FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures

Kyo C. Kang, Sajoong Kim, Jaejoon Lee[1], Kijoo Kim, Gerard Jounghyun Kim, Euiseob Shin[2]
Department of Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)
San 31 Pohang, Kyoungbuk 790-784, Korea
E-mail: {kck, gkim}@postech.ac.kr,{sjkim, gib, kijoo}@reuse.postech.ac.kr

Abstract

Systematic discovery and exploitation of commonality across related software systems is a fundamental technical requirement for achieving successful software reuse. By examining a class/family of related systems and the commonality underlying those systems, it is possible to obtain a set of reference models, i.e., software architectures and components needed for implementing applications in the class. FORM (Feature-Oriented Reuse Method) supports development of such reusable architectures and components (through a process called the "domain engineering") and development of applications using the domain artifacts produced from the domain engineering.

FORM starts with an analysis of commonality among applications in a particular domain in terms of services, operating environments, domain technologies, and implementation techniques. The model constructed during the analysis is called a "feature" model, and it captures commonality as an AND/OR graph, where AND nodes indicate mandatory features and OR nodes indicate alternative features selectable for different applications. Then, this model is used to define parameterized reference architectures and appropriate reusable components instantiatable during actual application development.

Architectures are defined from three different viewpoints (subsystem, process, and module) and have intimate association with the features. The subsystem architecture is used to package service features and allocate them to different computers in a distributed environment. Each subsystem is further decomposed into processes considering the operating environment features. Modules are defined based on the features on domain technology and implementation techniques.

Modules serve as basis for creating reusable components, and their specification defines how they are integrated into the application (e.g.,. as-is integration of pre-coded component, instantiation of parameterized templates, and filling-in skeletal codes). Our experiences have shown that for the electronic bulletin board and the private branch exchange (PBX) domains, "features" make up for a common domain language and the main communication medium among application users and developers. Thus, the feature model well represents a "decision space" of software development, and is a good starting point for identifying candidate reusable components.
.

---

[1] Department of Computer and Communications Engineering, Graduate School for Information Technology, POSTECH. Also, affiliated with LG Information & Communications Ltd., R&D Center.

[2] Multimedia Lab., Korea Telecom Research & Development Group, Korea Telecom, 17 Woomyung-dong, Seocho-Gu, Seoul, Korea, E-mail : esshin@case.kotel.co.kr.

1. Introduction

Reuse of software artifacts is one of the most promising solutions to the so-called "software crisis". Systematic discovery of commonality across related software systems and representation of the commonality in an exploitable form are fundamental technical requirements for achieving successful software reuse. By examining a class/family of related systems and the commonality underlying those systems, one can provide a set of reference models for describing the class, and architectures and components for implementing applications in the class. It can also provide a basis for understanding and communicating the problem space addressed by the software in the domain.

There have been many attempts to support software reuse, but most of these efforts have focused on exploratory researches to understand issues in domain specific software architecture [Mettala and Graham 1992], component integration and application generation mechanisms [Batory *et al*. 1995; Floch 1990, Neighbors 1984], theoretical researches on software architecture and architecture specification languages [Luckham *et al*. 1995; Moriconi *et al*. 1995; Shaw and Galran 1996; Tracz 1993], development of reusable patterns [Buschmann *et al*. 1996; Gamma *et al*. 1995; Mckenny 1996], and design recovery from existing code [Biggerstaff 1989]. There have been few efforts [Gomaa *et al*. 1994; Kang *et al*. 1990] to develop systematic methods for discovering commonalty and using this information to engineer software for reuse.

FORM (Feature-Oriented Reuse Method) is a systematic method that looks for and captures commonalties and differences of applications in a domain in terms of "features" and using the analysis results to develop domain architectures and components. The model that captures the commonalties and differences is called the "feature model" and it is used to support both engineering of reusable domain artifacts and development of applications using the domain artifacts. Once a domain is described and explained in terms of common and different "units" of computation, they are used to construct different "feasible" configurations of reusable architectures.

The use of "features" is motivated by the fact that customers and engineers often speak of product characteristics in terms of "features the product has and/or delivers". They communicate requirements or functions in terms of features and, to them, features are distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained. This fact, however, has not been supported or exploited fully by most of software engineering methods so far. We believe that features are abstractions that both customers and developers understand and should be the first class objects in software development.

The concept of feature-orientation is not completely new in software engineering, and there have been

efforts to base the development of software on the notion of features, especially in the telephony domain. For instance, [Kang *et al*. 1990; Palmer and Liang 1992] focused on feature-oriented requirements engineering; [Sloane and Holdsworth 1996] developed a feature-oriented testing scheme; and [Fekete 1993; Zave 1993] introduced a method for the feature specification and interaction analysis. Most of these efforts, however, have been either ad hoc or limited to a certain phase of the lifecycle. A comprehensive and systematic software development method that takes feature-orientation as its paradigm throughout the entire lifecycle has yet to emerge.

This paper extends the FODA method [Kang *et al*. 1990] proposed by the author. In FODA, the concept of using a feature model for requirements engineering was introduced. FORM extends FODA to the software design phase and prescribes how the feature model is used to develop domain architectures and components for reuse. The underlying philosophy for this extension is that the features of a domain characterize each variant product in the domain, and the code that implements the characterizing features should be packaged, managed, and reused as software modules.

Effective reuse must be planned and considered early in the life cycle. However, most requirements engineering methods are not appropriate for capturing requirements in a reusable format. A feature model that captures services provided by the applications in a domain in an abstract form can be used for such a purpose. That is, a feature model with additional model knowledge (e.g., how selection of features can affect choice of different architectures and associated set of modules), reusability can be considered during early phases of the software life cycle.

This paper is organized as the following. The underlying concept of FORM is introduced in the next section. Processes of developing artifacts for a given domain (called the domain engineering), and creating applications using the reusable artifacts (called the application engineering) are discussed in sections 3 and 4, respectively. Section 5 concludes this paper with a discussion of lessons learned from our experience with FORM. Application of FORM to the electronic bulletin board domain [Bryant 1994; Wolfe 1994] has been used in this paper as an illustrative example.

2. Method Concepts

The core of FORM lies in the analysis of domain features and use of these features to develop reusable domain artifacts. In this section, the concept of feature-orientation is introduced, followed by a discussion of the engineering principles used to develop reusable artifacts.

2. 1 Feature-Orientation

As an application domain matures, we see that a set of standard techniques used to implement applications

and a set of standard terms used by the users and developers naturally emerge. For example, in the graphics domain "rendering", "shading", etc. are techniques for drawing graphical objects on a screen, and "call-forwarding" and "call-back" represent service features in the telephony domain. Techniques used in the development and services provided by the applications evolve as a domain evolves, but, as a domain become stabilized, techniques are also stabilized and, consequently, the domain terminology used by users and developers become standardized. This domain-oriented terminology constitutes the vocabulary of a domain language used by domain experts and users to communicate their ideas, needs, and implementation techniques.

Any software engineering effort starts with a careful analysis of the domain. In a domain analysis, primary inputs are documents of applications (users manual, design documents, etc.). The volume of the documents to be analyzed, however, tends to be enormous in a domain of any reasonable size. ***Our experiences have shown that the domain analysis can be performed efficiently and effectively by instead analyzing the domain language. That is, services provided by and techniques used in applications are abstracted as "features", and they are used by domain experts to communicate their ideas, needs, and problems.*** An informal definition of feature is stated as essential characteristics of applications in a domain.

There are different types of features that are of concern depending on the interest one might have in system development. Users, system analysts, and developers are all usually involved in system development and have different interests. Users are concerned more about services or functions provided by the system (i.e., service features); system analysts and designers are concerned about domain technologies (e.g., navigation methods in the avionics domain, methods of fund transfer in the banking domain); and developers are concerned about implementation techniques (e.g., databases, sorting algorithms). Applications cannot be built unless a sound decision is made among them that will produce an implementable and consistent set of features. The selected set of features will constrain the space of feasible architectures.

Among these different types of features, there may exist composition relationships, which further constrain the space of corresponding feasible architectures. Architectures basically represent configurations of software modules and components that satisfy a given set of selected features, and give us glimpse of opportunities for reuse.

