problems may be identified (such as missing components, errors in components, or problems with component interfaces). The Domain Engineer then evaluates

the problems and makes any necessary changes to the domain model or reuse infrastructure. Another input from the Application Specialist is the identification of applications that should be incorporated into the knowledge base as primitive components. These reusable applications are discussed in the next section.

3.5.6 Reusable Applications. Along with the components identified during the domain analysis, the Application Specialist may want to use an existing application (or part of one) as a component in a new application. We call this reused application a **reusable application** (although all applications are reused in the sense that they can be reloaded and modified). When the Application Specialist is choosing components to use in composing an application, these reusable applications should be included in the list of choices. It is possible that a DOACS could implement reusable applications without any special processing; however, in general, we expect that these applications will need to be identified for reuse and somehow "processed" by the Domain and Software Engineers. This processing is completely dependent on the specific DOACS, and there could even be different methods for accomplishing this within the same DOACS.

The creation of a reusable application may be simple or complex, depending on the specific situation and the capabilities of the system. Primitive application creation could consist of stripping off desired data "sources" and "sinks" (components that generate or consume data that we now want to leave off the application) and identifying the interface to the reusable application that remains after these components are removed. If the specific DOACS does not have the capability of storing and using a reusable application like this, then the Software Engineer may have to somehow combine all the behaviors of the components of this application into this new component in the same manner as those identified in the domain analysis. Also, depending on the specific DOACS, it may be required that this new primitive application be added into the reuse infrastructure design.

Along with making the reusable application available as a new component to the Application Specialist, the Domain Engineer should consider adding it to the domain model (in the Create/Evolve Domain Model activity). If the reusable application is added, then its behavior should be captured in the Abstract Component Behavior activity and

consideration given to implementing it as a single component (rather than a collection of components).

3.6 General Process Support and Constraints

There are many reasons for having a general knowledge base population process. One reason is to provide a framework to use when developing tools and utilities that apply to populating many knowledge bases. Is there justification for developing a general process? How "general" is this process? This section is our attempt to support our development of a "general" knowledge base population process.

We have illustrated a possible knowledge base population process. We attempted to clarify the process by identifying the methods and constraints for each activity; they change depending on the specific application domain and the specific DOACS used to develop applications. This versatility, tying each of these traits to the characteristics of the domain or system, adds justification to the general applicability of the process. Although the domain model resulting from the domain analysis is independent of the knowledge base constraints, the model's structure and the domain analysis method used to create it can differ for each domain. This characteristic is captured in the first stage of our process. The same idea applies to building the reuse infrastructure. The infrastructure can be different for each DOACS. We do not suggest that this process will support every application domain or apply to all DOACSs (it has not even been shown that every domain can be modeled using current modeling techniques). However, we do suggest that this process will support many application domains and many systems.

There are some characteristics that must exist in the domain and the system. The system's knowledge base must have the capability to represent an organized collection of reusable software components and the rules for their composition. We described this framework in Section 3.2. The domain must be somewhat established (i.e., some structure must exist for building applications in the domain, either informal or formal). This implies that the domain is mature enough to have existing applications that could provide important information during a domain analysis.

We have developed our process with the goal of future automation. Constraints placed on each activity (such as knowing the particular software architecture) have been included only where necessary. However, software engineers may be concerned about changes made to these constraints while a domain model and knowledge base are evolving. For example, suppose we modify our knowledge base structure after already implementing several domains. What impact does this have on the knowledge base population process? Will we have to respecify and re-implement the domain model? Are we forced to stick to the original constraints? Although a more general knowledge base population process may exist, we feel that our process can handle these problems if the proper attention is given to the definition of each constraint.

3.7 Summary

A formal process for populating a knowledge base results when we apply the outline described in this chapter to a candidate DOACS. The process that we have developed incrementally evolves the domain – both the domain model and the reuse infrastructure in the knowledge base. The process explicitly captures the role of feedback through the evaluation of each activity. When the process is first started for some application domain, most of the effort will involve the Create/Evolve Domain Model and Abstract Component Behavior activities (activities one and two of Figure 3.4). As the process continues, effort will increase in the Design Reuse Infrastructure and Implement Reusable Components activities (activities three and four of Figure 3.4). The attention given to the Evaluating Domain Development activity will depend on the particular requirements for testing (which usually relates the size and complexity of the domain).

Although we have developed a general process, software engineers need to conduct more application-oriented research into knowledge base population to gain more experience and develop tools to assist in evolving domain models and structuring knowledge bases. The DOACS concept has not yet been proven in the software engineering community. Our research efforts and the efforts of other researchers will help to improve these software development technologies.

IV. APTAS Knowledge Base Population Process

4.1 Introduction

This chapter develops a population process for the APTAS system. Chapter II introduced APTAS as a prototype application composition system developed for the Air Force to assist radar tracking engineers in the design and testing of trackers (radar tracking algorithms). Therefore, the task is to design a process to populate the knowledge base of APTAS with additional tracking information.

Before formalizing a knowledge base population process, one must understand how an Application Specialist (called the **Tracking Engineer**) uses APTAS to compose trackers. Section 4.2 describes this composition process. One should also understand the types of information stored in the knowledge base. Although this information could conceivably belong in an appendix, understanding the knowledge base was crucial to developing the population process. Section 4.3 describes the knowledge base.

Section 4.4 describes the formal knowledge base population process that was developed using an instantiation strategy. The instantiation strategy is based on the general process introduced in Chapter III and uses the descriptions of APTAS and its knowledge base.

Section 4.6 presents some conclusions about the formal knowledge base population process. Several of these conclusions are demonstrated in Chapter V using the existing contents of the knowledge base.

4.2 Application Composition in APTAS

Chapter II introduced the APTAS system and briefly described its application composition process. This section thoroughly explains each stage of the application composition process. Figure 4.1 shows a data flow diagram capturing the composition process.

The composition process begins when the **Tracking Engineer** (the APTAS equivalent of the Application Specialist) responds to questions presented by the **Forms Generator and Display** (20:1). When the engineer completes the forms, the **Architecture**

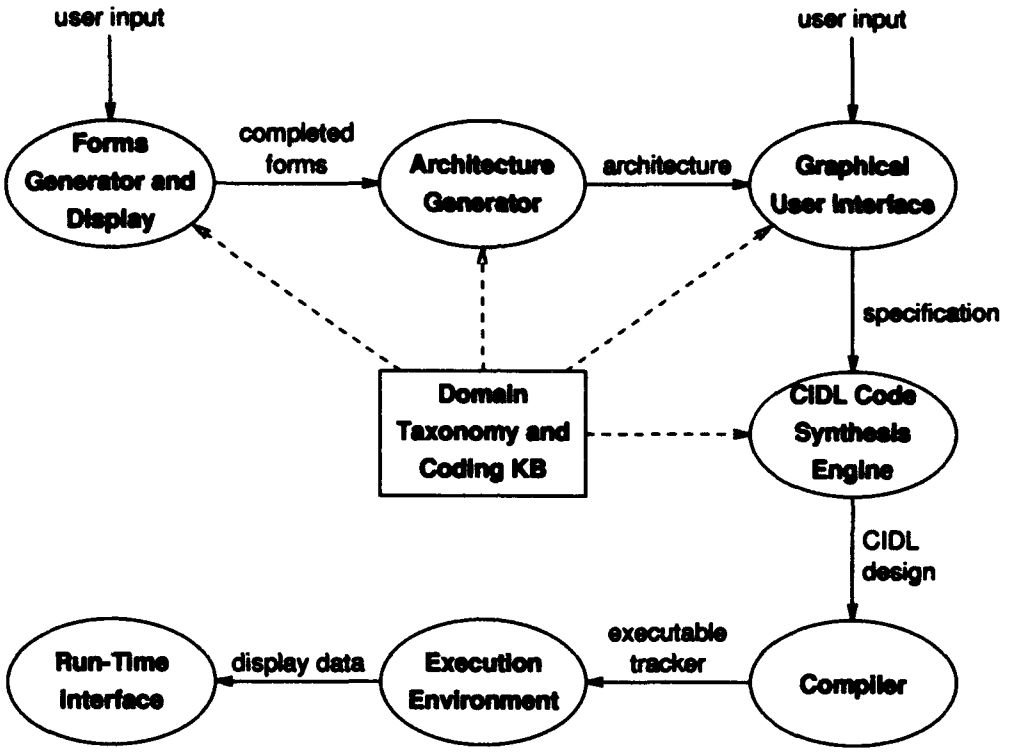


Figure 4.1 The APTAS Composition Process Data Flow Diagram (adapted from (17:3)).

Generator takes the specified tracker and displays a derived tracker architecture to the **Graphical User Interface**. The engineer then updates the architecture and passes the completed tracker specification to the **CIDL Code Synthesis Engine**. The synthesis engine transforms the specification into a CIDL program that represents the application design. Data files result from compiling and executing the tracker design. Test runs on the **Run-Time Interface** simulate the tracker design.

There are several stages of the composition process that use information from the knowledge base. These stages are the Forms Generator and Display, the Architecture Generator, the Graphical User Interface, and the CIDL Code Synthesis Engine. Each stage uses specific categories of information within the knowledge base. Because of the importance of the knowledge base, a separate section describes each information category. The rest of this section explains each composition stage in more detail.

4.2.1 Forms Generator and Display. The Forms Generator and Display stage consists of a forms generator and an interface, referred to as the **Dynamic Forms Interface**. The forms generator, using information from the knowledge base, builds a set of question forms and manages their presentation to the Tracking Engineer. The interface lists the available forms and displays them to the engineer.

A form is either active (available for selection) or not active. The engineer selects a form from the taxonomy summary (list of active forms). The selected form displays on the interface as a set of questions and their associated responses.

Questions take one of four responses. **Numeric** questions require a number (integer or real) within a specified range. **Text** questions require a string of characters within a specified length. **Exclusive choice** questions require a single selection from a list of responses. **Checklist** questions require the engineer to select a subset of the available responses. Each question response executes a set of associated actions.

Some actions save values to component parameters or set special variables. Other actions make forms active or not active. The engineer continues to respond to active forms until completed. The completed forms specify a hierarchical tracker architecture generated by the Architecture Generator.

4.2.2 Architecture Generator. The Architecture Generator builds tracker architectures. An architecture is a structure of component modules, submodules, and their relational interfaces. The Tracking Engineer specifies an architecture (or module structure) through a sequence of question responses. Many of the engineer's responses select component modules and set parameters that constrain (specify) the architecture.

The Architecture Generator transforms the specifications into a corresponding architecture using rules from the knowledge base. Each rule defines a specific module. Within the rule is a set of submodules that may compose the module along with the conditions that must be true for their inclusion in the architecture. Rules also include declarations of external interfaces available to other modules and information defining relations between the submodules. By applying the rules, the generator derives the remaining tracker architecture and passes an architecture description file to the Graphical User Interface.

4.2.3 Graphical User Interface. The Graphical User Interface presents the Tracking Engineer with a visual display of the tracker architecture. The display appears as a set of interconnected icons. Specific icons represent modules, interfaces, and parameters. The connections represent relationships between modules. Information from the knowledge base defines the visual representation of each icon and its relationships. Section B.1.3 describes the various icons used by the APTAS system.

The engineer updates a tracker specification by editing the architecture displayed on the canvas (interface window showing components). A canvas displays the structure of a particular module. For instance, the canvas of a primitive module only displays parameters and module interfaces. The user interface provides various operations to update the tracker specification. Operations allow the engineer to add new modules and relationships, access descriptive information from the knowledge base, change parameter values, and modify relationships (i.e., change data type transferred between modules). At any point during the update, the engineer can pass the revised tracker specification to the synthesis engine.

4.2.4 CIDL Code Synthesis Engine. The **CIDL Code Synthesis Engine** generates a CIDL representation of a tracker architecture. This representation corresponds to the tracker application specified and designed in previous composition stages by the Tracking Engineer. The engineer simulates the specified behavior using this design and can translate it into standard programming languages (e.g., Ada).

The synthesis engine performs a rule-based transformation of the specification using rules from the knowledge base. There are two kinds of rules: generic and domain-specific. Generic rules map modules, interfaces, relationships, and parameters identified in the specification to predefined structures in CIDL. These rules are part of the synthesis engine itself. The engine applies generic rules to the specification before applying any domain-specific rules.

Domain-specific rules act on combinations of parameter values for a particular module. These rules specialize a module's CIDL structure generated by the generic rules by substituting new CIDL structures supplied by the domain-specific rules stored in the

knowledge base. Substitutions continue until no more rules apply. The resulting collection of CIDL structures corresponds to the tracker design.

4.2.5 Compiler, Execution Environment, and Run-Time Interface. After synthesizing the tracker design, the system provides the Tracking Engineer with several capabilities. The engineer might decide to translate the design into an equivalent Ada implementation through the system's **CIDL to ADA Translator**. APTAS provides an Ada Compiler and Execution Environment. It also provides a Lisp Compiler and Execution Environment that works directly with the CIDL design. The engineer can test an executable tracker (Ada or Lisp equivalent) using the system's Run-Time Interface. This interface simulates the tracker behavior using data generated in either execution environment and simulates the tracker behavior from the engineer's specifications. If the tracker behavior does not meet the intended specifications, the engineer might return to a previous stage in the composition process, update the specification, synthesize a new design, and simulate the new behavior.

The Ada Compiler and Execution Environment is an off-the-shelf product and does not require any information from the knowledge base. The Lisp Compiler and Execution Environment does not require any information from the knowledge base either. The Run-Time Interface does require additional information that an APTAS user generates in the Graphical User Interface but does not require specific information from the knowledge base.

4.3 APTAS Knowledge Base

The knowledge base, introduced in Chapter II as the Domain Taxonomy and Coding Knowledge Base, "maintains all of the data pertaining to an application domain which is used to support the specification and synthesis of new code" (17:3). Figure 4.1 identified four stages in tracker composition that use information from the knowledge base. The knowledge base provides APTAS with several different categories of information (e.g., reusable CIDL implementations of primitive modules, rules generating architecture representations, questions eliciting specification information).

The information in the knowledge base for a particular domain is called a reuse infrastructure. APTAS organizes each reuse infrastructure into seven categories of information, including the Specification Taxonomy (question forms), Architecture Construction Rules, Display Conventions, Type Descriptions, Library Components Structure, Library Modules, and Coding Rules. APTAS calls these seven categories a Domain Taxonomy and Coding Knowledge Base.

Each category of the Domain Taxonomy and Coding Knowledge Base contains unique information (i.e., it provides some information about the domain not found in the other categories). Each category also contains information found in other categories. For instance, the variables included in the Architecture Construction Rules category are set by actions within the Specification Taxonomy category.

Most changes to the knowledge base affect many categories. Figure 4.2 identifies information in each category that is dependent on information appearing in other categories. The desired source of information is indicated using arrows to point away from the originating category. Notice that most information originates from the Architecture Construction Rules category. For instance, suppose a new module is added to the Architecture Construction Rules. The Specification Taxonomy may require a new question form to access the new module. The Display Conventions may require a new icon definition for the module. The Type Descriptions may require additional documentation on the module. Other categories may also require additional information. The Knowledge Engineer must remember to use the same information throughout the knowledge base categories, or the APTAS system may not correctly support the new information (the system does not make any consistency checks). By identifying the dependencies between information categories, some critical relationships needed to formalize a knowledge base population process were uncovered for the APTAS system.

Each domain has its own directory structure under a main directory. For instance, the radar tracking domain is stored in the `TRACKER_DOMAIN` directory. This directory holds the reuse infrastructure (Domain Taxonomy and Coding Knowledge Base) and any applications composed within the domain. The rest of this section describes the information contained in each category and identifies several special knowledge base files.

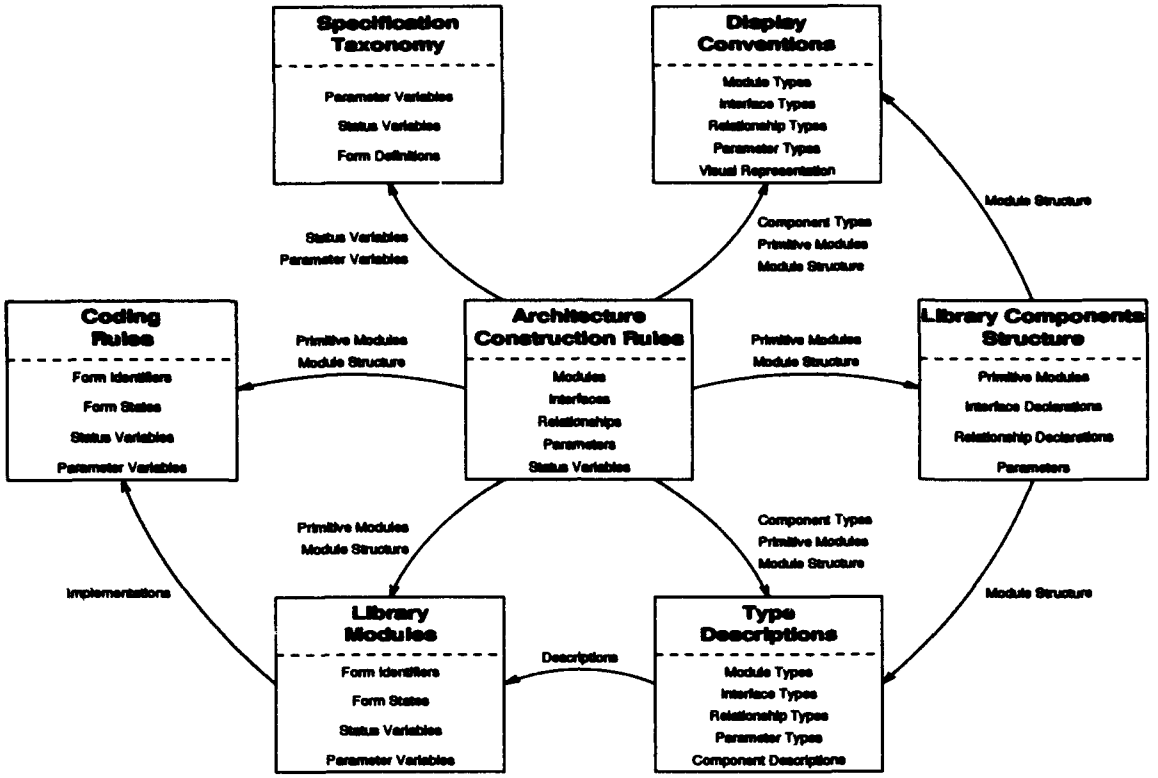


Figure 4.2 Association of Information Between Knowledge Base Categories.

4.3.1 Specification Taxonomy. The Specification Taxonomy category provides the Forms Generator and Display with a hierarchy of question forms. These forms contain sets of questions and related responses that capture some fundamental domain knowledge. As described in Section 4.2.1, the APTAS system elicits specification information from the Tracking Engineer through these forms and uses the set of responses to generate an application architecture.

This category describes a forms hierarchy similar to an abstract syntax tree with each node representing a specific form. The knowledge base refers to the forms as **levels**. Each level has a unique identifier (very similar to a pathname). The first form in the category is considered the root. Initially, one might think the forms hierarchy corresponds to the module hierarchy, because the level identifiers are often identical to the module names. The level identifiers, however, do not *have* to relate to a particular module. They define a separate hierarchy that may not correspond to the module hierarchy. A level identifier begins a form definition. The form definition shown in Figure 4.3 begins with the level

```
/TRACKING/SINGLE_CSO
"Single and_CSO Target Characteristics" FALSE

    "Target Type"
    STACK
        "Known Target Type 1"
            VARIABLE_SET(SINGLE_TARGET_MIN_VEL, 0.0)
            VARIABLE_SET(SINGLE_TARGET_MAX_VEL, 0.0)
            VARIABLE_SET(SINGLE_TARGET_MAX_ACC, 0.0)
            VARIABLE_SET(SINGLE_TARGET_MIN_ALT, 0.0)
            VARIABLE_SET(SINGLE_TARGET_MAX_ALT, 0.0)
            VARIABLE_SET(SINGLE_TARGET_MANEUV_ADT, "true")
            VARIABLE_SET(SINGLE_TARGET_MANEUV_DURATION, "prolonged")
            VARIABLE_SET(SINGLE_TARGET_RMS_MANEUV_LEVEL, 0.1)
        "Other"
            ACTIVATE_LEVEL(/TRACKING/SINGLE_CSO/TARGET_CHARS)
    ;

    "Target Density"
    NUMERIC [0.0, 10.0]
            [0.0, 10.0] SAVE_VALUE(TARGET_DENSITY)
END;
```

Figure 4.3 Sample Form Definition

identifier /TRACKING/SINGLE_CSO. The string appearing after the level identifier defines the title that displays when the form displays during the composition process. The title of this form is `Single and_CSO Target Characteristics`.