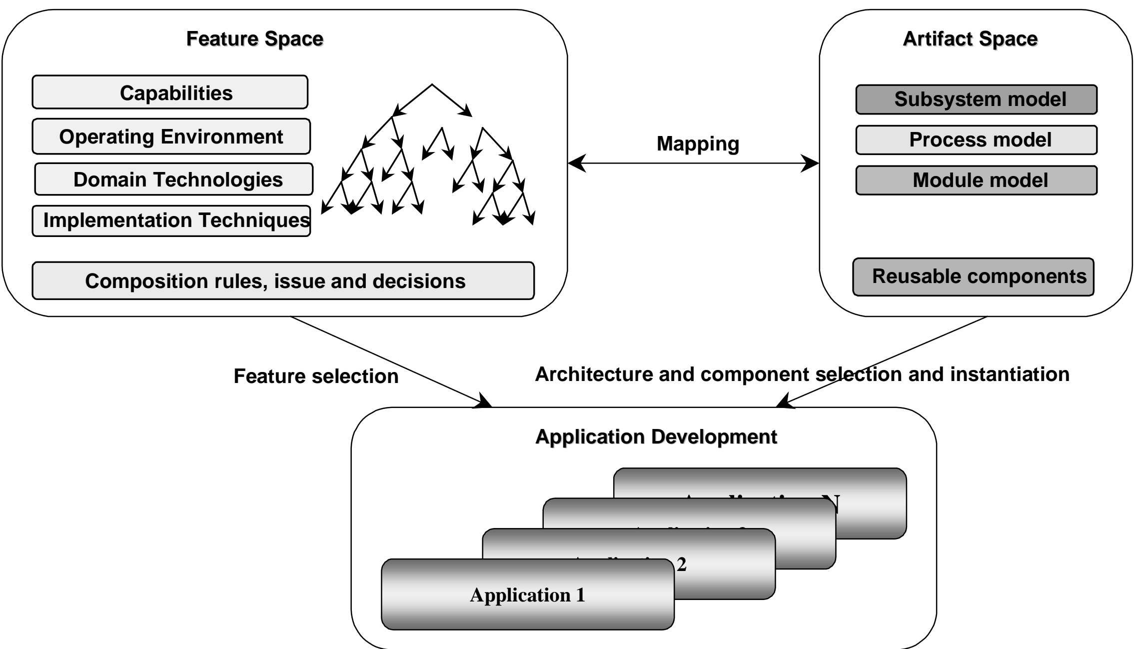

Figure 1-1: FORM's Concept of Application Development by Reuse of Domain Engineered Artifacts through Feature Selection.

The importance and significance of the feature modeling can be summarized as:

- By standardizing the meanings of features and integrating them as a model, it is possible to define standard terms and ease communication problems.
- The feature model can be used as a yardstick for evaluating commonalties and differences between applications.
- The feature model characterizes a domain and it can be used to compare with other domains.
- Construction of a feature model requires resolution of conflicting interpretations of the same feature, expediting the standardization process.
- Commonalties manifested in the model indicate clearly where reuse opportunities are.
- Complex interactions among features become visible.

The feature model, thus, defines a "decision space" for application development. Figure 1-1 illustrates this view of design as an association between the decision (or feature) and the artifact space.

The feature analysis for a domain creates a domain dictionary with a list of services provided by applications in the domain and techniques used for implementation. Description of each service and technique is also part of the dictionary along with rules of composing these services and techniques in application development.

2.2 Engineering Principles

Once commonality is analyzed and modeled with the feature analysis, the feature model is used to engineer architectures and components. Various software engineering principles are used in this process to further breath in adaptability and reusability into the target software structure. The engineering principles employed in FORM for designing reusable architectures and components are as follows:

- Separation of concerns and information hiding: Reusable artifacts are defined in four conceptually different levels of detail; subsystem models, process models, module models, and module specifications and implementations (See the upper right box of Figure 1-1.). In the development of the subsystem models, we are concerned with allocation of functional features to subsystems and interfaces between them. In designing the process models, we are mainly interested in performance. In specifying the modules, the information hiding is one engineering principle that we use to abstract their interfaces and accommodate for different implementations. This principle allows configuration of different applications, or different implementations of the same application, from different sets of reusable artifacts.
- Localization of function, data, and control: In the design of architectures, architectural components are defined so that each component performs one function. Also, by localizing frequently accessed data to appropriate components, and by localizing controls within the right components, we can develop loosely coupled architectures and minimize communication overheads.
- Parameterization of artifacts using the features: Features defined in the feature model are used to parameterize architectures and components. With this approach, we can develop components free of design decisions (to a certain level) by embedding the features in the components as instantiation parameters. Instantiation of these parameterized artifacts are delayed until design decisions are made during the application development.
- Layering: The FORM method provides a layered architectural framework (i.e., subsystem, process, module, etc.) according to the feature model hierarchy that separates service, operation, domain technology, and implementation modules. Since the architectural framework follows the structure of the feature model (e.g., a module hierarchy is derived from corresponding features), we believe that modules developed using this framework have high adaptability and reusability due to separation of design decisions of different kinds.
- Separation of components from the component connection mechanism: FORM provides inter-module communication mechanisms (See Figures 4-7 and 4-8.) that allow development of modules free of particular module communication techniques. Binding of particular communication techniques can be made at the module instantiation time considering performance and architectural configurations (e.g., distributed vs. centralized architecture).
- Synthesis of design components based on feature selection: Components are selected, instantiated, and integrated during the application development through selection of desired features. Components

are instantiated based on the set of features selected for an application and these components are integrated based on the integration rules defined in the model.

These engineering principles support the development of architectures that are adaptable and re-configurable for different applications. Parameterization of artifacts with features and development of applications through selection of feature sets are powerful synthesis techniques for implementing an effective application generator for a stable domain. The next section lays out the overall process of applying FORM, pointing out various lifecycle products that are produced.

2. 3 Engineering Process

The FORM method consists of two major engineering processes: domain engineering and application engineering, as shown in Figure 2-1. The domain engineering process consists of activities for analyzing systems in the domain and creating reference architectures and reusable components based on the analysis results. The reference architectures and reusable components are expected to accommodate the differences as well as commonalties of the systems in the domain.

The application engineering process consists of activities for developing applications using the artifacts created in the domain engineering. For typical applications in a given domain, the application engineering should be trivial compared to the traditional application development approach that does not have domain-orientation (if the domain engineering was performed appropriately). Details of activities and artifacts from domain engineering and application engineering are discussed in the following sections.

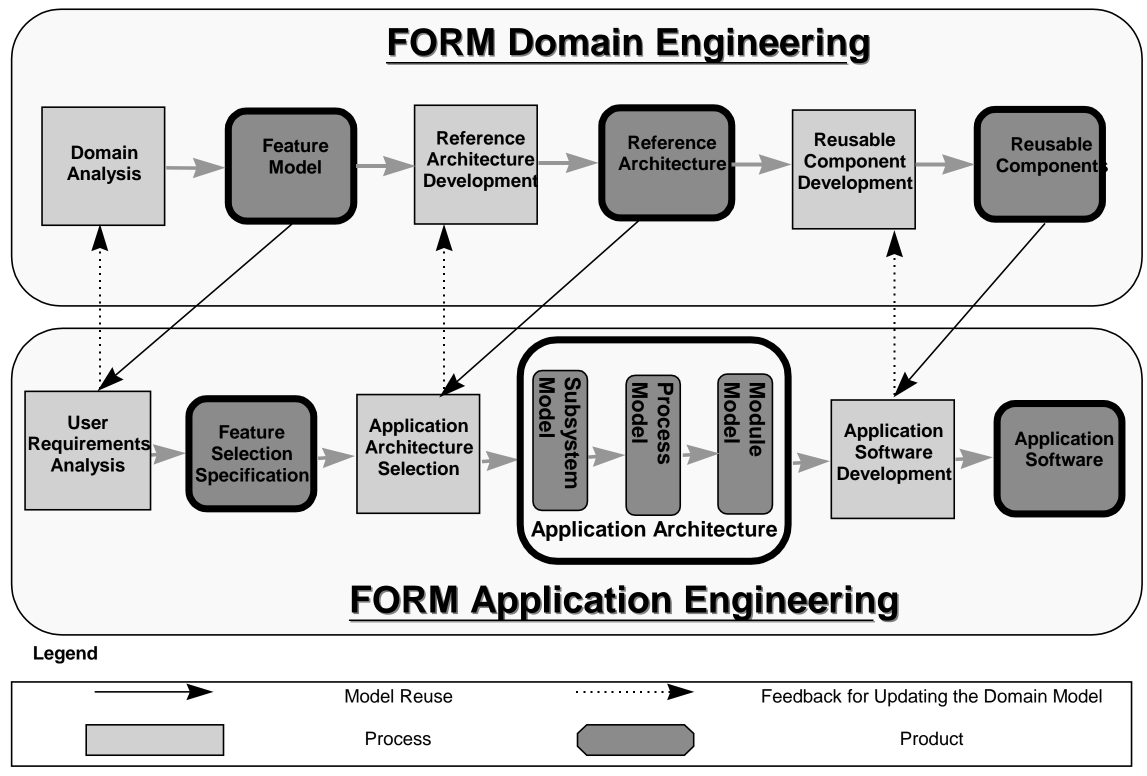

Figure 2-1: FORM Engineering Processes.

3. Domain Engineering

The purpose of domain engineering is to develop domain artifacts that may be used (and reused) in developing applications for a given domain. Domain engineering consists of activities for gathering and representing information on systems that share a common set of capabilities and data. In usual approaches to software reuse, the product of such domain engineering might include only the reusable components and their parametric representations applicable to that domain. In FORM, the domain knowledge is engineered and organized in a more comprehensive manner; it provides user observable and selectable common features and reference software architectures of the target domain in which roles of the reusable components are explicitly described in terms of their places in each architecture. This way, not only the effectiveness of the reuse environment is increased (e.g., through the use of user understandable feature model), but also the "adaptability" of the reusable components is increased by the mapping between the reference architecture and the target application model.

There are three phases in FORM domain engineering: context analysis, domain (or feature) modeling, and architecture (and component) modeling. During the context analysis, the exact scope of the domain and intended use of the domain application, various external conditions, and possible interactions with the external world are first identified. Then, during the domain (or feature) modeling phase, user understandable "features" representative of the already well defined target domain are identified, and their

interrelationships are modeled. A "feature model" is created and there may exist many consistent feature specifications derivable from the model (Such a feature specification is later used to find a corresponding reference architecture to be reused.). While the context analysis and domain modeling define a user selectable "feature space", the architectural modeling defines the "artifact space" where the reusable artifacts, that is, the reusable software components and their configurations and hierarchical decompositions, are constructed. Although the context analysis is an important part of domain engineering, we focus, in this paper, on domain analysis and architecture modeling.

3.1 Domain Analysis and Feature Modeling

The goal of domain analysis is to identify commonalties and differences of systems in a domain and represent them in an exploitable form. Domain analysis consists of three processes as shown in Figure 3-1: planning, feature analysis, and validation activities.

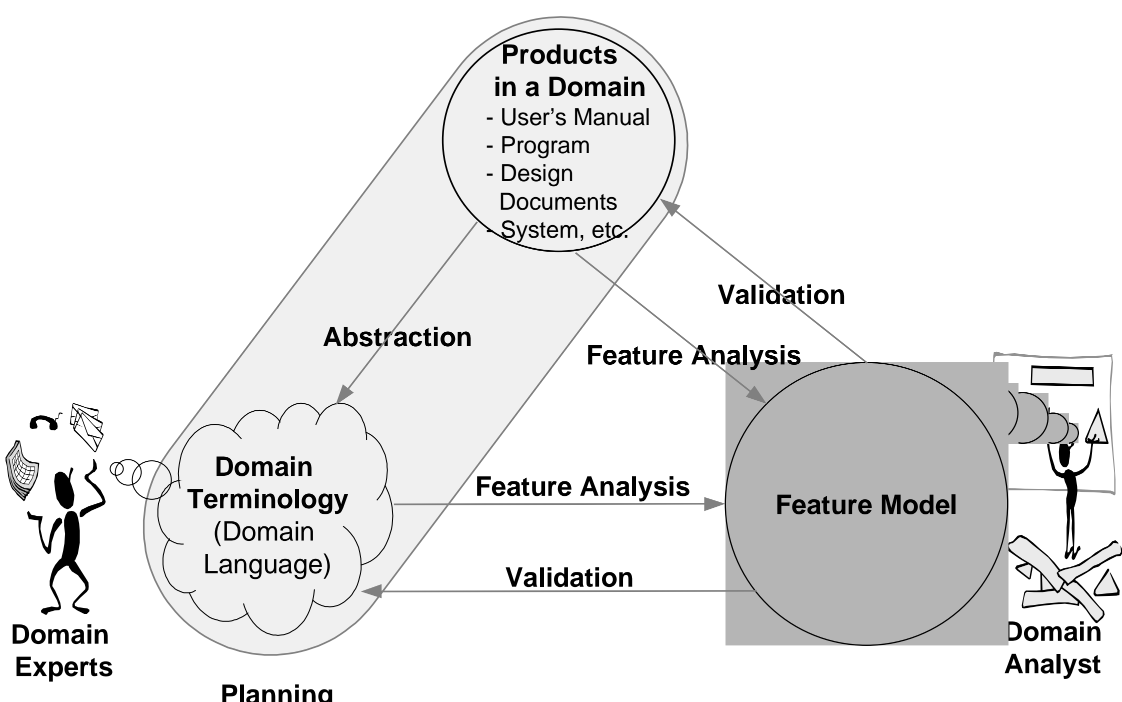

Figure 3-1: Domain Analysis Process.

In the planning activity, a family of products in the domain is identified and a preliminary assessment of commonality among these products is made. Usually, commonality can be found for products in mature domains, and maturity is indicated by the existence and utilization of standards, documented standard terminology, availability of experts, etc. ***For organizations that have not had any domain engineering experience, we strongly recommend that a domain be selected that is small and has a high level of commonality among the applications and for which domain experts and up-to-date documents***

are available. This will increase the chance of success and a positive experience. While working with industry partners, we have found that they are very much worried about the high cost of domain engineering and get easily turned away by a minor problem.

Feature analysis consists of activities for identifying product features, classifying them, and organizing them as a set of coherent models. Each of these activities is discussed below.

Feature Identification

Identification of features involves abstracting domain knowledge obtained from the domain experts (e.g., system users, analysts, and system developers) and other sources such as books, user manuals, design documents, and source programs. As discussed in section 2.1, we identify features from four different perspectives: capability, operating environment, domain technology, and implementation technique. Usually, user manuals are good sources for identifying capability features. Features on operating environment and domain technology can usually be found in the requirements and design documents, and implementation features can be found in the design documents and program sources. ***It is our experience that, in mature and stable domains, analyzing the terminology of the domain sub-language is an effective and efficient way to identify features of a given domain***.

Feature Classification

As potential features are identified, they are classified according to the types of information they represent. Types of decisions (or feature selection) that application developers make fall largely into four categories: those about application capabilities, operating environments, domain technologies, and implementation techniques (See Figure 3-2.). A "capability" feature literally characterizes a distinct service, operation, function, or performance that an application (for the given domain) may possess. They may be further divided into "functional" and "non-functional" ones. For instance, in the electronic bulletin board system (EBBS) example, "Private e-mail service", "Public message boards", "File transferring service", and "Chatting service" are features that characterize various functional aspects of the EBBS, while "For hobbyist use", "For entrepreneurial use", "Graphical interface", "Textual interface", and "Maximum 100 users" are non-functional features that describe intended use, or expected performance.

An "operating environment" feature represents attributes of environment in which an application is used and operated. Typical examples include hardware platform related features (e.g., "PC"), operating system related ones (e.g., "Windows 95"), database or file system related ones (e.g., "Relational DMBS") and network related ones (e.g., "Accessible via internet").

"Domain technology" and "implementation technique" features represent implementation details at lower

and more technical levels. The difference between these two groups of features is that a "domain technology" is more specific to a given domain (e.g., navigation methods in the aviation domain) and may not be usable in other domains, while "implementation technique" is more generic and may be used in other domains (in the realm of computer science). For instance, as alternatives for display domain technology, "Formatted Text Display" or "Unformatted Text Display" may be selected. The "Formatted Text Display" may use "ANSI" or "VT-100" terminal handling techniques, while the "Unformatted Text Display" can only use "TTY" terminal handling technique.

A feature model should cover all four categories of features for a domain and it should encompass as many applications as is feasible (economically), to include the fullest range of features and feature values. Figure 3-2 shows various features (represented as tree nodes) exhibited in the electronic bulletin board system (EBBS) domain.

Feature Organization and Analysis

A feature does not exist in isolation from other features, and there exist various relationships among these features. For example, a certain capability feature may entail an inclusion of operating environment features (e.g., "Pentium PC" and "Windows 95"), or a certain domain technology can only work with a selected implementation technique (e.g., "Unformatted Display" and "TTY"). In FORM, to support a description of the decision space (that fully depicts the interdependencies among features and selection criteria) and its use in developing an application through software reuse later, the following constructs are engineered to complete the feature model (See Figure 3-2.).