The forms hierarchy may contain many forms. During composition (as noted earlier), each form is either active (available for display) or not active. The boolean value following the form title defines the initial state of the form (FALSE being not active). The state of the form changes as the engineer responds to questions from active forms. Each response may activate any of the descendant forms (forms lower in hierarchy). However, all of the ancestor forms (forms higher in hierarchy) must also be activated for the form to become available. The END keyword marks the end of a form definition.

Each form may include a set of questions that elicit specification information about the tracker. Questions begin with a string representing the specification information it acquires (`Target Type` and `Target Density` in the example).

A condition may optionally precede the question string. Conditions define when the engineer can make a response to the question. Conditions are expressions (including

conjunctions and disjunctions) that evaluate to a boolean. The grammar in Figure 4.4 defines the structure for conditional expressions.

```
CONDITION ::= CONJUNCT | CONJUNCT "|" CONDITION
CONJUNCT  ::= EXPR | EXPR "&" CONJUNCT
EXPR      ::= TERM RELOP TERM
TERM      ::= Identifier | Number | StringLiteral
RELOP     ::= "EQ" | "NE" | "LT" | "GT" | "LE" | "GE"
```

Figure 4.4 Grammar for Conditional Expressions

The question type follows the question string. Questions may be one of four types. NUMERIC questions elicit a number (integer or real) within a specific range. The range follows the type declaration and consists of a minimum and maximum value separated by a comma and enclosed in square brackets (`[0.0, 10.0]`). TEXT questions elicit a string of characters within a specified length. The length follows the type declaration and must be a positive integer. STACK (or exclusive choice) questions elicit a single selection from a list of responses. CHECKLIST questions elicit the selection of a subset of the available responses.

STACK and CHECKLIST questions have a set of valid responses that follows the type declaration. Following each response is a set of actions to take when the response is selected by the engineer. NUMERIC questions have a set of valid subranges that follow the valid range. APTAS takes the set of actions following the subrange whenever the engineer's response is within the subrange. TEXT questions have a set of actions that follow the maximum length and execute whenever the engineer responds to the question.

There are only four types of actions defined for the taxonomy. SAVE_VALUE stores the engineer's response to the variable name passed as a parameter. VARIABLE_SET takes a variable name and some value (constant or identifier) and sets the variable to the value. ACTIVATE_LEVEL takes a level identifier and changes the state of its associated form to active. DEACTIVATE_LEVEL takes a level identifier and changes the state of its associated form to not active.

The knowledge base stores the forms defined by the Specification Taxonomy in a domain's `global.form` file. Form definitions begin after the LEVELS keyword, and a semicolon

separates each definition. A complete grammar for taxonomy files appears in the Software User's Manual for APTAS (20:38-39).

4.3.2 Architecture Construction Rules. The Architecture Construction Rules category provides the Architecture Generator with an organization of the modules available from a domain. The structure for each module is captured by a rule. The collection of rules defines a module organization similar to an abstract syntax tree; leaves represent primitive modules (modules with CIDL implementations) and nodes represent modules with submodules (i.e., modules composed of other modules). Primitive modules cannot have rules defining their structure (see Section 4.3.5). Modules that are not primitives must have their structure defined by a particular rule.

A rule begins with the module type. The module type, which must be unique, is followed by the definition of the module type's structure. The structure consists of an optional set of submodule declarations, followed by an optional set of interface declarations, and ending with an optional set of relationship declarations. The rule shown in Figure 4.5 defines the structure of a `TRACKER_ENVIRONMENT` module type. The `TRACKER_ENVIRONMENT` module type has four submodule declarations and three relationship declarations. There are no interface declarations in the structure.

Submodule declarations may begin with a conditional expression (see Figure 4.4) that must evaluate to true for the submodule to be included in the architecture. Conditions could imply a choice between alternate submodules or could imply an optional submodule. Conditional expressions are not required and a submodule is always included when no condition begins its declaration. The first submodule has no conditional expression and is, therefore, always included within a `TRACKER_ENVIRONMENT` module. Following the optional condition is the submodule's name and its type. The first submodule is named `Sensor_Data` and is a `SENSOR_MODEL` type. If the submodule type is a primitive module type (i.e., there is no rule that corresponds to the module type), then the declaration may conclude with a set of parameter assignments (begun with a {, separated by semicolons, and ended with a }. Otherwise, the module type has its own rule and the submodule declaration does not have a set of parameter assignments. The `SENSOR_MODEL` is a primitive module

```
"TRACKER_ENVIRONMENT"
      "Sensor_Data"              "SENSOR_MODEL"
         {"ITERATIONS" = TEST_ITERATIONS;
          "PERTURBATION_FACTOR" = 0.02;
          "TARGET_SPECS" = ~"[ <COMPUTE_FUN := ~DEFAULT_EQ1 ,
                               ARGS := [0.089, 0.009, 1.78,
                                        1.16, 92.31, 8.23]>,
                               <COMPUTE_FUN := ~DEFAULT_EQ2 ,
                               ARGS := [0.191, 0.083, 2.46,
                                        2.09, 81.01, -7.23]> ]";
          "SENSOR" = ~"~SENSOR_TYPE"}
  default_tracker EQ "true"
      "Tracker"                  "TARGET_TRACKER"
  default_tracker NE "true"
      "Tracker"                  "NEW_TRACKER"

      "Output"                   "OUTPUT_DISPLAY"
         {"TABLE_DATA_FILE" =
           "data_files/default_tracker_table_data.txt";
          "MAP_DATA_FILE" =
           "data_files/default_tracker_map_data.txt";
          "ITERATIONS" = TEST_ITERATIONS}

  REL("Sensor_Scan_Frame_To_Tracker", "Async",
      "Sensor_Data.scan_frame_out", "Tracker.scan_frame_in",
      ~"GenericScanFrame")
  REL("Operator_Query_to_Tracker", "Async",
      "Output.query_out", "Tracker.user_query_in",
      ~"GenericQuery")
  REL("Tracker_Data_to_Display", "Async",
      "Tracker.display_data_out", "Output.reply_in",
      ~"GenericTrackData")
;
```

Figure 4.5 Sample Construction Rule Definition

type with four parameters assigned in this submodule declaration. The second submodule declarations has a conditional expression and declares a submodule named `Tracker` of `TARGET_TRACKER` type (which has a rule defining its structure). The third submodule also has a conditional expression and declares a submodule named `Tracker` of `NEW_TRACKER` type. Only one submodule with the name `Tracker` may be included in the architecture and the conditional expressions ensure that only one submodule declaration evaluates (i.e., only one of the two conditions can be true). The fourth submodule declaration declares a submodule named `Output` of type `OUTPUT_DISPLAY` that is a primitive and assigns three parameters.

Interface declarations may follow submodule declarations, however, the rule in Figure 4.5 does not have any. Interface declarations begin with the keyword DECL. The interface includes three items (separated by commas and enclosed in parentheses). The first item declares the name of the interface. The second item declares the type of the interface. An interface is either an in_port (incoming data) or an out_port (outgoing data). The last item contains the initial value of the interface. An example interface declaration appears below:

```
DECL("Display_Data_Out", "Out_Port", "")
```

The module with this interface declaration includes an out_port named Display_Data_Out. Interfaces are used to pass information between modules. Interfaces are connected together (or to other modules) using relationships. APTAS defines six types of relationships. Async relationships occur between module interfaces and represent asynchronous communication where messages are buffered and modules do not wait for corresponding sends or receives. Sync relationships also occur between module interfaces and represent synchronous communication where modules wait for corresponding sends or receives. Alias relationships occur between the interfaces of a module and submodule where data is simply forwarded between the two. A parameter_module relationship occurs between an entire module and its submodule and represents the access to a submodules functions. Apply_function relationships occur between module in_ports and module (or submodule) functions and represent the execution of a function whenever data is received at the interface. Forward_function_result relationships occur between module out_ports and module (or submodule) functions and represent the transfer of a function's result to a port.

Relationship declarations appear at the end of the module structure and complete an Architecture Construction Rule. Relationships begin with the keyword REL and end with a semicolon. These declarations include five items (also separated by commas and enclosed in parenthesis). The first item declares the name of the relationship. The second item is the relationship type. The third item declares the origin of the relationship (relationships are one-way). The fourth item declares the target of the relationship. The last item is the

data type passed or expected by the relationship (**parameter_module** relations do not pass or expect data). The example rule (in Figure 4.5) has three **async** relationships.

There may be any number of relationships within each construction rule, and there may be many rules. However, the first construction rule is different from all the rest of the rules. The Architecture Generator *always* applies the first construction rule. This rule defines the application environment and determines the initial applicability of the remaining rules. Therefore, the sequence of the construction rules is very important. A rule defining the structure of a submodule should follow all the rules that include the submodule within their structure definition.

APTAS defines an abstract architecture description (as described in Section 4.2.3) by transforming a partial tracker specification through the application of these rules. The knowledge base stores the Architecture Construction Rules in the **global.form** file. The collection of rules begins with the keyword **MODULES** after the last form definition in the taxonomy. Semicolons separate each rule. The rules share the taxonomy grammar defined in the Software User's Manual for APTAS (20:38-39).

4.3.3 Display Conventions. The Display Conventions category provides the Graphical User Interface with visual representations (or icon rules) for modules, parameters, relationships, and interfaces. Icon rules take three forms: display rules, relation drivers, and constraint rules. A single display rule can define all component (module, parameter, and interface) representations, or different display rules can define specific component representations. Relation drivers define conventions for specific relationships. Constraint rules define a relationship's multiplicity (minimum and maximum number) with regard to a particular component. Section B.1.3 lists the visual standards for component icons and relationships adopted by APTAS for tracker applications.

Display rules define an icon (shape and color) to represent a component. The rule begins with some component identifier followed by an arrow (**->**). This identifier might begin the definition for an entire component class (e.g., **MODULE** or **IN_PORT**), or it might begin the definition for a specific module type. Figure 4.6 shows the display rule for a **SENSOR_MODEL**. The rest of the rule describes the visual representation by making assignments to graphical

```
SENSOR_MODEL -> ICON = MED_BLUE_RECT
          LABEL = TOP BOTH
          DEFAULT_POSITION = CENTER
          CONTENTS =
                MODULE -> ICON = SM_RECT
                          -- LABEL = CENTER NAME
                          DEFAULT_POSITION = CENTER
                IN_PORT -> ICON = MINI_CIRCLE
                           LABEL = LEFT NAME
                           DEFAULT_POSITION = LEFT
                OUT_PORT -> ICON = MINI_CIRCLE
                            LABEL = RIGHT NAME
                            DEFAULT_POSITION = RIGHT
                DISPLAY RELATIONS
```

Figure 4.6 Sample Display Rule

attributes. ICON is a required assignment that holds the name of some predefined icon files. The sample rule selects a MED_BLUE_RECT (medium-sized, blue rectangle) as the icon. LABEL is an optional assignment that declares the location of the icon's label and the contents of the label. The label of this icon appears at the TOP and consists of BOTH the component instance name and the component type. DEFAULT_POSITION is a required assignment that declares the desired location of the icon when displayed. This icon would appear in the CENTER of its parent module's display window (canvas). The last attribute, CONTENTS, describes the internal component structure of the module that appears within the icon and is also an optional assignment. In the example, a submodule (MODULE type) appears as a SM_RECT (small, black rectangle) and an IN_PORT interface appears as a MINI_CIRCLE (small, black circle). The final line of a display rule must declare whether the internal relationships appear (DISPLAY RELATIONS) or do not appear (HIDE RELATIONS).

Relation drivers define a line (solid, dashed, etc.) to represent a relationship. The driver begins with some relationship type (see Section 4.3.2) followed by a colon and the keyword RELATION. Figure 4.7 shows the driver for PARAMETER_MODULE relationships. The rest of the driver definition describes the visual representation for the relationship type defining the attributes of the line. LABEL is an optional attribute that operates similar to display rule labels. WIDTH is a required attribute that defines the width of the line. This sample driver defines a line of width 0. COLOR is an optional attribute that defines the color of the line (Orange Red in this example). If a relationship appears as some type of dashed

```
PARAMETER_MODULE : RELATION
        WIDTH = 0
        COLOR = "Orange Red"
        DASH_COUNT = 4
        DASH_LIST = 10 3 3 3
        FROM_END = PLAIN
        TO_END = ARROW
        VALID_PAIRS = (MODULE, MODULE)
```

Figure 4.7 Sample Relation Driver

line, two attributes describe the specific dash pattern. The DASH_COUNT attribute identifies the number of dashes and blanks (should be an even number), and the DASH_LIST attribute defines the length of each dash and each blank identified. The pattern is repeated over the length of the relationship line. FROM_END and TO_END are required attributes that define the appearance of the two ends of a relationship line. This example starts as a PLAIN line (no arrow) and ends with an ARROW. The term EXTRA_ARG may complete a relationship driver. However, the purpose of this term is not documented in the APTAS literature and its purpose could not be determined through conversations with the APTAS developers or through experimentation.

Constraint rules may define the multiplicity for some specific relationship type. Constraints begin with a number range ([#,#] or [#,+]) that might define the minimum and maximum number of a particular relationship (+ being no maximum). The number range is followed by the keyword RELATION and a relationship type (or possibly a relationship name). The constraint ends with a FROM, TO, or EITHER that probably constrains the relationship multiplicity in the specified direction. There are no constraint rules in the current reuse infrastructure, and the purpose of these rules is not documented in the APTAS literature. Their intended purpose could not be determined through conversations with the APTAS developers, but the constraint rules probably limit the number of relationships (by type) that may begin from (or end at) a particular module or interface.

The knowledge base stores the Display Conventions in the `global.gsdl-t` file. The conventions begin with the keyword TABLE followed by a domain identifier and end with the keyword END. The conventions have a special grammar defined in the Software User's Manual for APTAS (20:40-42).

4.3.4 Type Descriptions. The Type Descriptions category provides textual descriptions for components and relationships. The descriptions can include the function of each component, the purpose of parameters, the kinds of interfaces, and any other information that might describe the component. The descriptions can also include definitions for relationship types and any special data types available. Note that type descriptions are used solely as documentation and are entirely optional. However, every description is available to the Tracking Engineer (the APTAS Application Specialist) within the Graphical User Interface and can prove very valuable to less experienced engineers.

Each description begins with the name of the type being described, followed by a colon. An at-sign (@) signifies the end of a description. Figure 4.8 shows a description for the OUTPUT_DISPLAY module.

```
OUTPUT_DISPLAY:
This module is responsible for issuing run-time
requests for data to the database, including
any formatting of user inputs that may be
necessary, and displaying the data to the user.

PARAMETERS TO SPECIFY:
  ITERATIONS : int
  (used to determine how many sets of data
   it needs to process for the display;
   it should equal the value of the
   iterations parameter in the sensor_model)
  TABLE_DATA_FILE : string
  MAP_DATA_FILE : string
  (These are the names of the data files which
   will be created when the tracker is run.
   Use these data files as input to the
   run-time interface)
@
```

Figure 4.8 Sample Type Descriptions

The knowledge base stores the Type Descriptions in the `global.desc` file. The file begins with the first description and the special end-of-sequence symbol (`.:`) marks the end of all type descriptions. There is no special grammar defined for descriptions.

4.3.5 Library Components Structure. The Library Components Structure category provides the Graphical User Interface with parameter declarations, interface declarations,

and relationship declarations for library modules (primitives). These declarations map to specific icons (from the Display Components). The canvas for a selected primitive displays the icons for each declaration in the component's structure.

A component structure begins with the name of a module type, followed by a colon. The keyword **MODULE** follows the colon and precedes an equal sign. The component structure definition ends with the keyword **END**. Figure 4.9 shows the structure for a **TRACK_DATABASE_MANAGER** module. The **DECLARE** keyword identifies a list of any pa-

```
TRACK_DATABASE_MANAGER : MODULE =
DECLARE
        PLATFORM : PLATFORMTYPE;
        TRACK_BUFFER_SIZE : INT;
        TRACK_HISTORY_SIZE : INT;
        PLATFORM_POS_BUFFER_SIZE : INT;
        MISSION_BUFFER_SIZE : INT;
        REQUIRED_APPLICATION_MEMORY : INT;
        RAM_MEMORY : INT
        DM : DATABASE_MANAGER
                {MISSION_BUFFER_SIZE = 5,
                 PLATFORM_POS_BUFFER_SIZE = 8,
                 TRACK_HISTORY_SIZE = 5,
                 TRACK_BUFFER_SIZE = 2,
                 PLATFORM = AIRBORNE};
        BSB : BACKING_STORE_BUFFERING
RELATION
        BSB_TDB_IN_ASYNC : ASYNC(DM.TRACK_OUT, BSB.TRACK_IN) = STRING;
        DM_TDB_IN_ASYNC : ASYNC(BSB.DATA_OUT, DM.DATA_IN) = STRING
END
```

Figure 4.9 Sample Library Component Structure

rameter declarations and interface declarations, separated by semicolons. The **RELATION** keyword identifies a list of any relationship declarations, also separated by semicolons. This example structure has nine parameters, no interfaces, and two relationships.

Parameter declarations consist of a parameter name, followed by a colon and the parameter's data type. Data types may be any standard CIDL type (29) or a global data type defined in the Library Modules. A parameter initialization (value assignment) may follow the parameter type. If the parameter is a submodule (module type), then the submodule's parameter assignment list, enclosed in braces, may replace the optional initialization. The first declaration in this example (Figure 4.9) defines a **PLATFORM** parameter of

PLATFORMTYPE (special data type defined in another knowledge base category). The declaration for the DM parameter identifies a DATABASE_MANAGER submodule with five initialized parameters.

Interface declarations are very similar to parameter declarations. The declaration consists of an interface name, followed by a colon and the interface type. An initialization can assign a start value for interface ports.

Relationship declarations begin similar to other declarations. The relationship name precedes a colon and relationship type. However, a **reference pair** must follow the relationship type. A reference pair consists of two scope references, enclosed in parentheses and separated by a comma. A scope reference identifies a relationship player and takes the following form:

```
SCOPEREFERENCE ::= Identifier | SCOPEREFERENCE.Identifier
```

Depending on the relationship type, the first reference represents the sender and the second reference represents the receiver. A relation initialization may follow the reference pair. This initialization identifies the data type or defines a function assumed by the relationship. The first relationship declares the BSB_TDB_IN_ASYNC relationship between interfaces of the DM and BSB submodules that assumes a STRING data type.

Relationships declared for primitive modules in the Library Components Structure are similar to relationships declared in the the Architecture Construction Rules. The only difference is in form. Most primitive modules will not have relationship declarations because this often indicates that the primitive module could be separated into lower-level objects and, therefore, might not need to be a primitive module.

Notice that data types for parameters and relationships may range from simple structures (e.g., NUMBER or BOOLEAN) to very complex structures (e.g., RECORD [NAME : STRING; SSAN : STRING; ADDRESS : STRING]). Relationships initialized with function definitions can be very complex and follow the rules outlined in The CIDL Language User Manual (29).

The knowledge base stores the Library Components Structure in the `global.gsdl-1` file. The Library Components Structure begins with a domain identifier followed by the

structure declarations. A grammar for this category appears in the Software User's Manual for APTAS (20:40-41).

4.3.6 Library Modules. The Library Modules category provides the CIDL Code Synthesis Engine with primitive module implementations. The library also includes files with information needed by the primitive modules. The information includes code for global type definitions, domain-specific type definitions, functions called from multiple primitives, and any other information not normally encapsulated by the primitive module. The library gives APTAS the capability to generate executable code and run tracker simulations using the requested primitive implementations.

Primitive modules appear as leaves of the tree structure defined in the Architecture Construction Rules. The Library Components Structure defines the primitive's accessible parameters, interfaces, and relationships. Each primitive takes the form of a template with up to four parts. The first part, **PARAMETERS**, defines application-specific characteristics of the primitive (if any) through the declaration of attributes (parameters and interfaces) accessible to the Tracking Engineer. The second part, **INTERFACE**, defines external characteristics of the primitive (if any) that are visible outside the module (e.g., functions available to other primitive modules). The third part, **STRUCTURE**, defines the internal characteristics (local structure) of the primitive. The local structure may include the declaration of internal primitives (submodules of a primitive module not available through normal composition methods) and the establishment of relationships to and between the internal primitives. This structure may also include the declaration of local variables and functions. The fourth part, **BEHAVIOR**, defines the processing performed by the module whenever created (i.e., how it produces output data from its input data).

The knowledge base stores the Library Modules in the `include` directory (within the domain's home directory). Each module consists of a CIDL source file, a compiled file, and all the files on which it depends. The APTAS documentation ((20:19,20) and (17:24,25)) gives a format for Library Modules that actual implementations have never used (the existing modules are implemented in CIDL).

4.3.7 Coding Rules. The Coding Rules category provides the CIDL Code Synthesis Engine with transformation rules. There are two types of rules: generic and domain-specific (17:6,25-26). The generic coding rules are part of the synthesis engine itself and do not change. Each generic coding rule consists of an architecture pattern and its corresponding CIDL implementation. When portions of the tracker architecture match a pattern, the synthesis engine replaces the pattern with the corresponding CIDL implementation.

Each domain-specific coding rule consists of a CIDL implementation, a test expression pattern, and a corresponding replacement CIDL implementation. Test expressions usually involve a module's specific parameter values. When the test expression and portions of the synthesized code (resulting from generic coding rule replacements) match the pattern, the new CIDL implementation replaces the previously synthesized code. The addition or modification of domain-specific coding rules requires an experienced CIDL programmer.

The knowledge base stores the Coding Rules (domain-specific). However, there is no evidence outside the documentation for their existence. The current APTAS system does not have any of these rules.

4.3.8 Synthesis Support. There is another category of knowledge that provides information to the CIDL Code Synthesis engine. It is called Synthesis Support and was not shown in Figure 4.2 because it has no unique information. This category of knowledge "contains templates which identify the CIDL parameters of each primitive module type" (20:26). The templates describe an abstract syntax signature for the Library Modules, showing function names, parameter names and types, and optional default parameter values. This category maps a primitive module type to an actual CIDL implementation in the Library Modules (associates the name of the primitive module type to the name of an implementation file) (21). These templates represent the signature that would result from parsing a primitive module. Using the Synthesis Support, the code synthesis engine works more efficiently because it does not have to parse all of the Library Modules.

4.4 APTAS Knowledge Base Population Process

After studying the knowledge base structure, putting new information into the Domain Taxonomy and Coding Knowledge Base is not an easy task. This section develops a formal knowledge base population process that reduces the effort involved. The formal process depends on information learned from the examination of APTAS and its knowledge base, as well as the process outlined developed in Chapter III.

The APTAS developers summarized the knowledge base population process in the following statement (17:20):

> Building the knowledge base entails gathering and formalizing the expertise of application domain specialists. The overall object is to capture general information (terminology, parameters which affect system design, typical range values for parameters, etc.), the taxonomic structure of tracking components, and the reasoning used to construct designs (including how one design decision places constraints on subsequent decisions).... This information goes into an informal domain description document (or set of documents) to be used by a knowledge engineer for formalization.... The knowledge engineering effort (both collecting the information and formalizing it for storage in the knowledge base) will generally continue over time, enriching the knowledge base as more information becomes available. It is possible to generate designs when the knowledge base is in the early stages of development; however, the flexibility and diversity of those designs depend on the maturity of the knowledge base.

This statement is too informal to implement, however, it does contain several key roles involved in the population effort and several important results. General domain knowledge is captured by Domain Specialists into a domain description document (a domain model). This model is formalized by the Knowledge Engineer and stored in the knowledge base.

Figure 4.10 displays the APTAS Knowledge Base Population Process. To remain consistent with the quotation above, the **Domain Specialists** assume the role of the Domain Engineer, and the **Knowledge Engineer** assumes the role of the Software Engineer. Several other changes that are specific to the terminology used in the APTAS documentation appear in the process diagram. For instance, library modules represent reusable components and the Domain Taxonomy and Coding Knowledge Base represents the domain infrastructure. The rest of this chapter describes the development of this formal population process for the APTAS Knowledge Base.

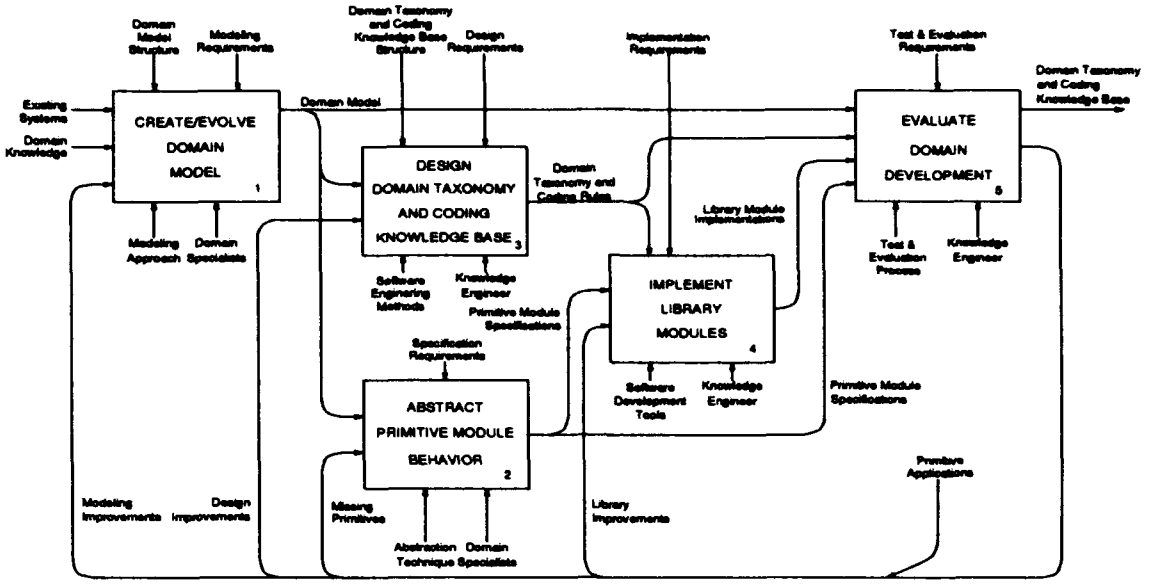


Figure 4.10 Knowledge Base Population Process for APTAS.

4.5 Instantiating the Knowledge Base Population Process

To instantiate a population process for a specific system, the methods and constraints involved in each of the general process outline activities must be defined. This section presents a strategy for instantiating a formal process for a specific system using the general population process.

Given: an application domain and a description of the Domain-Oriented Application Composition System.

Produce: the methods and constraints to populate the knowledge base with new domain information.

Strategy: Perform the following:

1. Formalize the domain modeling requirements:
   - bound the application domain (define what to analyze),
   - identify sources of domain information (especially existing applications), and
   - explain procedures to evolve the domain model.
2. Specify the domain model structure.
3. Select the domain modeling approach that meets the modeling requirements and best fits the structure of the domain.
4. Formalize the specification requirements (including interface standards).
5. Select an abstraction technique that sufficiently captures component (primitive module) behaviors and fits into the framework of the chosen modeling approach.

6. Formalize the design requirements:
   - capture standard design practices that control application composition and
   - explain procedures to evolve the reuse infrastructure (knowledge base) design.
7. Design the knowledge base structure and map corresponding structures from the domain model.
8. Formalize the reuse infrastructure implementation requirements:
   - identify the implementation language,
   - define the software architecture(s) required (or allowed) in the system, and
   - explain procedures to change component implementations without adversely affecting generated applications.
9. Select software development tools to meet the implementation requirements.
10. Formalize the test and evaluation requirements:
    - describe consistency checking requirements (of domain model and reuse infrastructure design),
    - define completeness requirements (e.g., all primitive components implemented), and
    - explain measures to test any proposed evolutions against existing versions.

4.5.1 Create/Evolve Domain Model. The first three steps of the instantiation strategy must often be repeated for each domain captured in the knowledge base. The first step formalizes the modeling requirements: bound the application domain, identify existing domain knowledge sources, and explain domain model update procedures. The second step defines the domain model structure. The third step selects a domain modeling approach (either an existing approach or a new one). The Create/Evolve Domain Model activity uses the domain modeling approach, controlled by the modeling requirements, to capture domain information into the domain model structure. A domain model results from the capturing of domain information during this activity.

Although this thesis does not focus on the modeling of specific domains, a specific domain must be selected and bound during the first step of the process. APTAS already captures a formal domain representation in its Tracking Taxonomy and Coding Knowledge Base. The information currently captured in the knowledge base involves the radar tracking domain (specifically, the domain of **track-while-scan, multiple-target tracking systems**).

During the first step, any existing sources of information must be identified. The primary source of domain information is the Tracking Taxonomy and Coding Knowledge Base itself. This information must be supported when adding new information to the knowledge base. Because there is no domain model for this existing information, a domain model must first be reverse engineered from this information before capturing new information. This adds some bias to the resulting domain model because the knowledge base structure depends on the system requirements. Two documents from Lockheed, the *Final Report for APTAS* (17) and the *Avionics Tracking System Specification for Automatic Code Synthesis* (16), have additional information about the tracking domain captured in the knowledge base. Existing tracker implementations and radar textbooks, like those written by Blackman (5) and Hovanessian (13), may provide new domain information to capture.

Procedures to update the domain model are also explained during the first step. Updates to the domain model result from capturing new domain information. The model organizes the process of knowledge identification and encapsulation through its formal structure of object classes and associations. Model updates result bv adding new classes, associations, or attributes and by following the specific modeling approach selected during the third step of the process instantiation.

Ideally, the Knowledge Engineer and the Domain Specialists define a formal domain model structure (meta-model) capable of representing every aspect of the specific domain in the second step. Although not based directly on the tracking domain, a domain model structure can be defined that is general enough to capture the information currently stored in the knowledge base and to capture new information identified in the domain. The model structure is relatively simple and follows the Rumbaugh object modeling guidelines (34). The object model structure shown in Figure 4.11 describes the information captured from the tracking domain. Object classes represent modules, parameters, interfaces, relationships, forms, questions, responses, and other important classes. Associations represent the relationship between forms and modules, as well as the structure of the modules and forms. Attributes represent object instance names, types, and other information relating to particular object instances.

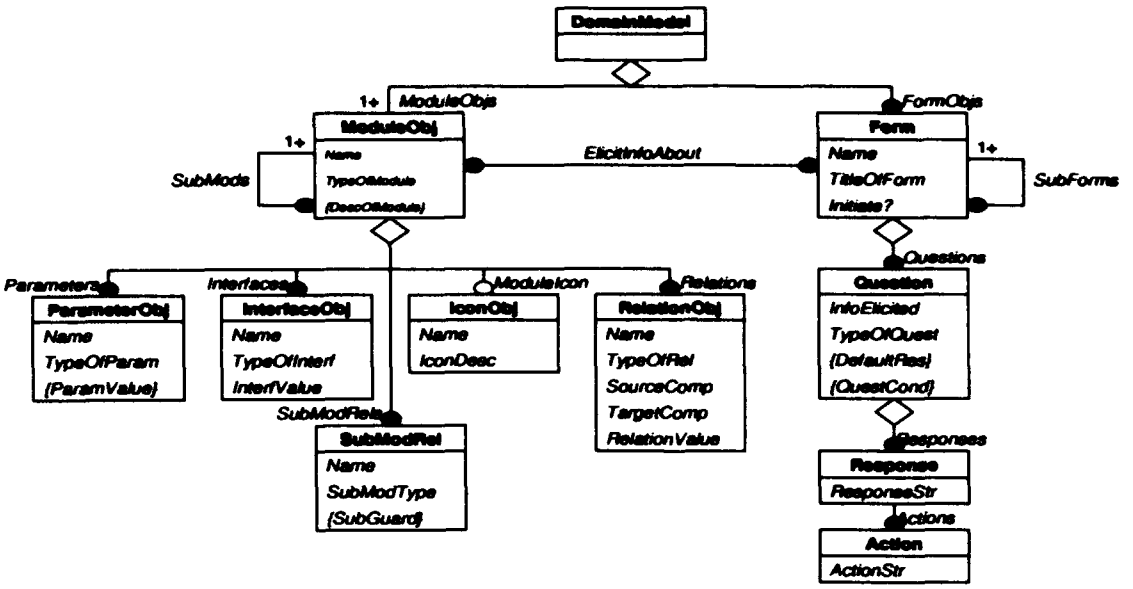


Figure 4.11 Domain Model Structure.

Because the domain model structure originates from the Rumbaugh Object Modeling Technique (34), this technique is selected as the domain modeling approach. A domain model is created through the application of the Object Modeling Technique on a specific domain. Updates to the object model, representing new domain information, evolve the existing domain model without corrupting previously captured information. The Domain Specialists can also use this approach to specify the behavior of any primitive modules because this technique includes more than just an object model. Functional models and dynamic models are used to specify behaviors during the Abstract Primitive Module Behavior activity shown in Figure 4.10.

4.5.2 Abstract Primitive Module Behavior. The next two steps (from Section 4.5) to instantiate the Knowledge Base Population Process for APTAS involve the Abstract Primitive Module Behavior activity. This activity specifies implementations for primitive module behaviors (i.e., their dynamic and functional models). The first of these two steps (step 4) directs us to formalize the specification requirements (e.g., interface standards) that define the results of behavior abstraction. The second step (step 5) directs us to select an abstraction technique.

It is important to realize that the Knowledge Engineer can generate many different implementations that meet a primitive module's specified behavior. The specification requirements declare the global data types and utilities that primitive modules must use. A dictionary explaining domain-specific terminology might also appear with these requirements. The specification requirements must also include some interface standard to ensure the proper passing of data between primitive modules regardless of specific implementation. Such standards might include communication protocols, naming conventions, or special data types. An interface standard might exist for every primitive identified in the domain model.

The interface standard for the domain consists of communications through data ports or function calls. Input data ports (In_Ports) receive a particular data type from another module. Output data ports (Out_Ports) send a particular data type to another module. Operations (external functions) may be called by another module (which also provides data to the operation). Operations might also send data to another module's in_port.

Functional models and dynamic models from the Rumbaugh technique (34) define the abstract primitive module behaviors. The object model portion of this technique is already used as the domain modeling approach. The dynamic and functional models specify the behavior to whatever detail necessary. Data dictionaries augment the understanding of these models by describing (in English) what the models represent.

The Create/Evolve Domain Model and Abstract Primitive Module Behavior activities results in the domain model and a set of primitive module definitions that fully specify an application domain and provides information to define the rest of the knowledge base population process.

4.5.3 Design Domain Taxonomy and Coding Knowledge Base. The next two steps (from Section 4.5) involve the Design Domain Taxonomy and Coding Knowledge Base activity. This activity takes the organization of modules captured in the domain model and represents it in the form required by the knowledge base. First (step 6), the design requirements are formalized by describing the knowledge base structure, mapping domain model structures to knowledge base structures, identifying structures that must be

implemented, and capturing standard design practices. Second (step 7), a software engineering methodology is selected (or developed) that can satisfy all the design requirements.

Section 4.3 described the knowledge base structure in detail. The domain model structure was derived from the knowledge base structure (because this thesis does not attempt to structure the tracking domain as an expert). This makes mapping structures between the model and the knowledge base rather straightforward. Structures requiring implementation have also been identified; namely, the primitive modules. The APTAS system already contains a structure to capture standard design practices through its question forms; therefore, the design requirements have already been presented.

Selecting a methodology to design a knowledge base from the domain model is not quite as easy. The method shown in Figure 4.12 was proposed after looking at the relations between the seven knowledge base categories (shown previously in Figure 4.2) and realizing that each category has its own grammar. The domain model should already capture

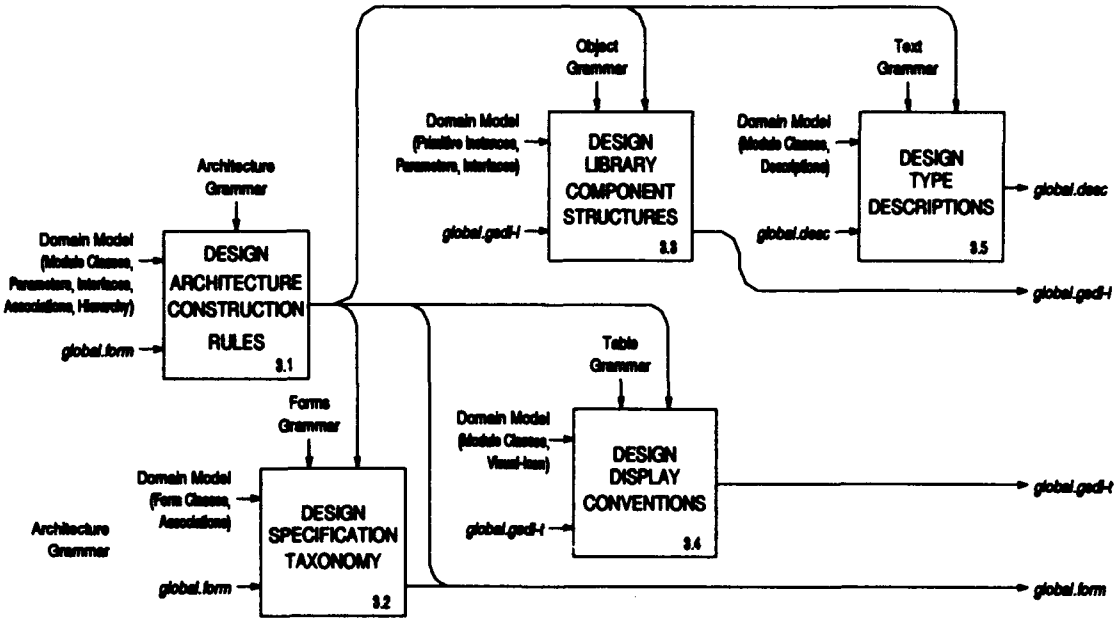


Figure 4.12 Method to Design a Domain Taxonomy and Coding Knowledge Base.

any design improvements passed back from the Evaluate Domain Development activity (described below in Section 4.5.5). The improvements may identify something in the domain model that was overlooked or misrepresented during design.

The updated domain model may result in the addition of new modules, the addition of new relations, the addition of new interfaces, or result in other changes to the existing Architecture Construction Rules (Design Architecture Construction Rules of Figure 4.12). The Knowledge Engineer must follow the Architecture Grammar when making these changes. New modules may allow the system to compose new applications. New modules might improve or change the performance of previously composed applications. New relations and interfaces might provide other modules access to more data and functions. Other improvements might result from modifications to any existing characteristics.