- A feature diagram, a graphical AND/OR hierarchy of features, that captures logical structural relationships (e.g., composition and generalization) among features. Three types of relationships are represented in this diagram: "composed-of", "generalization/specialization", and "implemented-by". Features themselves may be "mandatory" (unless specified otherwise), "optional" (denoted with a circle), or "alternative" (denoted with an arc).
- Composition rules that supplement the feature diagram with mutual dependency and mutual exclusion relationships (This is used to verify the consistency and completeness of the selected features.). For example, Figure 3-3 illustrates a composition rule that specifies that the "DOS" feature of "Operating system" must be selected together with "Multi-tasker" feature of "S/W platform".
- Issues and decisions that record various trade-offs, rationale, and justifications for feature selection. The lower portion of Figure 3-3 illustrates a textual description of a possible justification for selecting the "NAPLPS" standard over the "RIP" standard for a graphical interface, that "the NAPLPS standard is device-independent but it is little more than a window dressing".

A feature model with AND-nodes at an upper level and OR- nodes at a lower level usually indicates a high level of reuse opportunity. Alternatives (i.e., OR-nodes) at the upper level usually mean that applications in the domain do not share much commonalties in terms of services and functions provided by them. This indicates that the domain might not have much reuse opportunity at the application level, although there might still be opportunities for reuse at low level generic components such as math libraries and abstract data types. Alternatives (OR-nodes) at a lower level indicate different ways of implementing certain components and, with an appropriate application of information hiding and encapsulation techniques, reusable components can be developed.

We found that feature interaction problems [Aho and Griffeth 1995; Cameron and Velthuuijsen 1993; Griffeth and Lin 1993; Zave 1993] usually occurred among service features that may be in operation concurrently and thus, interactions among these features must be analyzed carefully. For instance, when we worked on a private branch exchange (PBX) system with our industry partner, we found that the "hot-line" feature (which automatically connects a caller and a callee when any of the subscriber's handset is picked up) resulted in an unexpected phone connection with a third party, due to the already existing "call-forwarding" feature (e.g., a call is forwarded to one of the hot-line subscriber). Other unpredictable behaviors were possible. It is difficult to predict the consequence of feature interaction in such a case unless there is a careful analysis of interactions among features and engineering of software to handle the interactions.

The AND/OR diagram tends to become very complex and unmanageable for domains of any reasonable size. We found that the use of a textual language (as shown in Figure 3-3) to define a feature model and using the AND/OR diagram to show the relationships among a set of selected features was most effective.

Model Validation

The feature model should be validated before use. The validation process is rather straightforward. Validation is performed by instantiating the model for each application considered in the domain anlysis and checking if the instantiated model correctly represents the application features. We found that the feature model is a powerful mechanism for comparing characteristics of applications.

An implementation of an application may be thought of as an instantiation of a feature model, that is, a particular selection of a set of capabilities, environmental conditions, domain technologies, and implementation techniques from the model. As the application developer defines an application by instantiating the feature model, s/he should be able to explore alternatives and verify the completeness and consistency of a set of selected features. For example, a requirement analyst should be able to describe a proposed system to a potential customer in terms of possible capability features and also discuss

alternatives. The consistency and completeness of the selected features, for instance, may be checked using a constraint checking algorithm, with particular choices explained through the use of "issues and decisions" domain knowledge.

A feature model, including feature definitions and composition rules, describes a domain theory. It not only includes the standard terms/concepts and their definitions, it also describes how they are related structurally and compositionally.

The next section describes how the features model is used to engineer domain products for reuse.

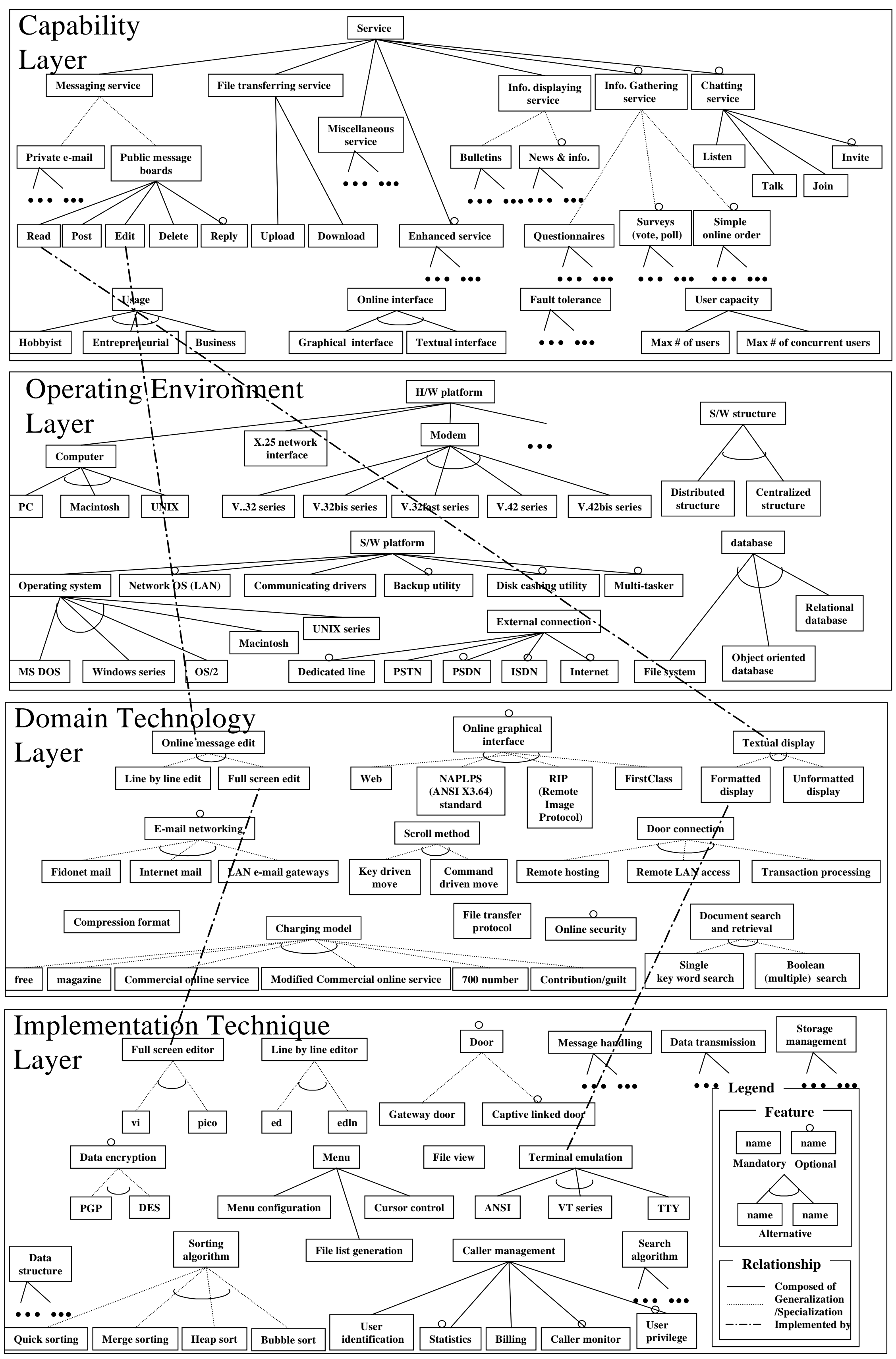

Figure 3-2: The Feature Model of EBBS Domain.

```
FEATURE EBBS;
    DESCRIPTION EBBS is an acronym for Electronic Bulletin Board System. This is software that allows
        a computer to be used as a message posting and reading system that has some similarities to a
        bulletin board you might find in an office or in a grocery store. EBBS also allow the users to send
        private messages to others, and to download files that are stored on the computer. Some EBBS also
        allow users to run other programs such as online games, online shopping, one line database.;
    TYPE CAPABILITY;
    COMMONALITY MANDATORY;
    COMPOSED_OF Service, Usage, Online_interface, Fault_tolerance, User_capacity,...;
    ALLOCATED_TO_SYSTEM EBBS;
END FEATURE;

FEATURE Public_messaging_boards;
    DESCRIPTION Public message boards, also called message boards or message areas, are places
        where users of the EBBS can communicate among themselves. Users can freely comment on the
        messages of other users, forming a roundtable discussion between the participants.
        Also called public message conference or message areas;
    TYPE CAPABILITY;
    COMMONALITY MANDATORY;
    COMPOSED_OF read, post, edit, delete, reply;
    ALLOCATED_TO_SUBSYSTEM Board;
END FEATURE;

FEATURE DOS;
    DESCRIPTION DOS is an acronym for Disk Operating System. Of various PC platforms,
        DOS-based BBS software represents the largest single segment of any of the various types
        of bulletin boards operating today.;
    ISSUE_AND_DECISION DOS-based EBBS software is also relatively economical. In addition to
        many low-cost shareware and freeware choices, the largest of the various EBBS software
        companies are marketing DOS-based applications.;
    TYPE OPERATING_ENVIRONMENT;
    COMMONALITY ALTERNATIVE;
    COMPOSITION_RULE REQUIRE Multi_tasker;
END FEATURE;

FEATURE NAPLPS;
    DESCRIPTION NAPLPS (the North American Presentation Level Protocol Syntax) is a documented
        standard. In the United States, it's available as ANSI standard X3.64, and in Canada, as CSA
        standard T500. An online graphical display mechanism developed in the 1980s for use
        on teletext systems.
    ISSUE_AND_DECISION NAPLPS' advantage is device independence, meaning that domain
        resolution is related proportionally to the screen resolution of the user, and that colors are rendered
        according to the capabilities of the user's screen as well. The disadvantage  is that it's little more
        than window dressing. (i.e., there is no accommodation for such things as mouse.);
    TYPE DOMAIN_TECHNOLOGY;
    COMMONALITY ALTERNATIVE;
    ALLOCATED_TO_SUBSYSTEM User_interface;
END FEATURE;
```

Figure 3-3: Example of EBBS Feature Model in Textual Specification Language.

3.2 Architecture Modeling and Component Development

In this section, we provide a set of guidelines that can be used to derive domain products from the feature model. Creating domain products requires engineering skills, creativity, and experience on the part of the domain engineer.

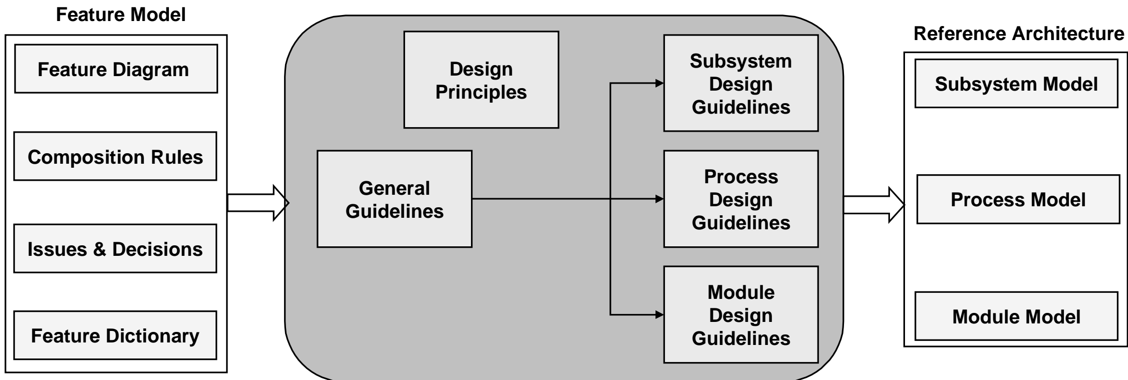

Figure 3-4: Engineering Principles.

The domain architecture, which is used as a reference model for creating architectures for different systems, is defined in terms of a set of models, each of which represents the architecture at a different level of abstraction (See Figure 3-6.). The models used in FORM are: subsystem model, process model, and module model.

The objective of domain engineering is to establish a mapping between the decision space (i.e., feature model) and the artifact space (i.e., architecture model). Each feature in the decision space somehow constrains the selection of the final reference model, and this aspect is modeled using two concepts: one by differentiating between the effect of selecting "functional" and "non-functional" features, and the other by considering the differences in types of features following the four level hierarchy. The functional features are mainly used to identify required components, while non-functional features are used to

partition components or to select types of connectors between components (See Figure 3-5.).

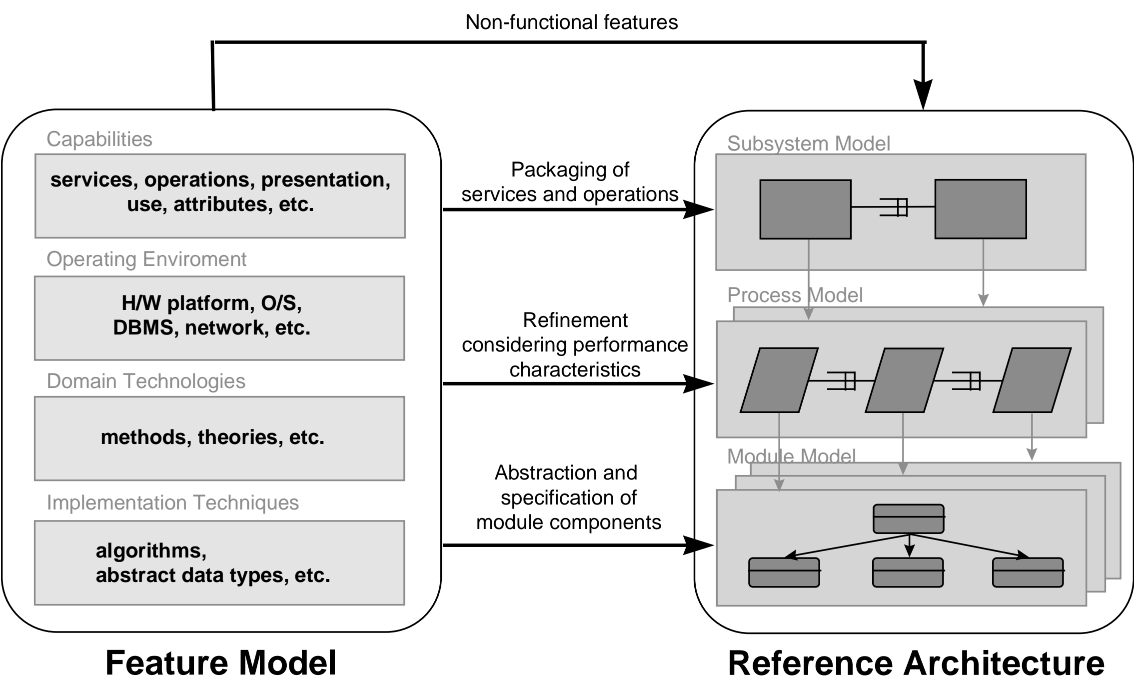

Figure 3-5: Mapping Features to Architecture Models.

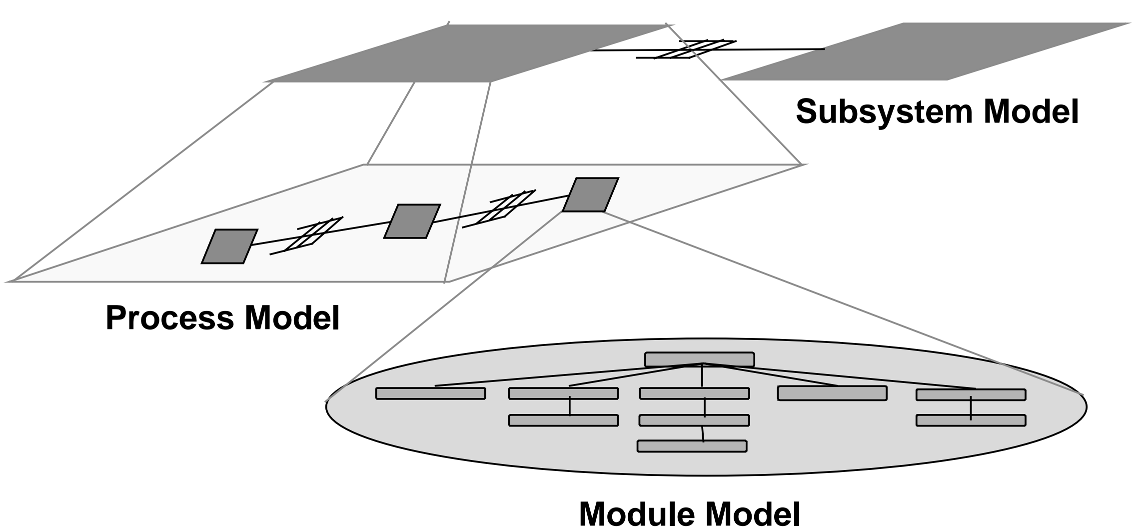

Figure 3-6: FORM Architecture Models.