The updated domain model may also result in the addition of new levels (forms), the addition of new questions, or the modification of questions to the Specification Taxonomy (Design Specification Taxonomy of Figure 4.12). The Knowledge Engineer can add new forms to elicit additional specification information during composition. New questions for existing forms might also elicit additional information. Modifying an existing question might improve submodule specialization through the activation of additional forms or the selection of new primitives or parameters. Other modification might cause questions to apply only under certain conditions. More subtle changes occur when the model results in different actions taken for the response of an existing question. Such modifications might not be visible until much later in composition. For instance, if a modified question now sets an additional parameter affecting application simulation, nothing will appear different until a user simulates the new application. Because actions are the only way to set the internal variables (status variables) used in the Architecture Construction Rules, these variables (as well as module parameters) must be closely monitored.

The remaining information captured in the domain model affects the three other knowledge base categories. Changes to these categories do not affect the performance of applications composed using APTAS. The Knowledge Engineer can add new icons, change existing icons, or improve the application domain documentation. The Application Specialist (particularly the Tracking Engineer) has access to new or updated information as soon as the changes occur. New primitive modules (or changes to existing primitives) require changes to the Library Components Structure (and possibly the Display Conventions or Type Descriptions) and also require new (or changed) implementations.

4.5.4 Implement Primitive Modules. Any new primitive modules, or any changes to existing primitives, require changes to the Library Modules (and possibly the Coding Rules or Synthesis Support). The new (or changed) primitives might give the Application Specialist the capability to compose applications with new behaviors. Changes to the synthesis support have the same impact. Modifications to existing library modules might improve tracker performance or correct previous errors.

All primitive module updates occur during the Implement Primitive Modules activity. Two steps from the process instantiation strategy (from Section 4.5) define this activity. The first (step 8) formalizes the implementation requirements: identify the implementation language, define a software architecture, determine priorities for implementing primitives, and explain procedures to avoid new implementation integration problems. The second (step 9) selects a software development environment to do the implementations.

The Knowledge Engineer must implement primitive modules using CIDL. As mentioned in Section 4.3.6, primitive implementations (Library Modules) consist of four parts. The **PARAMETERS** part represents a primitive's parameters and interfaces. This information is available in the new Architecture Construction Rules. The **BEHAVIOR** part implements (in CIDL) the behavior abstracted during the Abstract Primitive Module Behavior activity. Any data structures and functions (not externally available) needed for behavior implementation must be declared in the **STRUCTURE** part. The **INTERFACE** part is not well documented. Apparently, it serves to declare information originating within the primitive and accessible to other modules. This being the case, **PARAMETERS** would represent information required by the primitive and originating from another module. These four parts represent the software architecture APTAS uses for all primitive implementations. At this point, implementation priorities are all equal. With a robust domain model and knowledge base design, priorities might have more impact during implementation. However, the Knowledge Engineer must be careful when integrating new implementations to ensure they are accessible to the Tracking Engineer through the Specification Taxonomy (question forms).

Implementations must occur within the CIDL environment of the APTAS system. The Knowledge Engineer can access this environment using the **synthe** program in the

APTAS `bin` directory. Primitives are implemented using this development environment and an object-oriented programming technique (34).

4.5.5 Evaluate Domain Development. The last step in instantiating the Knowledge Base Population Process for APTAS involves the Evaluate Domain Development activity. This step (step 10 from Section 4.5) formalizes the test and evaluation requirements. These requirements describe the consistency checks that must be made during the other four activities. They define what determines the completion of each activity. They also explain measures that compare the updates made at each activity against the old information (i.e., measure the changes resulting from each activity execution). Different requirements apply to each of the other four activities.

The requirements for the Create/Evolve Domain Model activity involve checking the developing domain model for consistency against the domain model structure (see Figure 4.11) and determining when the modeling activity is complete. The model structure requires that forms have at least one question and questions have at least one response. Submodules must consist of at least one other module. The *elicits-information-about* association (between questions and modules) cannot be checked since it is optional; however, each question normally associates with at least one module. The *interfaces-with* association (between modules) cannot be checked since it is also optional. A final consistency check is the inclusion of at least one primitive module. Other consistency checks may be added to further constrain model creation/evolution, such as requiring a description for each module or requiring at least one parameter for each primitive. The evaluation of the Create/Evolve Domain Model activity is complete when all the consistency checks pass.

The requirements for the Abstract Primitive Module Behavior involve checking the dynamic and functional models against the standards developed by Rumbaugh (34). An event (resulting in a module changing state) requires a source (probably from another module's dynamic model). Events should involve a module interface; those that do not (external events) require a separate definition. All guards and data passed with events must be defined by some state's activities/actions or some module's attributes. Activities and actions should correspond to distinct functional models. Data passed into and out of

a functional model should correspond to data passed with events or to a module's interfaces and parameters. The evaluation of the Abstract Primitive Module Behavior activity is complete when all primitive modules identified in the domain model have consistent functional models, dynamic models, or appear as data stores (passive objects) in existing functional models.

The requirements for the Design Domain Taxonomy and Coding Knowledge Base involve checking the consistency of the information in the knowledge base categories. All parameters set in the Specification Taxonomy or listed in the Library Component Structures must correspond to parameters declared in the Architecture Construction Rules. Module icons defined in the Display Conventions must correspond to modules declared in the Architecture Construction Rules. Status variables used in the Architecture Construction Rules and Specification Taxonomy must be set by question responses in the Specification Taxonomy. The Architecture Construction Rules may declare only predefined relationship and interface types. The Display Conventions may define icons for these same relationship and interface types. Consistency checking the Type Descriptions would only provide a warning, and any checks of this category were ignored. A final consistency check involves ensuring that no forms or modules exist in these categories that do not appear in the domain model. The evaluation of the Design Domain Taxonomy and Coding Knowledge Base activity is complete when all forms, questions, modules, and their attributes are transferred from the domain model to the Domain Taxonomy and Coding Knowledge Base.

The requirements for the Implement Primitive Modules involve checking the remaining knowledge base categories for consistency with the other categories. Primitives defined in the Library Modules must correspond to primitive declarations in the Architecture Construction Rules. All the parameters and interfaces must appear in the **PARAMETERS** and **INTERFACE** parts in the Library Modules. Coding Rules must correspond to primitives in the Library Modules. Items in the Synthesis Support must correspond to primitives in the Library Modules. Variables used in a module's dynamic model (which are not parameters, interfaces, or passed with events) must appear as local variables in the **STRUCTURE** part of the primitive's implementation in the Library Modules. Checking that the **BEHAVIOR** part

is consistent with a primitive's dynamic and functional models is difficult and depends on the Knowledge Engineer's implementation technique. The evaluation of the Implement Primitive Modules activity is complete when all behaviors specified in the Abstract Primitive Module Behavior activity have corresponding Library Modules and Synthesis Support items.

The Domain Specialists and Knowledge Engineer may make many changes. Some changes correct consistency errors or move toward completeness. Other changes result from additional requirements from the Tracking Engineer and begin by adding new information to the domain model or behavior specifications. A suggested procedure might involve developing several example applications (like the **default_tracker**) whose behaviors are known. Every time the knowledge base is updated, these example applications should be simulated to verify their behaviors (make sure they still execute as expected). The procedures used to test and evaluate a domain's development must ensure the integrity of the previous information, thereby improving the confidence in the integration of any new information.

4.6 Summary

This chapter has instantiated a process to populate the APTAS knowledge base. The beginning of this chapter explained the general operation of the APTAS system and how each part of the system used the knowledge base. A great deal of information was covered on the formal structure of the knowledge base and the types of information required by the various categories. This information was crucial to developing the APTAS population process.

This process could conceivably be used for any application domain because the domain model structure is not dependent on a particular application domain (i.e., knowledge base structure was used to develop the domain model structure). In most cases, however, a Domain Engineer would develop a domain model structure based on the particular application domain itself. One should provide the same level of detail about the specific application domain (particularly when describing the domain model structure) that was

provided about the knowledge base. This information would result from bounding and scoping the domain.

The process itself consists of five activities: Create/Evolve Domain Model, Abstract Primitive Module Behaviors, Design Domain Taxonomy and Coding Knowledge Base, Implement Library Modules, and Evaluate Domain Development. The first activity results in a domain model. The second activity uses the domain model to identify and define primitive modules. The third activity places information captured in the domain model into the knowledge base. The fourth activity places the behaviors captured in the primitive module definitions into the knowledge base. The final activity catches any mistakes that might have been made during the other activities.

Some of the process activities could be automated or could have development tools built to assist in their execution. Graphical environments could support the creation and evolution of domain models, as well as the abstraction of primitive module behaviors. Transformation functions could take the domain model and behavior definitions and map them directly into corresponding knowledge base structures. Many consistency and completeness checks for the last activity could be automated. Automating these checks, as well as automating each of the other activities in the Knowledge Base Population Process, should be the goal of future research.

The next chapter describes how this formal population process was used to populate the Tracking Taxonomy and Coding Knowledge Base with new information.

V. A Typical APTAS Knowledge Base Population

5.1 Introduction

The process developed in previous chapters depends upon a domain model. The model results from the initial application of the Create/Evolve Domain Model activity. The Knowledge Engineer cannot use the five-step process to populate the knowledge base without a domain model. As mentioned previously, a domain model should capture information in a form that Domain Specialists understand. The model serves as input to every other population process activity, except the Implement Library Modules activity.

Currently, the APTAS Knowledge Base contains information that supports the generation and simulation of very simple trackers. However, the tracking information stored in the knowledge base, representing the tracking domain's reuse infrastructure, has no corresponding domain model (none found in the available documentation). Therefore, a tracking domain model must be created that corresponds to the existing contents of the knowledge base. Section 5.2 describes a procedure to reverse engineer the existing contents of the knowledge base and shows the resulting tracking domain model.

The Design Domain Taxonomy and Coding Knowledge Base activity transforms the information captured in the model into representations captured in the knowledge base. Section 5.3 describes a procedure to transform the tracking domain model into the form required by the APTAS Knowledge Base and compares the knowledge base files transformed from the new tracking domain model to those files currently in the knowledge base.

An important task of the Create/Evolve Domain Model activity involves updating (evolving) an existing domain model so new information can be propagated to the knowledge base. Section 5.4 describes a procedure to evolve the tracking domain model. Adding another primitive module (an intersecting tracks generator) as a substitute submodule of an existing primitive module (the sensor model) shows the implementation of these evolution procedures.

APTAS builds trackers using the information from the knowledge base, especially the primitive module implementations. The most tedious activities in the population process involve capturing the primitive modules: Abstract Primitive Module Behavior

and Implement Library Modules. The first of these two activities results in functional definitions (specifications) for the primitive modules while the second results in CIDL implementations for these specifications. Section 5.5 describes a procedure to specify a primitive module's behavior and defines the behavior of the intersecting tracks generator module. Section 5.6 describes a procedure to implement a primitive module from its specification and implements the intersecting tracks generator primitive module (adding it to the Library Modules and Synthesis Support).

The Knowledge Engineer can evaluate each stage of the domain development before proceeding to the next stage. The Knowledge Engineer should always evaluate changes to the contents of the knowledge base before making it available to any Tracking Engineers. Section 5.7 describes procedures to evaluate the domain model, the knowledge base transformed from the model, the functional specifications, and the CIDL implementations.

5.2 Creating the Tracking Domain Model

Because the existing contents of the knowledge base do not cause any errors during APTAS operation, the tracking domain model is reverse engineered from the knowledge base without first evaluating the contents formally. However, the resulting domain model was evaluated and indicated several small inconsistencies. The reverse engineering procedure shown in Figure 5.1 creates an initial tracking domain model. This reverse engineering procedure was implemented using Software Refinery.

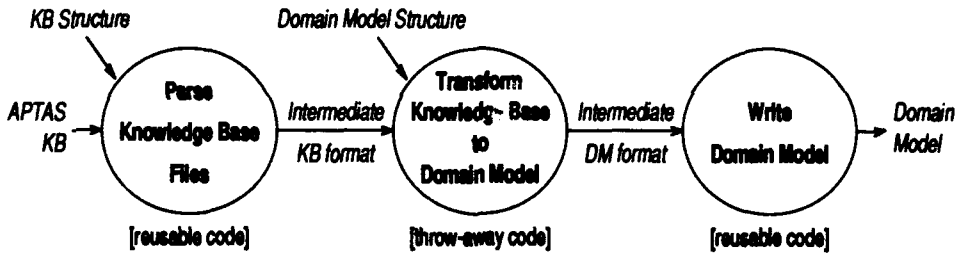


Figure 5.1 Reverse Engineering Procedure.

The procedure consisted of three functions. The first function parses the existing APTAS Knowledge Base information using the knowledge base structure described in

in Section 4.3. This structure was represented using four grammars developed with the DIALECT tool (Software Refinery's parser generator). These four grammars parse knowledge base files into an abstract syntax tree representation in Software Refinery's object base. The object base provides a collection of operations to create and manipulate objects in an abstract syntax tree. The intermediate formats for the knowledge base and domain model are actually abstract syntax trees of objects within the object base.

The second function transforms the abstract syntax tree representation of the knowledge base into an abstract syntax tree representation of the domain model using a simplified representation of the domain model structure described in Section 4.5.1. The domain model's abstract syntax tree was built by transforming knowledge base objects into their corresponding domain model objects by making the predicates in Figure 5.2 true. Note

$$
\begin{array}{llll}
\forall r & [r \in \mathit{ArchConstRule} & \Longrightarrow \exists m & [m \in \mathit{ModuleObjs} \\
 & & & \wedge\ \mathit{Name}(m) = \mathit{NameText}(r) \\
 & & & \wedge\ (r = \mathit{first}(\mathit{ArchConstRule}) \Rightarrow \\
 & & & \qquad \mathit{TypeOfModule}(m) = \mathit{ENV}) \\
 & & & \wedge\ (r \neq \mathit{first}(\mathit{ArchConstRule}) \Rightarrow \\
 & & & \qquad \mathit{TypeOfModule}(m) = \mathit{SUB}) \\
 & & & \wedge\ \mathit{SubModRels}(m) = \mathit{SubModuleList}(r) \\
 & & & \wedge\ \mathit{Relations}(m) = \mathit{ModRelList}(r) \\
 & & & \wedge\ \mathit{Interfaces}(m) = \mathit{ModDeclList}(r)]] \\
\forall l & [l \in \mathit{LibCompStruct} & \Longrightarrow \exists m & [m \in \mathit{ModuleObjs} \\
 & & & \wedge\ \mathit{Name}(m) = \mathit{Name}(l) \\
 & & & \wedge\ \mathit{TypeOfModule}(m) = \mathit{PRI} \\
 & & & \wedge\ \mathit{Relations}(m) = \mathit{RelList}(l) \\
 & & & \wedge\ \mathit{Interfaces}(m) = \mathit{DeclList}(l) \\
 & & & \wedge\ \mathit{Parameters}(m) = \mathit{DeclList}(l)]] \\
\forall d & [d \in \mathit{Descriptions} & \Longrightarrow \exists m & [m \in \mathit{ModuleObjs} \\
 & & & \wedge\ \mathit{Name}(m) = \mathit{Name}(d) \\
 & & & \wedge\ \mathit{DescOfModule}(m) = \mathit{TypeDesc}(d)]] \\
\forall c & [c \in \mathit{Conventions} & \Longrightarrow \exists m & [m \in \mathit{ModuleObjs} \\
 & & & \wedge\ \mathit{Name}(m) = \mathit{Name}(c) \\
 & & & \wedge\ \mathit{ModuleIcon}(m) = \mathit{IconIdentifier}(c)]] \\
\forall s & [s \in \mathit{SpecTaxonomy} & \Longrightarrow \exists f & [f \in \mathit{FormObjs} \\
 & & & \wedge\ \mathit{Name}(f) = \mathit{Name}(s) \\
 & & & \wedge\ \mathit{TitleOfForm}(f) = \mathit{FormTitle}(s) \\
 & & & \wedge\ \mathit{Initiate?}(f) = \mathit{ActiveFlag?}(s) \\
 & & & \wedge\ \mathit{Questions}(f) = \mathit{QuestionList}(s)]]
\end{array}
$$

Figure 5.2 Reverse Transformation Predicates.

that the normal direction of transformation is from domain model to knowledge base (forward transformation). To avoid influencing the domain model with knowledge base structure details the direction of transformation implemented by these rules (knowledge base to domain model) is not part of the usual population process.

The domain model structure was also represented using a grammar developed with the DIALECT tool. However, the domain model structure was simplified for the reverse transformation (primarily because this direction of transformation should not occur more than once). Therefore, a separate function traverses the transformed abstract syntax tree and writes the domain model in its proper ascii form (i.e., corresponding directly to the domain model grammar described in Appendix A).

The reverse engineering procedure built an initial tracking domain model. Figure 5.3 shows an object model representation of the tracking domain model. This representation

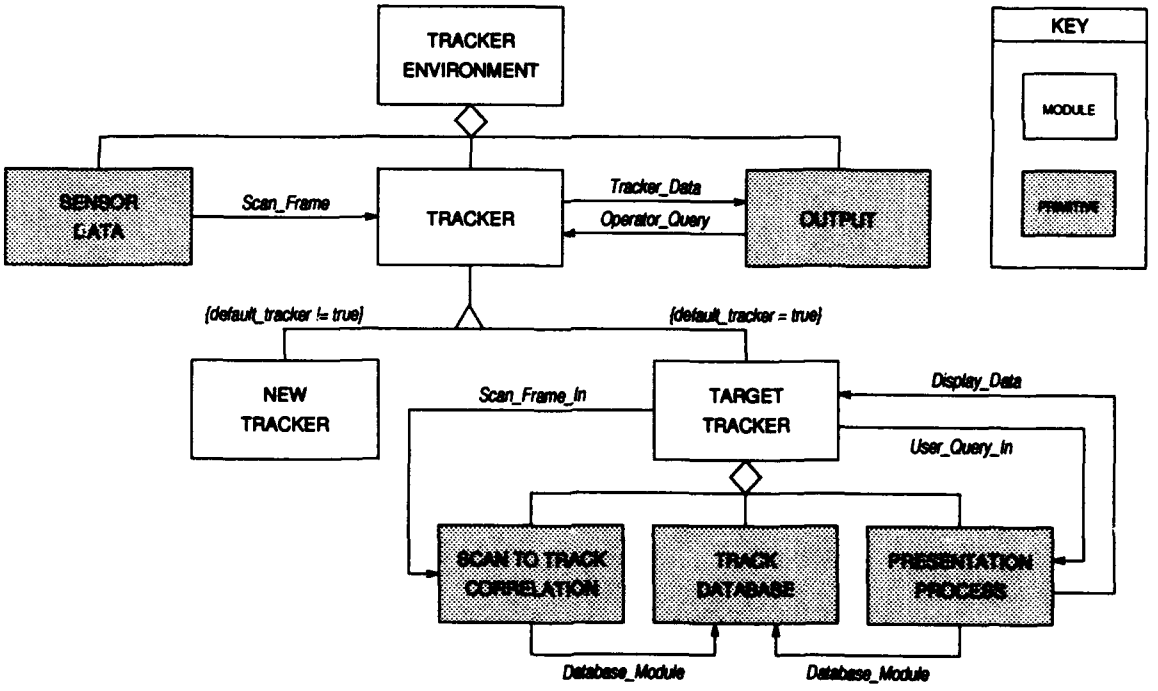


Figure 5.3 Partial Tracking Domain Model.

does not show all the modules or any of the question forms. It also does not show any of the module parameters or interfaces. The figure displays only the portion of the model

used to complete this research. Relation objects replaced Rumbaugh object associations and submodule objects replaced Rumbaugh object aggregations.

Before using this initial tracking domain model created with the reverse engineering procedure, the model was transformed back into the APTAS Knowledge Base, and the new knowledge base files were compared to the existing files.

5.3 Designing the Tracking Knowledge Base

Transforming the domain model file into the knowledge base files is the purpose of the Design Domain Taxonomy and Coding Knowledge Base activity. This procedure implements the normal transformation direction and consists of only two functions (versus the three developed for the reverse engineering procedure). The knowledge base files are written during the transformation function. The transformation procedure shown in Figure 5.4 create new knowledge base files from the initial tracking domain model file.

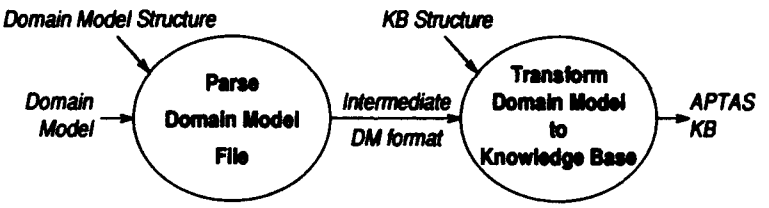


Figure 5.4 Forward Transformation Procedure.

The procedure consists of two functions. The first function uses the domain model grammar to parse the tracking domain model file into an abstract syntax tree. As mentioned earlier, the domain model grammar was built from the structure presented in Section 4.5.1. The second function traverses the abstract syntax tree and uses the knowledge base structure (from Section 4.3) to write new APTAS Knowledge Base files.

The rules for the forward transformation procedure were derived from predicates similar to those for the reverse engineering procedure. The forward transformation, however, required the creation of multiple knowledge base files from a single domain model file. The knowledge base files resulting from the transformed domain model in Figure 5.3 should appear identical (if not very similar) to the original knowledge base files. While informa-

tion was *condensed* in the reverse engineering procedure, information was *expanded* in the forward transformation. The forward transformation postconditions are characterized in Figure 5.5.

$$
\begin{array}{lll}
\forall f \ [f \in FormObjs & \Longrightarrow \ \exists s \ [s \in SpecTaxonomy \\
 & \quad \wedge Name(s) = Name(f) \\
 & \quad \wedge FormTitle(s) = TitleOfForm(f) \\
 & \quad \wedge ActiveFlag?(s) = Initiate?(f) \\
 & \quad \wedge QuestionList(s) = Questions(f)]] \\
 & \\
\forall m \ [(m \in ModuleObjs & \\
\quad \& \ TypeOfModule(m) \neq PRI) & \Longrightarrow \ \exists r \ [r \in ArchConstRule \\
 & \quad \wedge NameText(r) = Name(m) \\
 & \quad \wedge SubModuleList(r) = SubModRels(m) \\
 & \quad \wedge ModRelList(r) = Relations(m) \\
 & \quad \wedge ModDeclList(r) = Interfaces(m) \\
 & \quad \wedge (TypeOfModule(m) = ENV \Rightarrow \\
 & \qquad r = first(ArchConstRule)]] \\
 & \\
\forall m \ [(m \in ModuleObjs & \\
\quad \& \ TypeOfModule(m) = PRI) & \Longrightarrow \ \exists l \ [l \in LibCompStruct \\
 & \quad \wedge Name(l) = Name(m) \\
 & \quad \wedge DeclList(l) = Interfaces(m) \cup Parameters(m) \\
 & \quad \wedge RelList(l) = Relations(m)]] \\
 & \\
\forall m \ [m \in ModuleObjs & \Longrightarrow \ \exists d \ [d \in Descriptions \\
 & \quad \wedge Name(d) = Name(m) \\
 & \quad \wedge TypeDesc(d) = DescOfModule(m)]] \\
 & \\
\forall m \ [m \in ModuleObjs & \Longrightarrow \ \exists c \ [c \in Conventions \\
 & \quad \wedge Name(c) = Name(m) \\
 & \quad \wedge IconIdentifier(c) = ModuleIcon(m)]]
\end{array}
$$

Figure 5.5 Forward Transformation Postconditions.

As expected, the knowledge base files that resulted from executing the forward transformation procedure appeared almost identical to the original knowledge base files. The major differences were in the indentation and carriage returns between the two sets of knowledge base files, and the capitalization of variable names and symbols in the new files. Therefore, although the two knowledge bases were behaviorally equivalent, the transformation resulted in minor style differences.

The APTAS system operated no differently with either its new files or its existing files. Therefore, the two transformations (forward and reverse) succeeded in creating an initial domain model that captured the current contents of the knowledge base. The

initial tracking domain model served as a baseline to describe and test procedures for the remaining activities of the knowledge base population process.

5.4 Evolving the Tracking Domain Model

Evolving a domain model involves the participation of the Domain Specialists. The Create/Evolve Domain Model activity uses a selected modeling approach, controlled by a domain model structure and any modeling requirements, to represent domain knowledge (including information from any existing systems) in a domain model.

The domain model structure was used to create an initial tracking domain model and to transform the model into the knowledge base files. This domain model structure controls the modeling approach and is crucial to performing this stage of the population process. The selected modeling approach involves updating the domain model file using this domain model grammar. The grammar captured all aspects of the domain model structure. A Domain Specialist simply updates the tracking domain model file to represent any new domain knowledge (using any ascii text editor) and saves the new domain file. Evolving the initial tracking domain model illustrates this procedure.

The tracking domain model shown in Figure 5.3 included a primitive module with the instance name **SENSOR DATA**. A simple modification of this primitive through object specialization is shown in Figure 5.6. The modification specializes the **SENSOR DATA** submodule into two different primitive modules. The existing primitive module, **SENSOR MODEL**, is the default selection and remains unchanged (except for its new hierarchical position) in the domain model. The **INTERSECTING TRACKS GENERATOR** is a new primitive module. This presentation of evolving the tracking domain model only shows the concept for the proposed modification. The real evolution task involves a Domain Specialist actually making these changes to the text in the domain model file.

The domain model file required several changes to realize this modification. First, the **TRACKER ENVIRONMENT** must include the option of selecting **INTERSECTING TRACKS GENERATOR** submodule. The **SENSOR DATA** submodule declaration may be one of two types, either the existing **SENSOR MODEL** primitive or the new **INTERSECTING TRACKS GENERATOR**

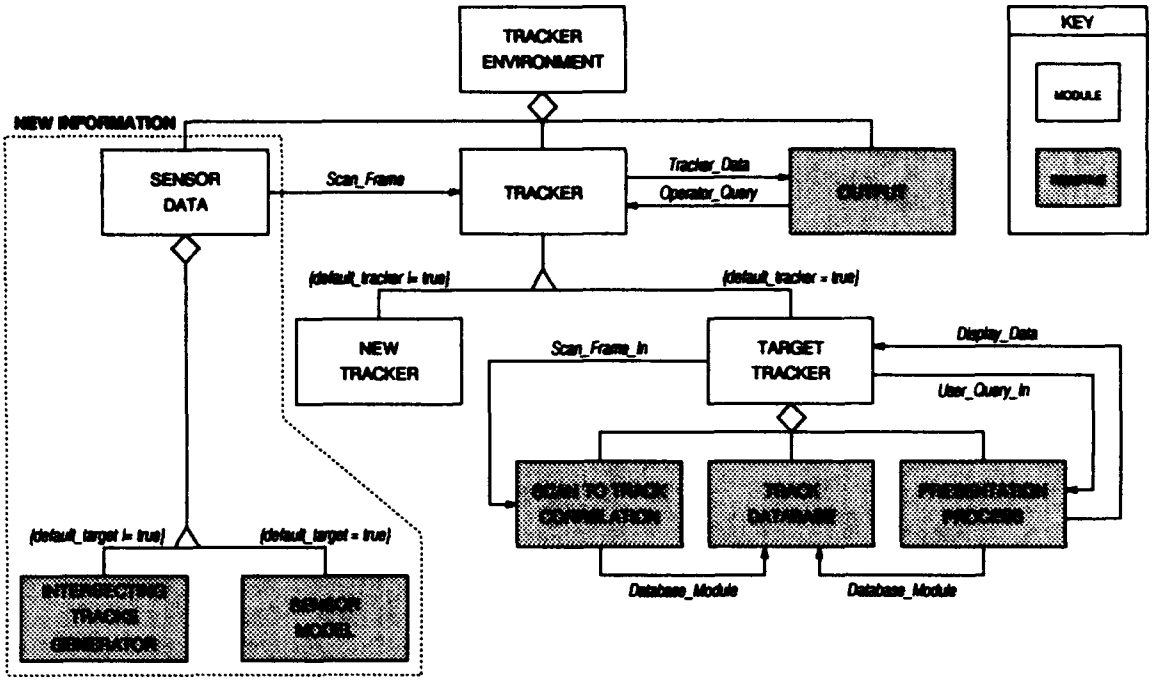


Figure 5.6 Evolved Tracking Domain Model.

primitive. Adding conditional expressions before the two submodule declarations (where only one condition is ever true) completes this change. Next, the new primitive module structure shown in Figure 5.7 was created. Notice that this primitive module structure includes the parameters and interfaces much like declaring the attributes of an object class. However, the parameters are initialized with values that must be generated by elicitation questions. A new elicitation form corresponding to this new primitive was added to the domain model that was not shown in Figure 5.6. A new question was also added to the root form (`/TRACKING`) to determine which option to use during tracker specification. The existing `SENSOR MODEL` primitive required no changes.

5.5 Abstracting the Behavior of the Intersecting Tracks Generator

The domain model file identifies several types of modules. The first module is the `ENVIRONMENT` module. This module is part of every tracker. Primitive modules are of the `PRIMITIVE` type. All other modules are of the `SUBMODULE` type. The type of module

```
MODULE INTERSECTING_TRACKS_GENERATOR (PRIMITIVE)
  Description:  "This module implements Lang Hong's routine for
                 generating noisy target data with false alarms.
                 This module first generates two perfect
                 intersecting tracks and then adds noise with a
                 normal distribution to obtain more realistic
                 target observations.  Uniformly distributed
                 false alarms within predetermined gates about
                 the track are then added."
  Icon: MED_BLUE_RECT
  Parameters:
    X_COORD_T1 (INT) = X_T1
    Y_COORD_T1 (INT) = Y_T1
    X_COORD_T2 (INT) = X_T2
    Y_COORD_T2 (INT) = Y_T2
    X_VEL_T1 (INT) = X_VEL
    Y_VEL_T1 (INT) = Y_VEL
    ITERATIONS (INT) = TEST_ITERATIONS
    X_NOISE_VARIANCE (INT) = X_NOISE
    Y_NOISE_VARIANCE (INT) = Y_NOISE
    MAX_FALSE_ALARMS (INT) = MAX_ALARMS
  Interfaces:
    SCAN_FRAME_OUT (OUT_PORT)
END
```

Figure 5.7 INTERSECTING TRACKS GENERATOR Primitive Module Structure.

is important to the Abstract Primitive Module Behavior activity since only modules of PRIMITIVE type have behaviors needing abstraction.

The domain model file provides information about the primitive's relations, interfaces, and parameters. Rumbaugh dynamic models and functional models abstract the primitive module's behavior. The knowledge base needs representations of both the domain model and the entire collection of primitive module definitions. Reverse engineering the existing primitive module behaviors was considered unnecessary.

The functional model shown in Figure 5.8 abstracts the behavior of the INTERSECTING TRACKS GENERATOR primitive module. The **Initialize** sets up the data for the two targets using several of the primitive module's parameters (including the target coordinates and initial target velocities). The **Produce Noisy Measures** generates a new position for each target using the target's initial state, the target's initial velocity, and the current scan iteration (e.g., first scan gives initial location). Noise is also added as specified in the parameters. The **Add False Alarms** includes additional target points (false alarms) with

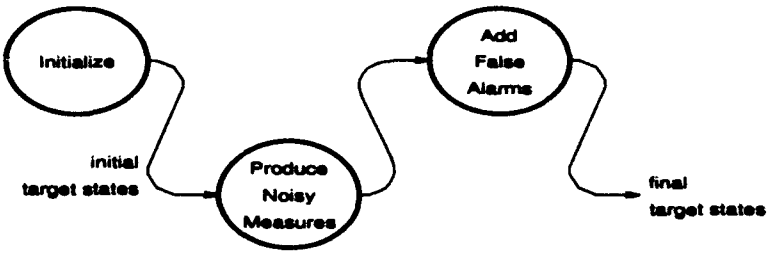


Figure 5.8 INTERSECTING TRACKS GENERATOR Abstract Behavior.

the noisy target information. A parameter of the primitive module specifies the maximum number of false alarms produced. This last function transfers information to the interface of this primitive.

A much more complex functional model could have been generated to define the behavior of the INTERSECTING TRACKS GENERATOR, but the behavior is simple enough to avoid complex definitions. Obviously, spending more effort defining a primitive module's behavior results in less effort required to implement the behavior in CIDL.

5.6 Implementing the Intersecting Tracks Generator

There are many ways to implement the INTERSECTING TRACKS GENERATOR using CIDL. The Implement Library Module procedure must support the relations, interfaces, and parameters identified in the domain model. The procedure must also support the synthesis engine of APTAS by updating the Synthesis Support category of the knowledge base.

The pseudo code shown in Figure 5.9 describes the CIDL implementation of the INTERSECTING TRACKS GENERATOR. An existing MATLAB (12) implementation of this module was used as a guide for creating and testing the CIDL implementation. A tool that transforms implementations from other source languages into CIDL would be a great help during this activity.

The INTERSECTING TRACKS GENERATOR implementation was placed within the Library Modules category of the APTAS Knowledge Base. APTAS expects Library Modules in the `include` directory of the domain. Both the primitive's CIDL source code and its

```
Initialize:
  tgt1pre = [X_COORD_T1  X_VEL_T1  Y_COORD_T1  Y_VEL_T1]
  tgt2pre = [X_COORD_T2  X_VEL_T1  Y_COORD_T2  -Y_VEL_T1]
Add Noisy Measures:
  tgt1cur = tgt1pre * Vel(scan) + Noise
  tgt2cur = tgt1pre * Vel(scan) + Noise
Add False Alarms:
  ScanFrameOut = [tgt1cur  tgt2cur  Alarms]
```

Figure 5.9 **INTERSECTING TRACKS GENERATOR** Primitive Module Implementation.

compiled file belong in this directory. The Synthesis Support file of the knowledge base must also be updated. Both of these tasks could be automated, however, these tasks were performed manually.

5.7 Evaluating the Tracking Domain Development

Once the primitive module implementations have been added to the knowledge base, the population process is almost finished (at least with the current iteration of the process). One important activity still remains – evaluation.

The knowledge base should work properly (be complete and consistent) before modifying the APTAS system's working knowledge base. As mentioned in Section 4.5.5, the Domain Specialists and Knowledge Engineer can describe evaluation procedures to check the consistency and completeness of each activity's results (domain model file, knowledge base files, abstract behaviors, and implementations). Evaluation procedures indicate any problems in a particular stage of domain development. Any indicated problems should be corrected before using the new knowledge base files. These procedures could also indicate when to stop an activity at each stage of the process.

After making the knowledge base consistent (and complete), the Knowledge Engineer can simulate several standard tracker simulations that exercise both new and old features of the knowledge base. This final testing should verify the integrity of the new knowledge base files (i.e., it still supports the applications that were composed prior to changing the knowledge base). After these last tests, the Knowledge Engineer can replace the APTAS system's Knowledge Base files with the new files.

5.7.1 Evaluating the Tracking Domain Model. Evaluating the domain model file involves checking that all conditions shown in Figure 5.10 are true. The conditions are

$$
\begin{array}{lcl}
\forall m[m \in ModuleObjs & \Longrightarrow & \neg\exists o[o \in ModuleObjs \\
 & & \quad \wedge\ o \neq m \\
 & & \quad \wedge\ Name(o) = Name(m)]] \\
 & & \\
\forall m[m \in ModuleObjs & \Longrightarrow & (TypeOfModule(m) = ENV \\
 & & \quad \vee\ TypeOfModule(m) = SUB \\
 & & \quad \vee\ TypeOfModule(m) = PRI)] \\
 & & \\
\forall m[(m \in ModuleObjs & & \\
\quad \wedge\ TypeOfModule(m) \neq PRI) & \Longrightarrow & Parameters(m) = \{\}] \\
 & & \\
\forall m[m \in ModuleObjs & \Longrightarrow & \forall i[i \in Interfaces(m) \\
 & & \quad \wedge\ (TypeOfInterf(i) = IN\_PORT \\
 & & \quad\quad \vee\ TypeOfInterf(i) = OUT\_PORT)]] \\
 & & \\
\forall m[m \in ModuleObjs & \Longrightarrow & \forall r[r \in Relations(m) \\
 & & \quad \wedge\ (TypeOfRel(r) = SYNC \\
 & & \quad\quad \vee\ TypeOfRel(r) = ASYNC \\
 & & \quad\quad \vee\ TypeOfRel(r) = ALIAS \\
 & & \quad\quad \vee\ TypeOfRel(r) = FORWARD\_FUNCTION \\
 & & \quad\quad \vee\ TypeOfRel(r) = APPLY\_FUNCTION \\
 & & \quad\quad \vee\ TypeOfRel(r) = PARAMETER\_MODULE)]] \\
 & & \\
\exists! m[(m \in ModuleObjs\ \wedge\ TypeOfModule(m) = ENV)]] & &
\end{array}
$$

Figure 5.10 Domain Model Evaluation Conditions.

derived from the domain model structure. Each module must have a unique name and must be one of the three valid module types (environment, submodule, or primitive). Only primitive modules may have parameters. Interfaces must be either in_ports or out_ports. Relations must be one of the six valid relation types (see Section 4.3.2). All source modules and target modules of the relations must exist (this condition not shown in Figure 5.10). One and only one module is the environment module. The domain model is complete whenever it passes these tests (i.e., i. is complete whenever it is consistent).

5.7.2 Evaluating the Tracking Knowledge Base. Evaluating the knowledge base files would involve checking several categories that certain conditions are true. The conditions are derived from the knowledge base structure (similar to conditions derived from domain model structure). The categories involved for this particular evaluation include the Specification Taxonomy, the Architecture Construction Rules, the Type Descriptions,

the Library Components Structure, and the Display Conventions. Ideally, the Knowledge Engineer would not have to evaluate the knowledge base files if the domain model file passes evaluation and a correct forward transformation is guaranteed.

In the Specification Taxonomy, there must be at least one level. Each level must have a unique name, a title string, and a flag symbolizing the level's initial state. A level may have any number of questions, but each question must have a string representing the information required. Questions may have a condition expression and must be one of four valid question types. `STACK` and `CHECKLIST` questions must have at least one response. Each response has a response string and may have actions. `NUMERIC` questions must have a total response range and may have some numeric responses. Each numeric response has a number range within the total range and may have actions. `TEXT` questions must have a maximum string size and may have actions. Each question type may have some default response (or responses for `CHECKLIST` questions). All actions must be one of the four valid types. Variable set actions must have a variable name and a value. Save value actions must have a variable name. Activate and deactivate level actions must have a level name that corresponds to an existing level.

In the Architecture Construction Rules, there must be at least one module. Each module must have a unique name and may have submodules, interface declarations, and relations. Submodules must have an instance name (not necessarily unique) and the name of an existing module or primitive. If the submodule is a primitive, then the submodule must have a list of parameter assignments (name and value pairs). Submodules may have condition expressions. Interface declarations must have a name (unique within the module), an interface type (either in or out port), and some initial value. Relations must have a name (unique within the module), a valid relation type, the names of existing source and target modules or interfaces, and some initial value (or module function).

The Type Descriptions must have at least one description. Each description must have a unique name and may have some description text. Names may identify existing modules or primitives. However, names may also identify global data types, interface types, or relation types.

The Library Component Structures must have the name of the domain and at least one scope (primitive module structure). Scopes must have a unique name and may have declarations and relations. Declarations must have a name (unique within the scope) and a valid data type or interface type. Declarations may have either an expression definition or a list of parameters. Relations must have a name (unique within the scope), a valid relation type, and the names of existing source and target modules or interfaces. Relations may have an expression definition. Notice that all the parameters identified within a submodule (in the Architecture Construction Rules) must have declarations.