In modeling an architecture, we apply the engineering principles discussed in section 2.2 to define models and allocate features to model components. Our additional guideline used is that ***the logical boundaries created by the feature model should correspond to the physical boundaries created by the architecture models***. This will increase the chance for modular composition and minimize the feature interaction problem. If there is difficulty in establishing this relation, the related features should be examined for

further decomposition or refinement. Other guidelines used to define specific models are as follows:

Subsystem Model

Subsystem model defines the overall system structure by grouping functions (the most important focus of attention) into subsystems that can be allocated to different hardware (See Figure 3-7.). This process is guided by matching capability features (identified during the domain modeling phase) to the required function blocks (Capability/service features are abstraction of the functionality as seen from the users. Decomposition or refinement of these features may be necessary for allocation to specific model components.).

Equally important is the localization of function blocks (i.e., subsystems) through appropriate selection of interfaces. Data flow between subsystems is modeled in terms of non-blocking communication via a loosely coupled message queue, or blocking communication via a tightly coupled message/reply mechanism [Gomaa 1993].

One of the most important considerations to be made in allocating features is the performance issue. A feature that requires a time-critical function(s) may be allocated to its own subsystem. Features that have a strong data or control dependency may be allocated to the same subsystem to minimize the communication overhead. User or database interfaces are usually separate subsystems as they may run at different rates than other subsystems.

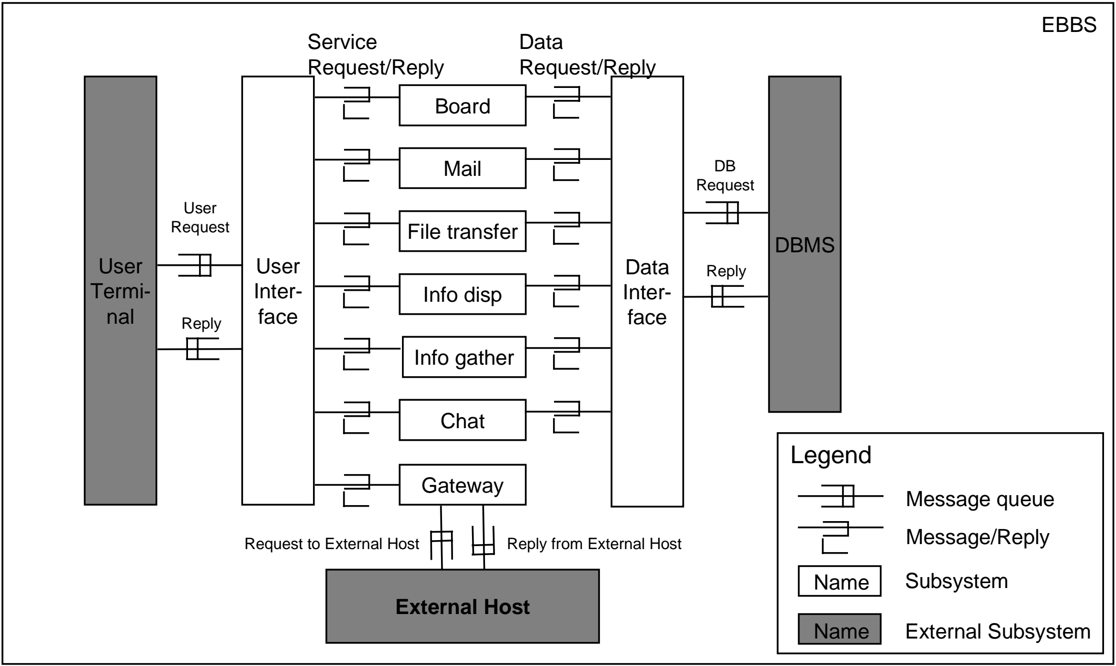

Figure 3-7: A Subsystem Model of EBBS.

Process Model

The process model represents the dynamic behavior of each subsystem (e.g., concurrency within a subsystem). For this, a subsystem is further refined into a set of concurrent "processes", allocating operational features of the service feature assigned to the subsystem to its processes. This refinement process is guided by the notions of "separation" (i.e., loosely coupled) and "cohesion", with consideration of localization of data, localization of control, and execution frequency. For instance, tasks of resident or periodic nature must be separated, while tasks with similar functionality must be made "cohesive". Each process may be categorized as "resident" or "transient" with respect to their duration of activation, and as "multiple" or "single" with respect to whether it can be forked off. For example, the "Editor" process in Figure 3-8 is defined as a separate process as it contains editing functions that are different from functions of other processes and as there may be multiple instances.

Operational features that are time-critical or have high priority may be allocated to separate processes. Also, features that have the same execution frequency may be allocated to the same process. Operational features triggered by the same event, although they may not be functionally cohesive, may be allocated to the same process to simplify the model and improve understandability.

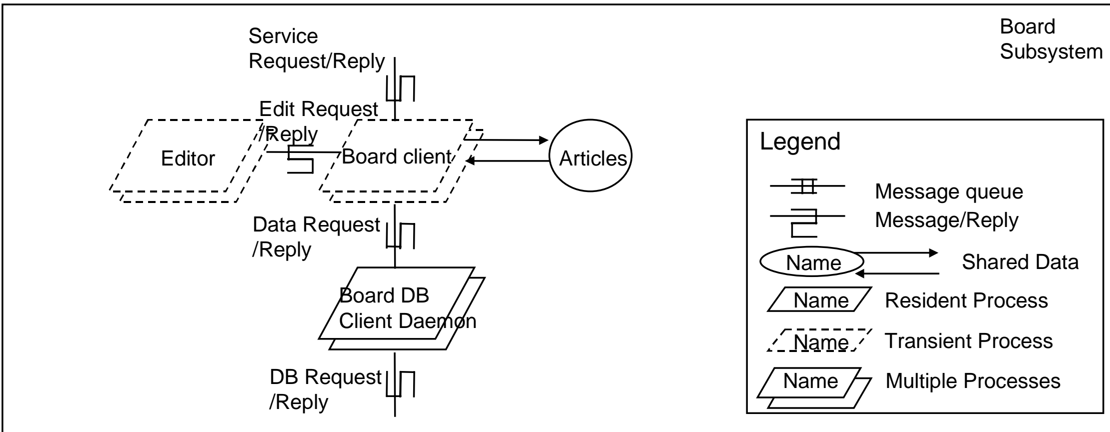

Figure 3-8: A Process Model of the Board Subsystem in EBBS.

Module Model

In developing module models, features at all levels of the feature model are used. ***It is our experience that the module hierarchy largely corresponds to the feature hierarchy, as shown in Figure 3-9. Also, alternative features may be implemented as a template module or a higher level module with an interface that could hide all different alternatives.***

This way, a module may be associated with a set of relevant features. Here, a reusable component (or a module) only contains an abstract specification (e.g., multiple code level reusable components may exist

that matches the specification of one module). To facilitate reusability, principles of "modularity", "information hiding" and "data abstraction" have been used in engineering the specifications for the modules to accommodate different code-level implementations. A module may be specified in various ways to accommodate different reuse strategies, instructing users to: (1) select a pre-coded component based on certain criteria, (2) instantiate a parameterized template by supplying the parameter values, or simply (3) choose and complete a skeletal code component [Buschmann *et al.* 1996; Gamma *et al.* 1995; Mckenny 96; Perry and Wolf 1992].

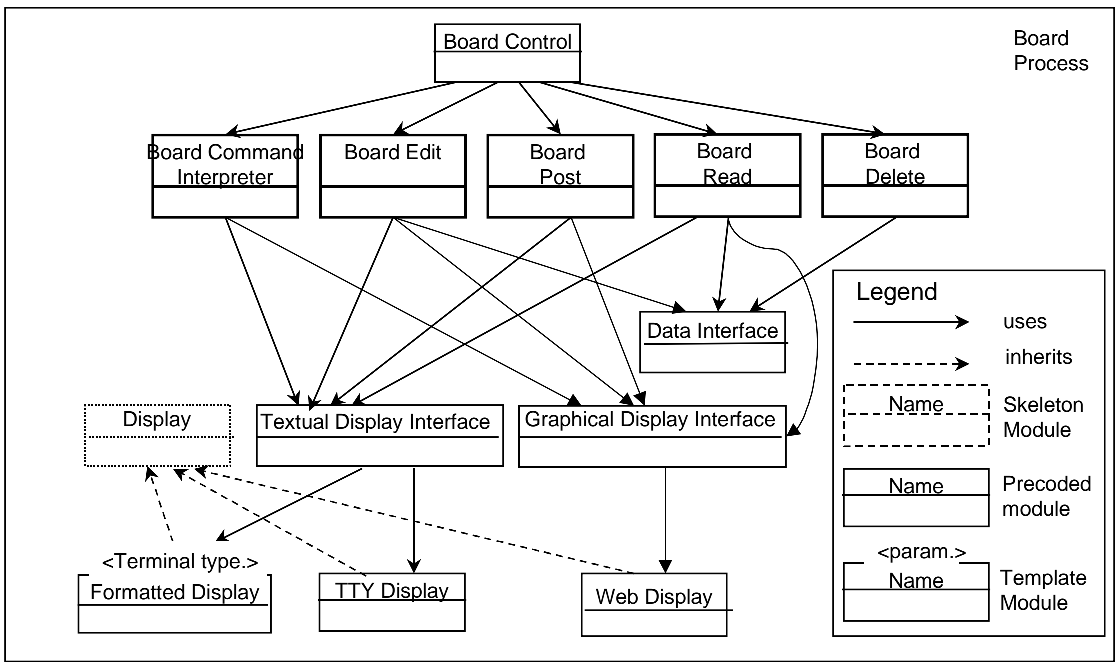

Figure 3-9: The Module model of the Board Process in EBBS.