The Display Conventions must have the name of the domain and at least one driver. Drivers may be either type display drivers, relation drivers, or constraint drivers. Type display drivers must have at least one display rule and a relation display flag. Display rules must have a unique name, an icon identifier, and an icon position. Display rules may have an icon label position, an icon label content, and some content display rule. Relation drivers must have a line width, source symbol, end symbol, an argument flag, and at least one relation pair (e.g. module to module, out_port to in_port). Relation drivers may have a label position and content, line color, as well as dash number and lengths.

One final evaluation checks whether variables tested in the question and submodule conditions exist within some variable set action. This particular check would complete the knowledge base evaluation.

5.7.3 Evaluating the Primitive Module Abstractions. No procedure automatically evaluates primitive module abstractions. Automatic evaluation requires the abstraction be available to a program. The abstraction technique involves functional and dynamic models that may exist only on paper or within some graphics package. Because there is no abstraction environment within Software Refinery, primitive module abstractions must be evaluated manually.

An evaluation procedure uses the structure of functional and dynamic models (as defined in the Rumbaugh Object Modeling Technique) to define consistency conditions. Each primitive module abstraction passes evaluation whenever all conditions are true. Because of the simplicity of the `INTERSECTING TRACKS GENERATOR` primitive module abstraction,

all the possible conditions were not necessary, nor were any identified. The abstraction was not formally evaluated.

5.7.4 Evaluating the Library Module Implementations. Evaluating primitive module implementations is similar to common program execution testing. Implementations are written in CIDL, and the APTAS system provides a CIDL programming environment (see Section 4.5.4). Obviously, the evaluation of a primitive module implementation fails if the module does not compile. Because many primitive modules may depend on data from other primitive modules, evaluation procedures may involve the creation of code "stubs" or test data files. However, the implementation would need no evaluation at all if the transformation was guaranteed to maintain the correctness of a primitive abstraction that passed its evaluation. An existing primitive module implementation was modified and, therefore, was not extensively tested.

5.8 Summary

Procedures were developed for each of the population process activities (some were automated, while others remain manual). The Create/Evolve Domain Model procedure involved building a domain model file using a specific domain model grammar. The Design Domain Taxonomy and Coding Knowledge Base procedure is entirely automated. It transforms the domain model file into several knowledge base files. The Abstract Primitive Module Behavior procedure involves using Rumbaugh's Object Modeling Technique to build functional and dynamic models for each primitive module identified in the domain model. The Implement Library Modules procedure involves programming the behavior abstractions in the CIDL programming language. These last two activities (abstraction and implementation) were identified as the most tedious tasks of the population process. The Evaluate Domain Development included two automatic procedures to check the domain model and knowledge base. The evaluation would also include procedures to test the abstractions and implementations. However, only the complexities of these test procedures were identified, and no formal procedures were developed.

VI. Research Conclusions

6.1 Objectives

Many knowledge-based software engineering capabilities depend on the information stored in the knowledge base. The knowledge base organizes information about a modeled application domain into several categories. All this information must either already exist in the knowledge base or some method must exist to store new information.

This thesis had one primary research objective: document and demonstrate a formal population process to solve the following problem:

> **There is currently no understood formal process to efficiently capture and store new domain information within the APTAS Knowledge Base.**

There were also two secondary objectives: study the APTAS Knowledge Base and outline a general population process that captures several important characteristics applying to many knowledge-based software engineering systems.

Section 6.2 summarizes the accomplishments of this thesis. Section 6.3 draws several conclusions concerning this thesis research. Section 6.4 suggests several research topics directly related to our accomplishments.

6.2 Accomplishments

This thesis proposed a general knowledge base population process outline that applies to many knowledge-based software engineering systems (38). Using the general process outline, a formal population process was developed to populate the knowledge base of APTAS. This formal process was documented and demonstrated by storing new information from the radar tracking domain into the knowledge base. Several procedures (some automatic and some manual) were implemented during the formal process demonstration. The development of the formal process also depended on relationships between knowledge categories and unique information stored in each category that was identified by studying the knowledge base of APTAS. This thesis:

1. **Proposed a general knowledge base population process outline** in Section 3.5. This process outline integrates and expands methods proposed by Prieto-Díaz (30, 31), Arango (3), McCain (25), and others. The outline consists of five activities that capture the three key stages of knowledge base population: Domain Analysis, Domain Implementation, and Evaluation. These three stages are described as follows:

    - *Domain Analysis*: A population process generally begins with the creation or evolution of a domain model using some domain modeling approach (Create/Evolve Domain Model activity) and the abstraction of component behaviors (Abstract Component Behaviors activity). The activities often vary depending on the type of application domain being analyzed and its complexity. The Domain Engineer can conceivably perform these two activities independent of the knowledge-based software engineering system requirements.

    - *Domain Implementation*: The process transforms specific structures in the domain model into corresponding structures in the knowledge base (Design Reuse Infrastructure activity). The process transforms the abstract component behavior definitions into reusable component implementations stored in the knowledge base (Implement Reusable Components activity). The Software Engineer performs these two activities to produce information that is stored directly in the knowledge-based software engineering system. These activities are therefore predominantly dependent of the system requirements.

    - *Evaluation and Feedback*: The process evaluates the results of the other four activities to determine whether any additional changes are necessary before using the new reuse infrastructure (Evaluate Domain Development activity). Both engineers (Domain Engineer and Software Engineer) use this activity to indicate inconsistencies in their activities results and to determine whether an activity is completed.

2. **Studied the APTAS Knowledge Base** in Section 4.3. The knowledge base consists of seven categories of information used by particular stages of application

composition. Each category contains unique information and has its own grammar (see Appendix B). The associations between the seven categories are repeated in Figure 6.1 with the unique information displayed in bold face text (CIDL implementations are unique to the Library Modules category, and code synthesis rules are unique to the Coding Rules category).

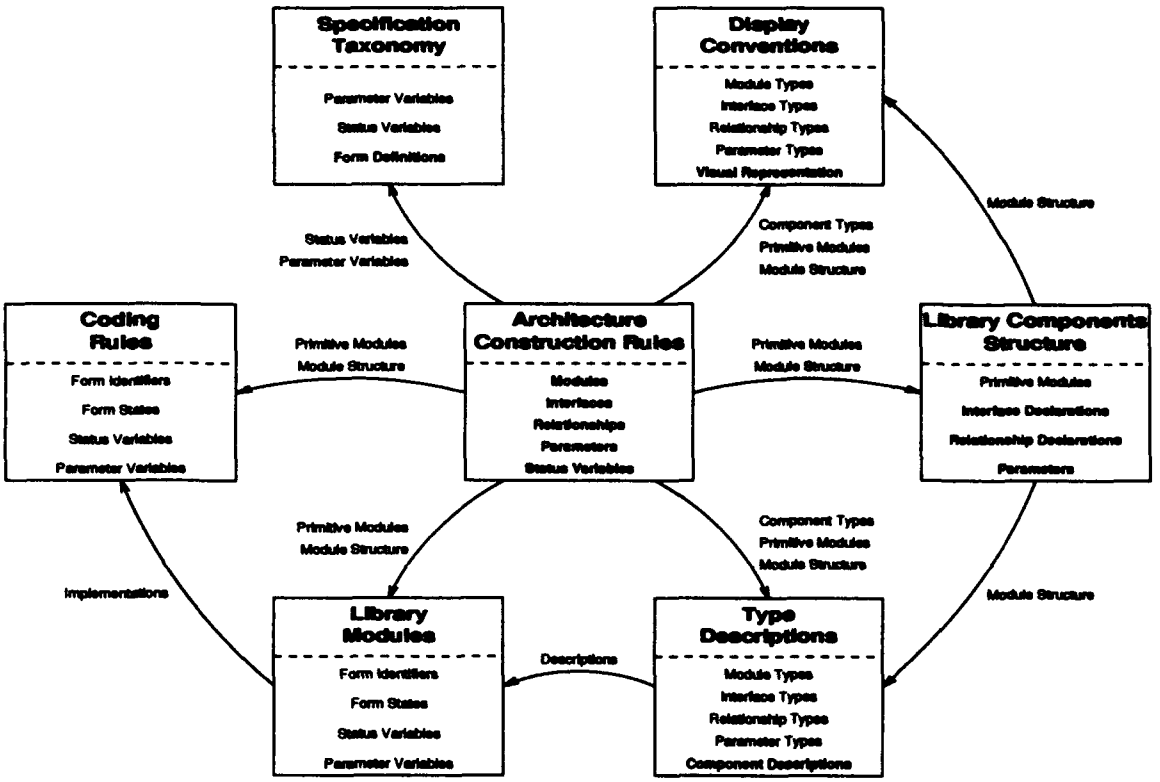


Figure 6.1 Association of Information Between Knowledge Base Categories.

3. **Documented a formal knowledge base population process for APTAS** consisting of five activities. The formal process was instantiated from the general process outline. Components were represented by primitive modules, the reuse infrastructure was represented by the Domain Taxonomy and Coding Knowledge Base, and reusable components were represented by library modules. An application domain was selected (the radar tracking domain) and scoped (track-while-scan, multiple-target tracking algorithms) to formalize the first two process activities. A description of the APTAS system and a definition of the knowledge base structure were used to formalize the second two process activities. The last activity was formalized

using the description of the four formalized activities. Section 3.5 described these five activities as follows:

- *Create/Evolve Domain Model*: This activity results in an object model of the radar tracking domain using the Rumbaugh Object Modeling Technique (34). Objects represent modules and elicitation forms. Modules are composed of several other objects, including parameters, interfaces, relations, submodules, and icons. Forms are composed of question objects. Domain Specialists perform this activity.

- *Abstract Primitive Module Behaviors*: This activity results in dynamic and functional models of primitive modules using the Rumbaugh Object Modeling Technique. Primitive modules are identified (using the domain model) as those modules that have parameters (and no submodules). Domain Specialists also perform this activity.

- *Design Domain Taxonomy and Coding Knowledge Base*: This activity results in five knowledge base files representing the information captured in the domain model. Domain model structures are transformed into their corresponding knowledge base structures using a set of mapping predicates. For instance, every primitive module identified in the domain model corresponds to a Library Component Structure in the knowledge base. The Knowledge Engineer performs this activity.

- *Implement Library Modules*: This activity results in CIDL implementations of the primitive modules. The dynamic and functional models are translated into CIDL, manually, using a programming environment provided by the APTAS system. The Knowledge Engineer performs this activity.

- *Evaluate Domain Development*: This activity provides feedback about the results of the other process activities. Domain Specialists receive evaluations on the consistency and completion status of the domain model and primitive module abstractions. The Knowledge Engineer receives evaluations on the consistency and completion status of the knowledge base files and primitive module

implementations. Procedures to evaluate the domain model and knowledge base files are easy to automate, while procedures to evaluate the primitive module abstractions and implementations are not easy to automate.

4. **Demonstrated the formal knowledge base population process** in Chapter V. Each activity of the formal process was implemented using information from the existing knowledge base, as well as some new domain information obtained from tracking experts. The demonstration:

   - *Created a tracking domain model.* The current contents of the Tracking Taxonomy and Coding Knowledge Base were reverse engineered into a tracking domain model. The formal process requires a domain model be created as the first step to knowledge base population. Because the knowledge base already contained information, the tracking domain model resulting from the reverse engineering directly supported this information. This reverse engineering procedure demonstrated part of the Create/Evolve Domain Model activity.

   - *Transformed the domain model into knowledge base files.* A set of predicates were developed that define the mapping between objects in the domain model and structures in the knowledge base. Using these predicates, the tracking domain model was transformed into its corresponding knowledge base files. Although not identical in style, the existing knowledge base files and those transformed from the tracking domain model were behaviorally equivalent. This transformation procedure demonstrated the Design Domain Taxonomy and Coding Knowledge Base activity.

   - *Evolved the tracking domain model* by adding new domain information. A primitive module was added by updating the tracking domain model file with a new primitive module description and an elicitation form that sets the primitive's parameters. An existing module and elicitation form were also modified to support the inclusion of the new primitive module. This update procedure demonstrated the other part of the Create/Evolve Domain Model activity.

- *Abstracted the behavior of a new primitive module.* A very simple functional model was developed to define the behavior of the primitive module added to the tracking domain model. This abstraction procedure demonstrated the Abstract Primitive Module Behaviors activity.
- *Implemented a new primitive module.* A CIDL implementation was developed to represent the functional model of the new primitive module. The knowledge base was updated with the new CIDL implementation. This implementation procedure demonstrated the Implement Library Modules activity.
- *Evaluated the new tracking domain model.* A set of predicates were developed to evaluate the consistency of the tracking domain model. The predicates were derived from the domain model structure. Automatic evaluation of the evolved tracking domain model was performed using a procedure based on these predicates. This evaluation procedure demonstrated the Evaluate Domain Development activity.

6.3 Conclusions

6.3.1 General Population Process Outline Conclusions. The following conclusions resulted from the general process outline:

1. **Object-oriented methods support the process.** The general process was developed for a class of systems that have an object-oriented knowledge base (Section 3.2). Therefore, object-oriented software development methods can support domain implementation (Design Reuse Infrastructure and Implement Reusable Components activities). As the formal process demonstrated, object-oriented techniques also support domain analysis (Create/Evolve Domain Model and Abstract Component Behaviors activities). Building and evolving the domain model was similar to creating an object model for a single application (although at a much larger scale). Abstracting component behaviors and reusable component implementations was identical to creating dynamic and functional models for a single application.

2. **Process conceivably benefits from improvements to software development methods.** The process does not mandate a particular development method and is therefore not limited to an existing method. A population processes instantiated from the general process outline can conceivably accommodate improvements to existing methods (e.g., improvements to Rumbaugh's technique). The process can conceivably accommodate new software development methods (e.g., new object-oriented methods).

3. **Domains can be developed incrementally.** Separating the domain analysis into two activities allows the performance of each activity to progress at a different pace. The tracking domain model was not required to have all the dynamic and functional models (i.e., primitive module behavior definitions) before being transformed into knowledge base files. Therefore, domains with a large set of components would not require all their definitions prior to designing a reuse infrastructure. Component behaviors could be abstracted as required.

4. **Domains can be transferred between systems.** The general process outline separates the results of domain analysis from the particular system requirements. Therefore, the domain model and component abstractions generated during domain analysis could be used by other systems. A different set of predicates to transform the tracking domain model could conceivably produce knowledge base files for a system other than APTAS.

6.3.2 Formal Population Process Conclusions. The following conclusions resulted from the development and demonstration of the formal knowledge base population process:

1. **Population effort can be reduced.** The transformation procedure demonstrated that information repeated in several categories of the Tracking Taxonomy and Coding Knowledge Base could be automatically reproduced from the tracking domain model. Therefore, the Domain Specialists only need to add or modify the information once (in the domain model).

2. **Process activities have different levels of effort.** The evolution of the tracking domain model demonstrated that the level of effort depends partly on the specific

population process activity. Adding the new primitive to the tracking domain model was relatively simple (as was generating the corresponding knowledge base files). However, abstracting the primitive module behavior and implementing it in CIDL required much more effort.

3. **Process can enforce standards.** The transformation procedure demonstrated that standard display conventions, although not strictly enforced by the knowledge base structure, can be enforced during transformation. The primitive module added during the evolution of the tracking domain model was transformed with a standard display representation in the Display Conventions category of the knowledge base.

6.4 Future Knowledge Base Population Research Topics

Future research of the knowledge base population problem should include the following topics:

1. **Further define and refine the general process.** Although a general process was developed, additional research could decompose each of the activities in more detail. For instance, the Create/Evolve Domain Model activity might decompose into more concrete stages defining model structure, component hierarchy, and component relationships. Decomposing each activity might produce additional constraints to the applicability of the process on certain domains and systems.

2. **Do domain analysis on simple application domains.** The population process requires expert knowledge of an application domain. Domain information should originate from an expert through a formal knowledge acquisition process. Some application domains may require only a few components to sufficiently build applications (e.g., logic circuits could be built using only simple logic gates). When developing the domain model and abstracting the component behaviors of these more simple domains, the first two population activities could be combined into a single domain analysis activity. Such domains would be ideal research areas to develop domain modeling and component behavior abstraction tools.

3. **Develop knowledge base population tools.** Additional research into knowledge base population could result in the development of more population tools and utilities. Cecil and Fullenkamp (6) observed that an Object-Oriented Database Management System could archive domain artifacts (e.g., reuse infrastructures and component implementations). As more object-oriented utilities are developed, software engineers could integrate them into the knowledge base population process. Utilities that capture object models could help construct and evolve domain models and could help transform models into reuse infrastructures. Utilities that capture functional and dynamic models could help define component behaviors and could even help implement components.

4. **Build application test sets.** It is no small matter to make sure applications still execute as expected when making changes to the knowledge base files. A set of applications that test the operation of the knowledge base over significant characteristics should be maintained. As the Knowledge Engineer makes changes to the knowledge base files, new applications that test any new characteristics should be added to the existing test application set. Using the changed knowledge base files to successfully synthesize code and simulate expected behaviors for that application test set would significantly boost the confidence in the new files. Maintaining such a test set would be easier if done in conjunction with knowledge base updates. However, developing and supporting the initial test set is a major undertaking and could be an area of future research efforts.

5. **Transfer domains between systems.** As mentioned in the conclusions above, domain analysis results can be independent of system requirements. Therefore, the results of a domain analysis can be transferred between two or more systems. For instance, the information captured by the tracking domain model should transfer to another system (like Architect). This knowledge transfer could result in better domain models and more generic transformation procedures.

6.5 Final Remarks

Overall, this research effort identified several of the general characteristics of knowledge base population. Although the general process was developed for Domain-Oriented Application Composition Systems, the concepts introduced by this research could apply to other knowledge-based software engineering systems.

Appendix A. Domain Model

A.1 Domain Model Grammar

The first activity of the knowledge base population process (developed in Section 4.5.1) involves the creation and evolution of a domain model. A formal domain model structure is required as part of this activity. The ideal domain model structure would describe a domain language. The grammar in Figure A.1 represents the domain model structure.

```
DomainModel        ::= Name "model" "structure" ":" ModuleObj+
                       {"elicitation" ":" FormObj+}
ModuleObj          ::= "module" Name "(" TypeOfModule ")"
                       {"description" ":" DescOfModule}
                       {"icon" ":" IconObj} {"parameters" ":" ParameterObj+}
                       {"interfaces" ":" InterfaceObj+}
                       {"relations" ":" RelationObj+}
                       {"submodules" ":" SubModRel+} "end"
IconObj            ::= IconDesc
ParameterObj       ::= Name "(" TypeOfParam ")" {"=" ParamValue}
InterfaceObj       ::= Name "(" TypeOfInterf ")" {"=" InterfValue}
RelationObj        ::= Name "(" TypeOfRel {":" RelationValue} ")"
                       "from" ":" SourceComp "to" ":" TargetComp
SubModRel          ::= Name "is-a" SubModType {"guard" ":" SubGuard}
FormObj            ::= "form" Name {"[" "initialize" "]"} "title" ":" TitleOfForm
                       {"questions" ":" QuestionObj+} "end"
QuestionObj        ::= TextQuestion | NumericQuestion | StackQuestion |
                       ChecklistQuestion
TextQuestion       ::= "text" ":" InfoElicited MaxLength {"guard" ":" QuestCond}
                       {"default" ":" DefaultChoice} {"actions" ":" ActionObj+}
NumericQuestion    ::= "numeric" ":" InfoElicited "[" RangeMin "," RangeMax "]"
                       {"guard" ":" QuestCond} {"default" ":" DefaultChoice}
                       {"responses" ":" ResponseObj+}
StackQuestion      ::= "stack" ":" InfoElicited {"guard" ":" QuestCond}
                       {"default" ":" DefaultChoice}
                       {"responses" ":" ResponseObj+}
ChecklistQuestion  ::= "checklist" ":" InfoElicited {"guard" ":" QuestCond}
                       {"default" ":" "(" DefaultChoices+ ")"}
                       {"responses" ":" ResponseObj+}
ResponseObj        ::= ResponseStr {"actions" ":" ActionObj+}
ResponseObj        ::= "[" RespMin "," RespMax "]" {"actions" ":" ActionObj+}
ActionObj          ::= VariableSetAction | SaveValueAction |
                       ActiviateAction | DeactivateAction
VariableSetAction  ::= "variable_set" "(" ActionVar "," ActionVal ")"
SaveValueAction    ::= "save_value" "(" ActionVar ")"
ActivateAction     ::= "activate_level" "(" ActionLevel ")"
DeactivateAction   ::= "deactivate_level" "(" ActionLevel ")"
```

Figure A.1 Domain Model Grammar.

A domain language was developed with the DIALECT tool using the domain model grammar. This particular domain language is not related to the attributes of a specific domain and could be used as a language to model many different domains (particularly when modeling domains for the APTAS system).

A.2 Creating the Initial Domain Model File

As described in Section 5.2, reverse engineering the knowledge base files into a domain model consisted of the three functions shown in Figure A.2. The first function uses

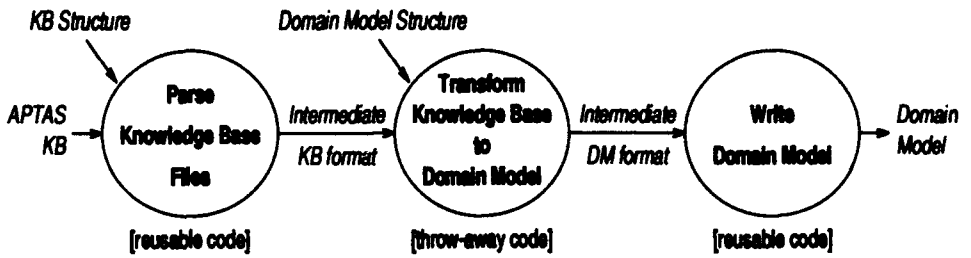


Figure A.2 Reverse Engineering Procedure.

four grammars to create an abstract syntax tree in the REFINE object base. The complete knowledge base grammars are defined in Appendix B. The second function uses the knowledge base syntax tree to create a domain model abstract syntax tree. The structure of the domain model abstract syntax tree uses the grammar defined in Figure A.1. The third function creates a domain model file from its abstract syntax tree.

A REFINE procedure reverse engineers the knowledge base files. The procedure loads the four knowledge base grammars, loads the reverse transformation procedures, and creates the domain model file. However, explicit pathnames and filenames were used within the REFINE code (e.g., resulting domain model file is `tracker-domain.txt`). One should modify the reverse transformation procedure to reflect the desired pathnames and filenames. If you are interested in the REFINE code, see Appendix C.

A.2.1 Parse Knowledge Base Files. Four grammars, each described in Appendix B, were used to parse the necessary information from the knowledge base files. The first knowledge base grammar defines the Specification Taxonomy and Architecture

Construction Rules. Using this grammar and the information stored in the `global.form` file, one may parse the sequence of form (`Level`) structures and the sequence of `module` structures into the object base.

The second knowledge base grammar defines the Library Component Structures. Using this grammar and the information stored in the `global.gsdl-l` file, one may parse the sequence of primitive module (`Scope`) structures into the object base.

The third knowledge base grammar defines the Display Conventions. Using this grammar and the information stored in the `global.gsdl-t` file, one may parse the sequence of display convention (`Driver`) structures into the object base.

The fourth knowledge base grammar defines the Type Descriptions. Using this grammar and the information stored in the `global.desc` file, one may parse the sequence of type description (`Desc`) structures into the object base.

A.2.2 Reverse Transformation. With the four knowledge base files parsed into the REFINE object base, the knowledge base abstract syntax tree can be transformed into a domain model abstract syntax tree. This transformation creates a new domain model abstract syntax tree instead of modifying the knowledge base abstract syntax tree. The reverse transformation is based on the conditions described in Section 5.2 and must use the domain model grammar and the knowledge base grammars.

A.2.3 Write Domain Model File. The domain model abstract syntax tree can be output into the domain model file. The output function must ensure the file resulting from the domain model abstract syntax tree corresponds to the domain model grammar. The resulting file is the initial domain model and can be evolved using the domain model grammar.

A.3 Evolving the Domain Model

Performing the evolution to the tracking domain model involved defining the new information, identifying the necessary changes, and making them. The population process was demonstrated by adding a new primitive module. The new primitive, called

INTERSECTING TRACKS GENERATOR, provides two targets with intersecting tracks. The Tracking Engineer may select the original SENSOR MODEL primitive, by default, or this new primitive. Several of the necessary changes were identified.

A.3.1 Update TRACKER_ENVIRONMENT *Module Definition.* The initial domain model file defines the TRACKER_ENVIRONMENT module with a SENSOR_DATA submodule of SENSOR_MODEL type. This original module architecture is shown in Figure A.3. This definition was modified to include the option of specifying a SENSOR_DATA submodule of INTERSECTING_TRACKS_GENERATOR type. The modified module architecture is shown in Figure A.4 (changes are highlighted with -- <<<).

```
MODULE TRACKER_ENVIRONMENT (ENVIRONMENT)
  Relations:
    SENSOR_SCAN_FRAME_TO_TRACKER (ASYNC : ~"GenericScanFrame")
       from:  Sensor_Data.scan_frame_out   to:  Tracker.scan_frame_in
    OPERATOR_QUERY_TO_TRACKER (ASYNC : ~"GenericQuery")
       from:  Output.query_out   to:  Tracker.user_query_in
    TRACKER_DATA_TO_DISPLAY (ASYNC : ~"GenericTrackData")
       from:  Tracker.display_data_out   to:  Output.reply_in
  SubModules:
    SENSOR_DATA is-a SENSOR_MODEL
    TRACKER is-a TARGET_TRACKER
      Guard:  DEFAULT_TRACKER != "false"
    TRACKER is-a NEW_TRACKER
      Guard:  DEFAULT_TRACKER != "true"
    OUTPUT is-a OUTPUT_DISPLAY
END
```

Figure A.3 Original TRACKER_ENVIRONMENT Module Architecture.

Note that a new variable has been defined, DEFAULT_TARGET, that determines which submodule type has been selected. This variable must be set using a question form (i.e., need to modify the elicitation section of the domain model file).

A.3.2 Update TRACKING *Elicitation Form.* A new question was added to the TRACKING elicitation form to set the DEFAULT_TARGET variable. The default selection results in the original selection (i.e., SENSOR_MODEL type). The modification is represented in Figure A.5.

```
MODULE TRACKER_ENVIRONMENT (ENVIRONMENT)
                          ...

  SubModules:
    SENSOR_DATA is-a SENSOR_MODEL                          -- <<<
      Guard:  DEFAULT_TARGET != "false"                    -- <<<
    SENSOR_DATA is-a INTERSECTING_TRACKS_GENERATOR         -- <<<
      Guard:  DEFAULT_TARGET = "false"                     -- <<<
    TRACKER is-a TARGET_TRACKER
      Guard:  DEFAULT_TRACKER != "false"
    TRACKER is-a NEW_TRACKER
      Guard:  DEFAULT_TRACKER != "true"
    OUTPUT is-a OUTPUT_DISPLAY
END
```

Figure A.4 New TRACKER_ENVIRONMENT Module Architecture.

```
FORM /TRACKING [INITIALIZE]
                          ...

    STACK: "Default Target Model?"                         --- <<<
      Default:  0                                          --- <<<
      Responses:                                           --- <<<
        "true"                                             --- <<<
          Actions:                                         --- <<<
            variable_set(DEFAULT_TARGET, "true")           --- <<<
            deactivate_level(/TRACKING/TARGET_MODELS)      --- <<<
        "false"                                            --- <<<
          Actions:                                         --- <<<
            variable_set(DEFAULT_TARGET, "false")          --- <<<
            activate_level(/TRACKING/TARGET_MODELS)        --- <<<
                          ...
END
```

Figure A.5 New Question in TRACKER Form.

Note that a new form (level) was also created to further define the specific behavior of the new primitive module. However, a primitive module structure is needed before defining the new form.

A.3.3 Create INTERSECTING_TRACKS_GENERATOR *Module Definition.* The new primitive module must have a definition in the domain model file. The definition given in Section 5.4 is repeated in Figure A.6.

This definition introduces ten integer parameters (e.g., X_COORD_T1) that must be set by the Tracking Engineer to specify the behavior of the primitive module. These

```
MODULE INTERSECTING_TRACKS_GENERATOR (PRIMITIVE)
  Description:  "This module implements Lang Hong's routine for
                 generating noisy target data with false alarms.
                 This module first generates two perfect
                 intersecting tracks and then adds noise with a
                 normal distribution to obtain more realistic
                 target observations.  Uniformly distributed
                 false alarms within predetermined gates about
                 the track are then added."
  Icon: MED_BLUE_RECT
  Parameters:
    X_COORD_T1 (INT) = X_T1
    Y_COORD_T1 (INT) = Y_T1
    X_COORD_T2 (INT) = X_T2
    Y_COORD_T2 (INT) = Y_T2
    X_VEL_T1 (INT) = X_VEL
    Y_VEL_T1 (INT) = Y_VEL
    ITERATIONS (INT) = TEST_ITERATIONS
    X_NOISE_VARIANCE (INT) = X_NOISE
    Y_NOISE_VARIANCE (INT) = Y_NOISE
    MAX_FALSE_ALARMS (INT) = MAX_ALARMS
  Interfaces:
    SCAN_FRAME_OUT (OUT_PORT)
END
```

Figure A.6 INTERSECTING TRACKS GENERATOR Primitive Module Structure.

parameters are directly related to the abstract behavior definition and functional implementation of the primitive module. The coordinate parameter values provide the true location of each target. The velocity parameter values provide the necessary information to calculate each target's subsequent locations. Other parameter values provide the number of positions to calculate, the variance between reported location and true location, and the number of possible false targets generated.

A.3.4 Create TRACKINGTARGET_MODELS *Elicitation Form.* The new elicitation form is defined in Figure A.7. Nine of the ten parameters are set by responses made through this form (the value of the tenth parameter, the ITERATIONS, is set through the TRACKING form). Each of these parameters are integer values that would ideally range from $-\infty$ to $+\infty$ (except for noise and false alarms); however, APTAS requires a finite range specification.

```
FORM /TRACKING/TARGET_MODELS
  Title:  "Intersecting Target Characteristics"
  Questions:
    NUMERIC : "X Coordinate of Target 1" [-99999, 99999]
      Default:  2000
      Responses:
        [-99999, 99999]  Actions:  save_value(X_T1)
    NUMERIC : "Y Coordinate of Target 1" [-99999, 99999]
      Default:  11000
      Responses:
        [-99999, 99999]  Actions:  save_value(Y_T1)
    NUMERIC : "X Coordinate of Target 2" [-99999, 99999]
      Default:  2000
      Responses:
        [-99999, 99999]  Actions:  save_value(X_T2)
    NUMERIC : "Y Coordinate of Target 2" [-99999, 99999]
      Default:  10000
      Responses:
        [-99999, 99999]  Actions:  save_value(Y_T2)
    NUMERIC : "X Velocity for both Targets" [-99999, 99999]
      Default:  25
      Responses:
        [-99999, 99999]  Actions:  save_value(X_VEL)
    NUMERIC : "Y Velocity both Targets" [-99999, 99999]
      Default:  5
      Responses:
        [-99999, 99999]  Actions:  save_value(Y_VEL)
    NUMERIC : "X Noise" [0, 99999]
      Default:  100
      Responses:
        [0, 99999]  Actions:  save_value(X_NOISE)
    NUMERIC : "Y Noise" [0, 99999]
      Default:  15000
      Responses:
        [0, 99999]  Actions:  save_value(Y_NOISE)
    NUMERIC : "Maximum Number of False Alarms (per scan)" [0, 99]
      Default: 3
      Responses:
        [0, 99]  Actions:  save_value(MAX_ALARMS)
END
```

Figure A.7 TRACKINGTARGET_MODELS Form Definition.

A.4 Evaluating the Domain Model

Only the actions involved with creating the initial domain model and adding new information have been demonstrated. The process demonstration was unique because it started with information already in the knowledge base that had no corresponding domain model. Creating a domain model from scratch (i.e., no existing knowledge base files) would involve developing a brand new domain model file that follows the domain model grammar.

Whenever the Domain Specialists are satisfied with the state of the domain model file, they should evaluate it for consistency to the domain model structure. Several predicates were defined in Section 5.7.1 that must be true for the domain model file to pass its evaluation. These conditions are repeated in Figure A.8.

$$\forall m[m \in ModuleObjs \implies \neg\exists o[o \in ModuleObjs \wedge o \neq m \wedge Name(o) = Name(m)]]$$

$$\forall m[m \in ModuleObjs \implies (TypeOfModule(m) = ENV \vee TypeOfModule(m) = SUB \vee TypeOfModule(m) = PRI)]$$

$$\forall m[(m \in ModuleObjs \wedge TypeOfModule(m) \neq PRI) \implies Parameters(m) = \{\}]$$

$$\forall m[m \in ModuleObjs \implies \forall i[i \in Interfaces(m) \wedge (TypeOfInterf(i) = IN\_PORT \vee TypeOfInterf(i) = OUT\_PORT)]]$$

$$\forall m[m \in ModuleObjs \implies \forall r[r \in Relations(m) \wedge (TypeOfRel(r) = SYNC \vee TypeOfRel(r) = ASYNC \vee TypeOfRel(r) = ALIAS \vee TypeOfRel(r) = FORWARD\_FUNCTION \vee TypeOfRel(r) = APPLY\_FUNCTION \vee TypeOfRel(r) = PARAMETER\_MODULE)]]$$

$$\exists! m[(m \in ModuleObjs \wedge TypeOfModule(m) = ENV)]]$$

Figure A.8 Domain Model Evaluation Conditions.

When the domain model passes its evaluation, one may transform the model into its corresponding knowledge base files. Appendix B describes the procedure used to transform a domain model file into its corresponding knowledge base files.

Appendix B. APTAS and its Knowledge Base

B.1 Knowledge Base Grammars

Section 4.3 described the knowledge base structure in great detail. Five of the seven knowledge base categories were directly related to information captured in the domain model. An object model of these five categories, showing their structure, is shown in Figure B.1. There is a great deal of information captured in this picture of the knowledge

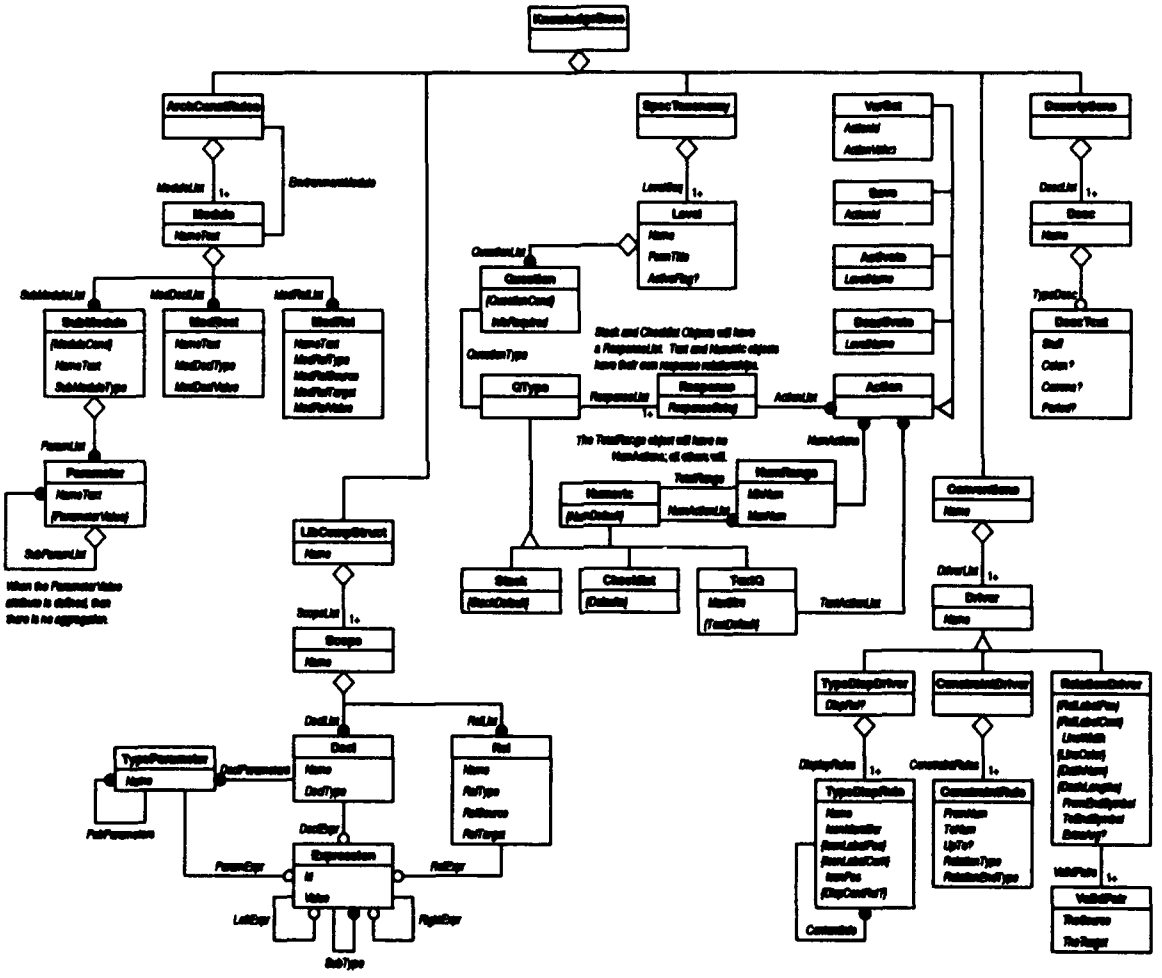


Figure B.1 APTAS Knowledge Base Structure.

base structure. Each category has its own structure, and except for two categories that share a grammar, each has its own grammar. The Architecture Construction Rules and Specification Taxonomy share a grammar because their information is stored in the same

knowledge base file. Each of these grammars was developed using the DIALECT tool of REFINE. See Appendix C for the actual implementation of these grammars.

B.1.1 Specification Taxonomy and Architecture Construction Rules. As described in Sections 4.3.1 and 4.3.2, the Specification Taxonomy and Architecture Construction Rules categories have a formal structure. This structure is captured in the grammar shown in Figure B.2 (a similar grammar appears in (20:38-39).

```
TaxonomyArchitecture ::= "levels" Level+ "modules" Module+

Level                ::= Name FormTitle ("true" | "false") {QuestionList+} "end"
Question             ::= {Condition} InfoRequired+ QuestionType
QuestionType         ::= Stack | TextQ | Numeric | Checklist
Stack                ::= "stack" Response+ {Value}
TextQ                ::= "text" MaxSize {Action+} {Value}
Numeric              ::= "numeric" NumRange {NumAction+} {Value}
Checklist            ::= "checklist" Response+ {Value+}
Response             ::= ResponseString {Action+}
NumAction            ::= NumRange Action+
NumRange             ::= "[" MinNum "," MaxNum "]"
VarSet               ::= "variable_set" "(" ActionId "," Value ")"
Save                 ::= "save_value" "(" ActionId ")"
Activate             ::= "activate_level" "(" LevelName ")"
Deactivate           ::= "deactivate_level" "(" LevelName ")"

Module               ::= NameText {SubModule+} {ModDecl+} {ModRel+}
SubModule            ::= {Condition} NameText SubModuleType {"{" Parameter+ "}"}
Parameter            ::= NameText "=" (Value | "{" Parameter+ "}")
ModDecl              ::= "decl" "(" NameText "," ModDeclType "," Value ")"
ModRel               ::= "rel" "(" NameText "," ModRelType ","
                          ModRelSource "," ModRelTarget "," Value ")"

Value                ::= SubValue | StringLiteral | "~" StringLiteral
SubValue             ::= IntNum | RealNum | Identifier
Condition            ::= SimpleCond "&" SimpleCond | Condition "|" SimpleCond
SimpleCond           ::= IsEqual | NotEqual | LessThan | GreaterThan |
                         LessThanOrEqual | GreaterThanOrEqual
IsEqual              ::= Value "eq" Value
NotEqual             ::= Value "ne" Value
LessThan             ::= Value "lt" Value
GreaterThan          ::= Value "gt" Value
LessThanOrEqual      ::= Value "le" Value
GreaterThanOrEqual   ::= Value "ge" Value
```

Figure B.2 Specification Taxonomy and Architecture Construction Rules Grammar.