This separation of concern also stems out of the 3C model [Tracz 1990], in which the architecture models, features, and module implementations correspond to the context, concept, and content, respectively. Such a representation is expected to be very conducive to adapting and reengineering an architecture in a situation where its counter part feature specification is similar to the new feature specification at hand. This is because by the principle of separation of concern, the represented architecture is easier to understand, and one may focus with a particular view on the portion of the system interested in modifying. At the module model level, where matching reusable components may be sought, the subsystem and process models explain which reusable components or how the reusable components fit into the larger architecture.

Features are considered following the four level feature hierarchy, since the hierarchy reflects step-wise refinement in the artifact space. At each corresponding level between the feature space and the artifact space, appropriate functional or non-functional features are used to select and partition the required

components. Therefore, there may exist several reference architecture models consistent with a number of feature selection specifications, and they basically act as reusable architectural templates in application engineering. Thus, in FORM, application of software reuse is further extended from the traditional module level up to the design level through flexible architecture configuration with a parameterized feature space.

Once a feature model and domain artifacts are developed for a domain, they are used in the development of applications. This process is discussed in the following section.

4. Application Engineering

In FORM, application engineering is a process of developing a specific application making use of the domain knowledge obtained during domain engineering (i.e., through finding a correct reference architecture and plugging in reusable software components). Application engineering proceeds by first analyzing user's requirements and selecting appropriate and valid domain features from the feature model, identifying the corresponding reference model, and completing the application development by reusing software components in a bottom-up fashion.

4.1 Requirements Analysis and Feature Selection

As indicated above, application engineering starts with a feature selection specification. This is done by first analyzing user's requirements for the target application, and finding a matching set of features from the feature model. The feature model not only lists user selectable characteristics of the system, but also includes their interrelationships and selection criteria (e.g., rationale). Therefore, it is important for the software engineer to work with users, so that one can find a valid set of features that most resemble or can accommodate what users really want. One effective method of finding such a feature set is by following the four level feature hierarchy, i.e. first considering capabilities, then operating environments, and finally domain technologies and implementation techniques, because in the artifact space, these levels correspond to increasing levels of details, and therefore, reflect stepwise refinement (See Figure 3-2.). Perturbation in the decision space is reflected in the artifact space in many different ways. As explained in the previous section, for instance, functional features in the capability layer affect selection of subsystem components, while non-functional features in turn determine their partitions, or, an operating environment feature (e.g., "distributed system") might determine communication methods among subsystems.

Figures 4-1,2,3 and Figures 4-4,5,6 show two different feature selections. In the feature diagram, selected features are indicated by bold-rectangular nodes. Rectangular nodes denote mandatory features that must be selected for the given domain, and rectangular nodes with a circle represent optionally selected features. Note that the structure of the feature selection model is in a form of a tree, a consequence of the categorical order in which the features are considered (i.e. the four level hierarchies). In progressively attempting to find required features, there may be a case when a user is not allowed (by the

interdependencies among the features) to select one. In such a case, the designer can track and analyze the path of feature selection and either negotiate with the user to change one's requirements accordingly, or try to find other possible alternatives that will satisfy the original requirements. Features in the domain technology and implementation technique layer are, in many cases, quite closely related to one another, for one of the purpose of the domain engineering is to associate domain technology with "often used" implementation techniques. This way, the context in which reusable components are developed and used becomes much clearer.

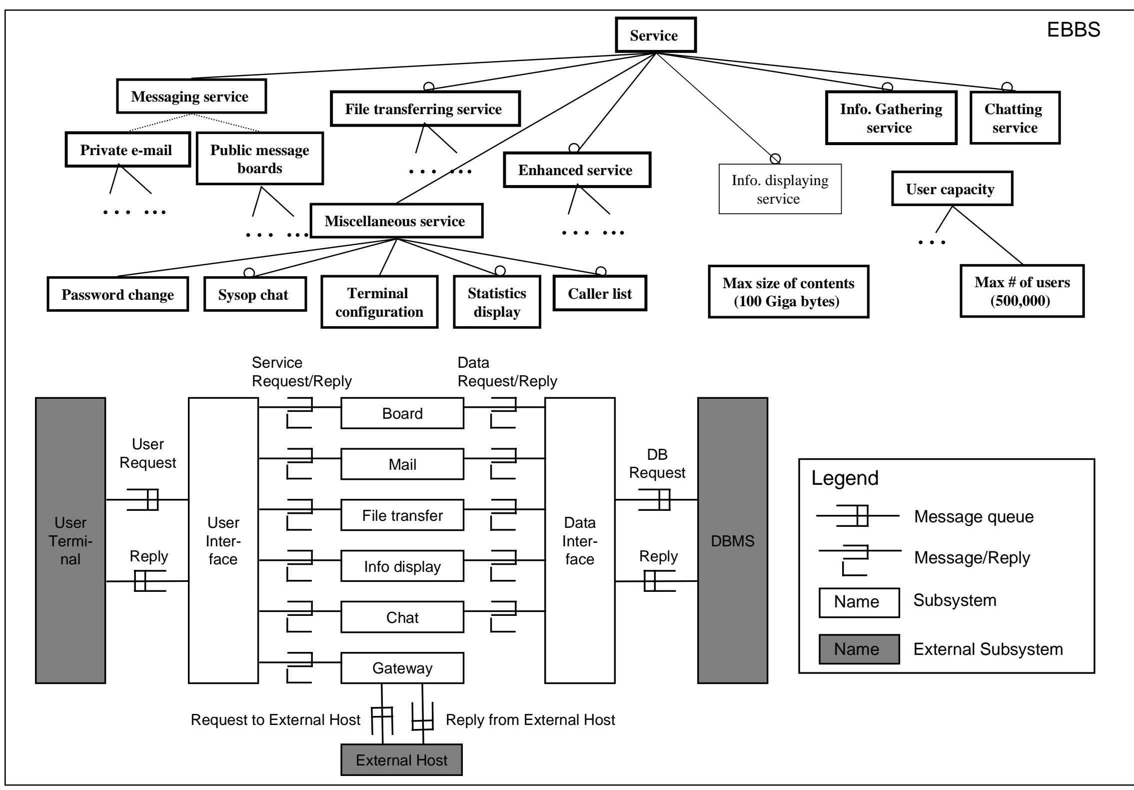

Figure 4-1: The Subsystem Model for the HITEL-POP EBBS.

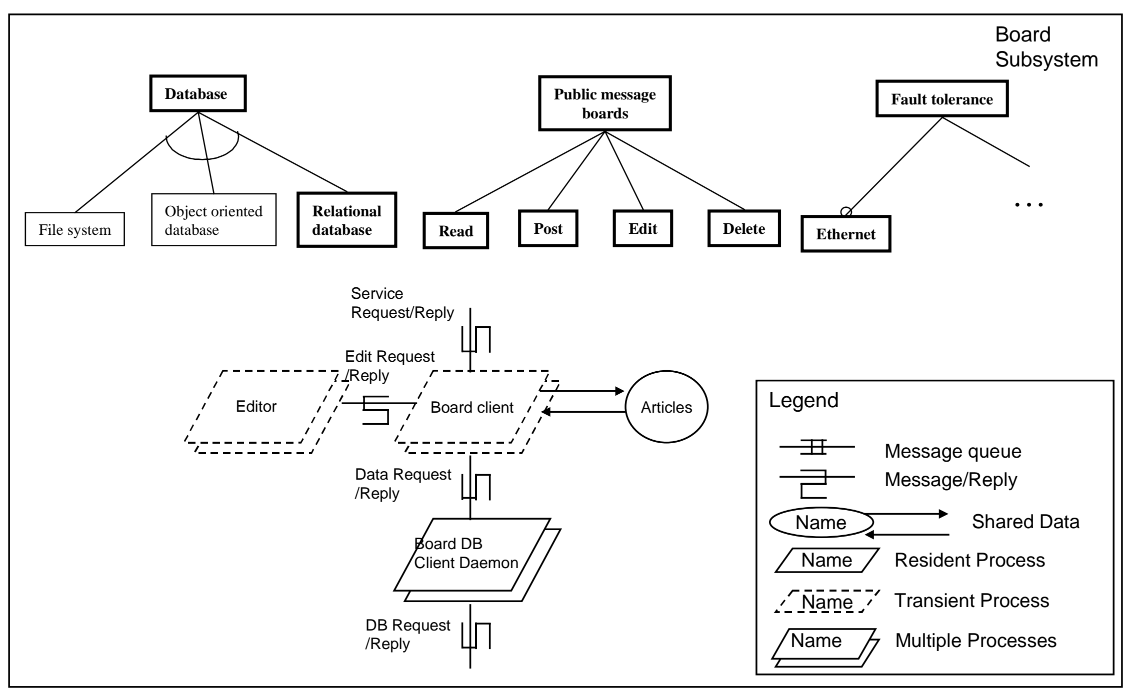

Figure 4-2: The Process Model for the Board Subsystem of the HITEL-POP EBBS.

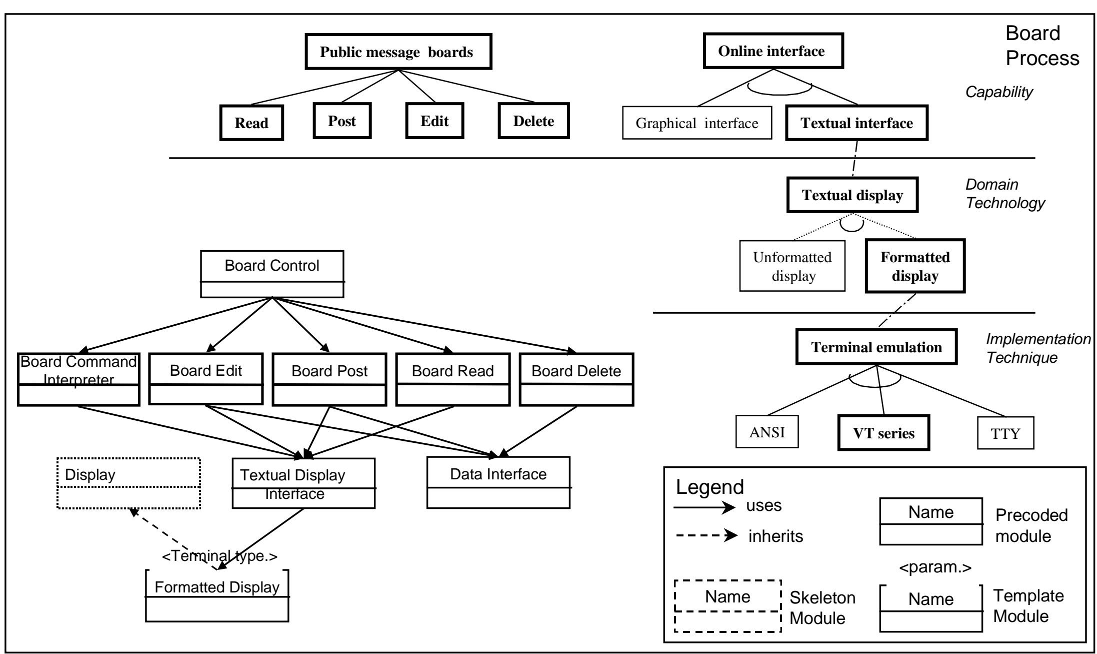

Figure 4-3: An Implementation of the Board Process of the HITEL-POP EBBS.

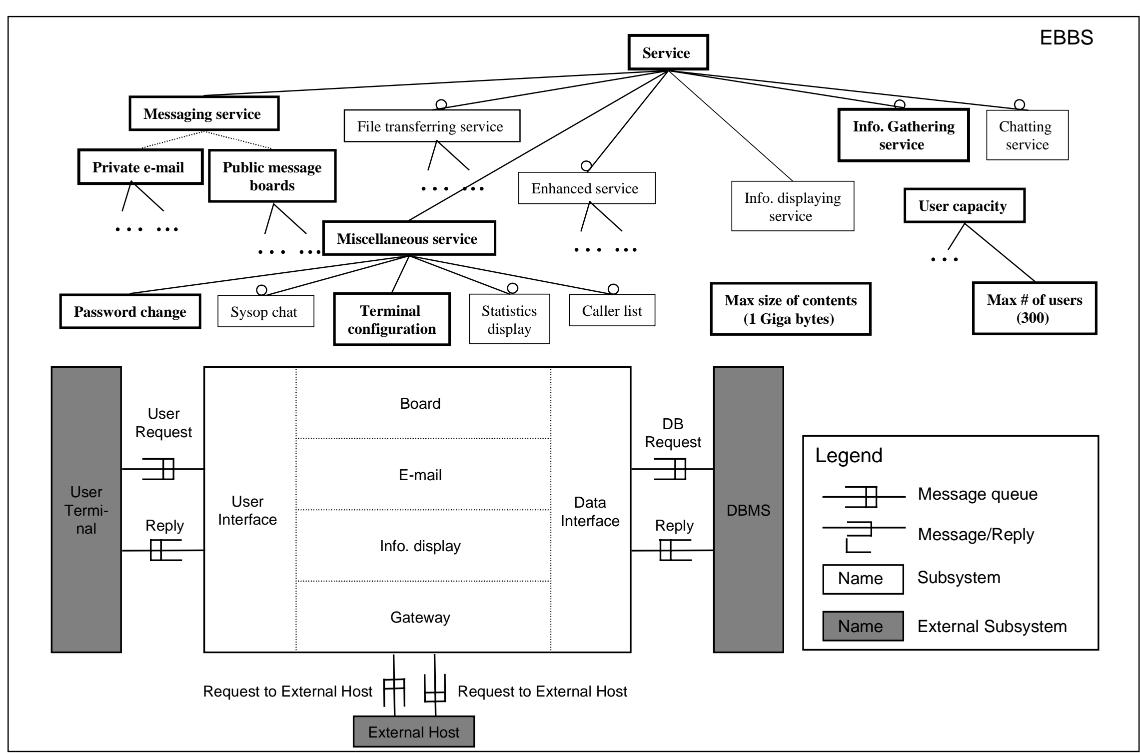

Figure 4-4: The Subsystem Model for the LION EBBS.

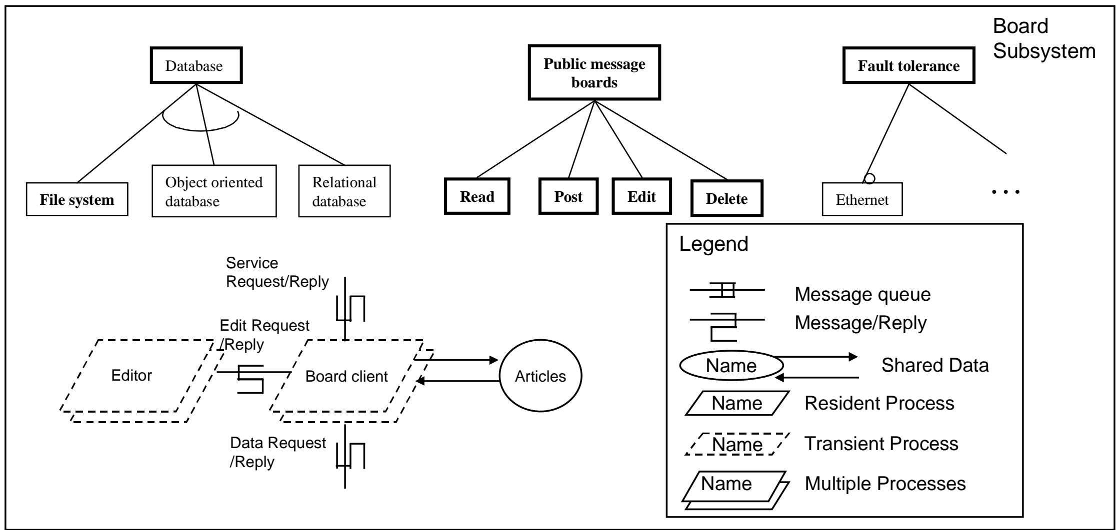

Figure 4-5: The Process Model for the Board Subsystem of the LION EBBS.