B.1.2 Library Component Structures. As described in Section 4.3.5, the Library Component Structures category also defines a formal grammar shown in Figure B.3 (a similar grammar appears in (20:40-41).

```
LibCompStruct   ::= Name Scope+
Scope           ::= Name ":" "module" "=" {"declare" Decl+} {"relation" Rel+} "end"
Decl            ::= Name ":" ("type" "=" BaseType |
                              BaseType {"{" TypeParameter+ "}"} {"=" Expr})
TypeParameter   ::= Name "=" (Expr | "{" TypeParameter+ "}")
Rel             ::= Name ":" BaseType "(" RelSource+ "," RelTarget+ ")" {"=" Expr}
Expr            ::= NumericExpr | BoolExpr | String | ScopeReference |
                    FunctionExpr | AtExpr | RecordExpr | BaseType
NumericExpr     ::= Int | Real | NegativeExpr | AddExpr | SubExpr |
                    MulExpr | DivExpr
NegativeExpr    ::= "-" Expr
AddExpr         ::= Expr "+" Expr
SubExpr         ::= Expr "-" Expr
MulExpr         ::= Expr "*" Expr
DivExpr         ::= Expr "/" Expr
BoolExpr        ::= "true" | "false" | NotExpr | AndExpr | OrExpr | GtExpr |
                    LtExpr | EqExpr | LeExpr | GeExpr | NeExpr
NotExpr         ::= "not" Expr
AndExpr         ::= Expr "and" Expr
OrExpr          ::= Expr "or" Expr
GtExpr          ::= Expr ">" Expr
LtExpr          ::= Expr "<" Expr
EqExpr          ::= Expr "=" Expr
LeExpr          ::= Expr "le" Expr
GeExpr          ::= Expr "ge" Expr
NeExpr          ::= Expr "ne" Expr
ScopeReference  ::= Identifier++
FunctionExpr    ::= Name "(" Expr+ ")"
AtExpr          ::= Expr "@" Identifier
RecordExpr      ::= "[" Expr+ "]"
BaseType        ::= Identifier | RecordType | SequenceType | SetType
RecordType      ::= "record" "<" RecordField+ ">"
RecordField     ::= Name {":" BaseType} {"=" Expr}
SequenceType    ::= "sequence" "[" BaseType "]"
SetType         ::= "set" "(" BaseType ")"
```

Figure B.3 Library Component Structures Grammar.

B.1.3 Display Conventions. As described in Section 4.3.3, the Display Conventions category has a formal structure. This structure is captured with the grammar shown in Figure B.4 (a similar grammar appears in (20:40,41-42). There is an established display convention standard for visual components defined using this category. APTAS uses interconnected icons to represent an application architecture. This architecture represents

```
Conventions       ::= "table" Name TypeDispDriver+ "end"
TypeDispDriver    ::= Name ":" "type" TypeDispRule+  ("display" | "hide")
                      "relations" ";"
TypeDispRule      ::= Name "->" "icon" "=" Identifier
                      {"label" "=" IconLabelPos IconLabelContent}
                      "default_position" "=" IconPos
                      {"contents" "=" ContentInfo+ ("display" | "hide") "relations"}
RelationDriver    ::= Name ":" "relation" {"label" "=" RelLabelPos RelLabelContent}
                      "width" "=" LineWidth {"color" "=" LineColor}
                      {"dash_count" "=" DashNum "dash_list" "=" DashLengths+}
                      "from_end" "=" FromEndSymbol "to_end" "=" ToEndSymbol
                      "valid_pairs" "=" ValidPair+ {"extra_arg"} ";"
ValidPair         ::= "(" TheSource "," TheTarget ")"
ConstraintDriver  ::= Name ":" "constraint" ConstraintRule+ ";"
ConstraintRule    ::= "[" FromNum "," (ToNum | "+") "]"
                      "relation" RelationType RelationEndType
```

Figure B.4 Display Convention Grammar.

the functional modules, their associated parameters and interfaces, and their relationships to other modules. The knowledge base stores the specific icons used and their relative display position on an architecture display. Shape, color, and position define what an icon represents (20:8,9).

Boxes refer to functional modules. Blue boxes represent primitive modules, which have reusable implementations in the knowledge base. Black boxes represent modules composed of one or more submodules. Selecting a black box opens another canvas displaying its submodules and their relationships. Orange boxes represent submodules that provide interface functions to their parent module.

Circles and diamonds refer to a module's interfaces. Circles appearing on the left side of a module represent incoming data (in_port). Circles appearing on the right side of a module represent outgoing data (out_port). Diamonds refer to interface functions available to parent modules. Color has no current importance; however, one might use color to represent specific data types.

Lines refer to relationships between modules. Black lines connecting two circles represent a synchronous communication (handshake) between modules. Pink lines connecting two circles represent an asynchronous communication (no handshake) between modules. Red lines connecting a circle to a box represent an alias relation where data transmits

between a submodule and the interface. Orange lines connecting an orange box to another box represent a parameter module relationship where the parent module accesses the interface functions of the other submodule. Pink lines connecting a circle to a diamond represent apply function relationships where the interface functions execute on incoming data. Pink lines connecting a diamond to a circle represent a forward function result relationship where outgoing data transmits the results of interface functions.

B.1.4 Type Descriptions. As described in Section 4.3.4, the Type Descriptions category also has a formal structure. This structure is captured with the grammar shown in Figure B.4.

```
Descriptions ::= Desc+ ".:"
Desc         ::= Name ":" {DescText+} "@"
DescText     ::= Text {":" | "," | "."}
```

Figure B.5 Type Description Grammar.

B.2 Transforming the Domain Model into the Knowledge Base

Transforming an evaluated domain model file into its corresponding knowledge base files consisted of two functions. The first function parses the domain model file (using the grammar from Figure A.1) into the REFINE object base. The second function writes a set of knowledge base files based on the state of the object base. The actual transformation was described in Section 5.3. The transformation predicates are repeated in Figure B.6. To see the REFINE code that performs this transformation, refer to Appendix C.

B.3 Capabilities and Limitations of APTAS

The research of the APTAS composition process uncovered several powerful capabilities (and some limitations). Some of these capabilities include specification elicitation, design construction, rapid prototype simulation, and code generation.

The system elicits specification information through form questions and maps the set of responses to a corresponding application architecture. Information required to specify an application is separate from the information used to build an application architecture.

$$
\begin{array}{lll}
\forall f \ [f \in FormObjs & \Longrightarrow \ \exists s \ [s \in SpecTaxonomy \\
 & \quad \wedge \ Name(s) = Name(f) \\
 & \quad \wedge \ FormTitle(s) = TitleOfForm(f) \\
 & \quad \wedge \ ActiveFlag?(s) = Initiate?(f) \\
 & \quad \wedge \ QuestionList(s) = Questions(f)]] \\
 & \\
\forall m \ [(m \in ModuleObjs & \\
\quad \& \ TypeOfModule(m) \neq PRI) & \Longrightarrow \ \exists r \ [r \in ArchConstRule \\
 & \quad \wedge \ NameText(r) = Name(m) \\
 & \quad \wedge \ SubModuleList(r) = SubModRels(m) \\
 & \quad \wedge \ ModRelList(r) = Relations(m) \\
 & \quad \wedge \ ModDeclList(r) = Interfaces(m) \\
 & \quad \wedge \ (TypeOfModule(m) = ENV \Rightarrow \\
 & \qquad r = first(ArchConstRule))]] \\
 & \\
\forall m \ [(m \in ModuleObjs & \\
\quad \& \ TypeOfModule(m) = PRI) & \Longrightarrow \ \exists l \ [l \in LibCompStruct \\
 & \quad \wedge \ Name(l) = Name(m) \\
 & \quad \wedge \ DeclList(l) = Interfaces(m) \cup Parameters(m) \\
 & \quad \wedge \ RelList(l) = Relations(m)]] \\
 & \\
\forall m \ [m \in ModuleObjs & \Longrightarrow \ \exists d \ [d \in Descriptions \\
 & \quad \wedge \ Name(d) = Name(m) \\
 & \quad \wedge \ TypeDesc(d) = DescOfModule(m)]] \\
 & \\
\forall m \ [m \in ModuleObjs & \Longrightarrow \ \exists c \ [c \in Conventions \\
 & \quad \wedge \ Name(c) = Name(m) \\
 & \quad \wedge \ IconIdentifier(c) = ModuleIcon(m)]]
\end{array}
$$

Figure B.6 Forward Transformation Postconditions.

The current state of a specification is identified using the forms stored in the Specification Taxonomy. This category describes a powerful hierarchy similar to an abstract syntax tree, with each node representing a specific form. Each form has a level identifier (very similar to a pathname). The first form in the category is considered the root. Each form may activate any of its descendant forms. However, all of a form's ancestors must also be activated for the form to become available. The power of this category has a frustrating limitation. A form cannot be used in a loop to specify information for a set of similar components. For example, suppose a form sets up the attributes of a single target, and an application may have any number of targets. If each target has different attributes, there must be a separate form or set of questions for every component. Adding the capability to loop through a set of questions to define a sequence of response sets would resolve this limitation in eliciting specification information.

The system automatically transforms a specification into its corresponding architecture by "triggering" a sequence of rules in the Architecture Construction Rules. This category, like the Specification Taxonomy, describes a powerful hierarchy similar to an abstract syntax tree, with every node representing a specific module and every leaf representing primitive modules. The first module in this category is considered the root and is the only rule that *always* "triggers" (regardless of specification information). This "environment" rule APTAS could be used to set up an initial design if no forms are defined in the Specification Taxonomy. The same loop limitation occurs with this category. Using the example above, one must define a different module for every target even though they are all the same except for their attribute values. Adding the loop capability would greatly improve architecture construction.

Another limitation of APTAS is that the system has no capability to transform an architecture design into its corresponding specification information. In fact, the system only saves architecture designs (the application specification is never saved and the system has no capability to save them). Therefore, the Tracking Engineer must respond to question forms all over again to "playback" an application's specifications. There should be some capability added to the system to save a specification state.

The system has a Graphical User Interface to display this architecture and to provide an environment to add design details to the displayed architecture. Three knowledge base categories capture support information used to display and modify architectures. The Display Conventions category defines the visual representation for each module (component), for each parameter (component attribute), and for each relationship type (component relationships). Display standards were declared in the APTAS documentation (20:8-9). No documentation was found on how to build new visual representations (i.e., icons). The Library Component Structures category defines the attribute (parameter and interface) names and types for primitive modules. The value of an interface or parameter can be changed or initialized using the Graphical User Interface. The Type Descriptions category describes the modules, interface types, parameter types, and relationship types. Description information is a form of documentation and, although helpful, is not required. Most of

the information in these three categories can be derived from the Architecture Construction Rules.

The system uses an architecture to synthesize CIDL code using the synthesis engine. This engine uses two other knowledge base categories, as well as the Synthesis Support information. The impact of the Coding Rules category was not studied. The Library Modules category provides CIDL implementations of primitive modules. Each primitive module implementation must support the parameters, interfaces, and relationships declared in the Architecture Construction Rules and the Library Component Structures. The CIDL programming language was a significant limitation to the APTAS system. Implementing primitive modules in CIDL is not easy unless the programmer is *very* familiar with both CIDL and the APTAS system. CIDL supports parallel processing and could be argued to support object-oriented paradigms. However, there appears to be no reason to select CIDL over another more familiar language that supports parallel processing and object-oriented paradigms.

Overall, APTAS has great potential to improve software system quality and expedite software system production. The single, most important obstacle to the APTAS system is populating the knowledge base. As more domains are modeled, the potential for APTAS increases (particularly when using the process described in this research).

Appendix C. REFINE Source Code

The REFINE source code may be obtained, upon request, from:

Maj Paul Bailor

AFIT/ENG

2950 P Street

Wright-Patterson AFB, OH 45433-7765

(513) 255-9263

DSN 785-9263

email: pbailor@afit.af.mil

Bibliography

1. Aho, Alfred B., et al. *Compilers: Principles, Techniques, and Tools*. Mark S. Dalton, 1985.
2. Anderson, Cynthia G. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/92D-01, AFIT, 1992.
3. Arango, Guillermo. "Domain Analysis - From Art Form to Engineering Discipline." *Proceedings of the Fifth International Workshop on Software Specification and Design*. 152–159. May 1989.
4. Basili, Victor R. and John D. Musa. "The Future Engineering of Software: A Management Perspective," *IEEE Computer*, 90–96 (September 1991).
5. Blackman, Samuel S. *Multiple-Target Tracking with Radar Applications*. Dedham, MA: Artech House, Inc., 1986.
6. Cecil, Danny A. and Joseph A. Fullenkamp. *Using Database Technology to Support Domain-Oriented Application Composition Systems*. MS thesis, AFIT/GCS/ENG/93D-03, Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1993.
7. Cossentine, Jay A. *Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation Systems*. MS thesis, AFIT/GCS/ENG/93D-04, Graduate School of Engineering, Air Force Institute of Technology, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1993.
8. Crowley, Nancy. *On the Automation of Object-Oriented Requirements Analysis*. PhD Dissertation, Air Force Institute of Technology, 1993.
9. D'Ippolito, Richard S. "Using Models in Software Engineering." *Proceedings: TRI-Ada '89*. 256–264. 1989.
10. Gool, Warren E. *Alternative Architectures for Domain-Oriented Application Composition and Generation Systems*. MS thesis, AFIT/GCS/ENG/93D-11, Graduate School of Engineering, Air Force Institute of Techonolgy (AU), Wright-Patterson AFB, OH, December 1993.
11. Hayes-Roth, Frederick, et al. *Building Expert Systems*. Addison-Wesley Publishing Company, Inc., 1983.
12. Hong, Lang, "MATLAB Implemenation: targtgen.m." Wright Laboratories, 1993.
13. Hovanessian, Shahan A. *Introduction to Sensor Systems*. Norwood, MA: Artech House, Inc., 1988.
14. Iscoe, Neil. "Domain-Specific Reuse: An Object-Oriented and Knowledge-Based Approach." *Tutorial from Software Reuse: Emerging Technology* edited by Will Tracz, 299–308, IEEE Computer Society Press, 1989.
15. Iscoe, Neil. "Domain Modeling - Evolving Research." *Proceedings of the Sixth Annual Knowledge-Based Software Engineering Conference*. 300–304. 1991.

16. Jensen, Paul S. *Avionics Tracking System Specification for Automatic Code Synthesis.* Technical Report, Lockheed Software Technology Center, April 1990.

17. Jensen, Paul S. and Lori Ogata. *DRAFT Final Report for Automatic Programming Technologies for Avionics Software (APTAS).* Technical Report, Lockheed Software Technology Center, July 1991.

18. Kyo C. Kang, et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report, Software Engineering Institute, Carnegie Mellon University, November 1990.

19. Lee, Kenneth J., et al. *Model-Based Software Development (draft).* Technical Report, Software Engineering Institute, December 1991.

20. Lockheed Software Technology Center. *Software User's Manual for the Automatic Programming Technologies for Avionics Software (APTAS) System*, June 1991.

21. Louie, Joe, "Informal Electronic Mail Correspondence," August 1993.

22. Lowry, Michael R. "Software Engineering in the Twenty-first Century." *Automating Software Design* edited by Michael R. Lowry and Rob D. McCartnery, 627–654, AAAI Press/MIT Press, 1991.

23. Lubars, Mitchell D. "Domain Analysis and Domain Engineering in IDeA." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 163–177, IEEE Computer Society Press, 1991.

24. Marciniak, John J. and Donal J. Reifer. *Software Acquisition Management Managing the Acquisition of Custom Software Systems.* Wiley Series in Industrial Software Engineering Practice, John Wiley and Sons, Inc., 1990.

25. McCain, Ron. "Reusable Software Component Construction: A Product-Oriented Paradigm." *AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference11.* 125–135. October 1985.

26. Neighbors, James M. *Software Construction Using Components.* PhD dissertation, University of California, Irvine, 1981.

27. Neighbors, James M. "Draco: A Method for Engineering Reusable Software Systems." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 34–52, IEEE Computer Society Press, 1991.

28. Ogush, Mike. "A Software Reuse Lexicon," *CrossTalk* (December 1992).

29. Polak, Wolfgang and Henson Graves. *The CIDL Language User Manual.* Lockheed Software Technology Center, 1990.

30. Prieto-Díaz, Rubén. "Domain Analysis for Reusability." *Proceedings of the 11th Annual International Computer Software and Application Conference.* 23–29. IEEE Computer Society Press, 1987.

31. Prieto-Díaz, Rubén. "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, *15*(2):47–54 (1990).

32. Randour, Marry Anne. *Creating and Manipulating a Domain Specific Formal Object Base*. MS thesis, AFIT/GCS/ENG/92D-13, Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1992.

33. Rasure, John, et al. "Visual Language and Software Development Environment for Image Processing," *International Journal of Imaging Systems and Technology*, *2*:183–199 (1990).

34. Rumbaugh, James, et al. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.

35. Smith, Douglas R. "KIDS: A Semiautomatic Program Development System," *IEEE Transactions on Software Engineering*, *16*:1024–1043 (September 1990).

36. Tracz, Will. "Domain Analysis Working Group Report - First International Workshop on Software Reusability," *ACM SIGSOFT Software Engineering Notes*, *17*(3):27–34 (July 1992).

37. Waggoner, Robert. *Domain Modeling of Time-Dependent Systems*. MS thesis, AFIT/GCS/ENG/93D-23, Graduate School of Engineering, Air Force Institute of Techonology (AU), Wright-Patterson AFB, OH, December 1993.

38. Warner, Russel M. *A Method for Populating the Knowledge Base of AFIT's Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93D-24, Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1993.

39. Wartik, Steven and Rubén Prieto-Díaz. "Criteria for Comparing Reuse-Oriented Domain Analysis Approaches," *International Journal of Software Engineering and Knowledge Engineering*, *2*:403–431 (September 1992).

40. Weide, Timothy. *Development of a Visual System Interface to Support a Domain-oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93M-05, Graduate School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, March 1993.

41. Welgan, Robert L. *Domain Analysis and Modeling of a Model-Based Software Executive*. MS thesis, AFIT/GCS/ENG/93D-25, Graduate School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1993.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE

A METHOD FOR POPULATING THE KNOWLEDGE BASEOF APTAS, A DOMAIN-ORIENTED APPLICATIONCOMPOSITION SYSTEM

5. FUNDING NUMBERS

6. AUTHOR(S)

Raleigh A. Sandy III

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/93D-13

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Mr John Werthmann
WL/AART BLDG 22
Wright-Patterson AFB, OH 45433-7765

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A formal process is described for populating the knowledge base of the Automatic Programming Technologies for Avionics Software (APTAS) system. This process was developed using a general knowledge base population process that applies to many knowledge-based software engineering systems. This general process is also described. The formal process for APTAS was demonstrated by storing new information from the radar tracking domain into the knowledge base of APTAS. Several procedures (some automatic and some manual) were implemented to support the formal process.

14. SUBJECT TERMS

Knowledge Base Population, Domain Analysis, Knowledge-Based Systems, Expert Systems, Knowledge Acquisition, Software Engineering

15. NUMBER OF PAGES

130

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL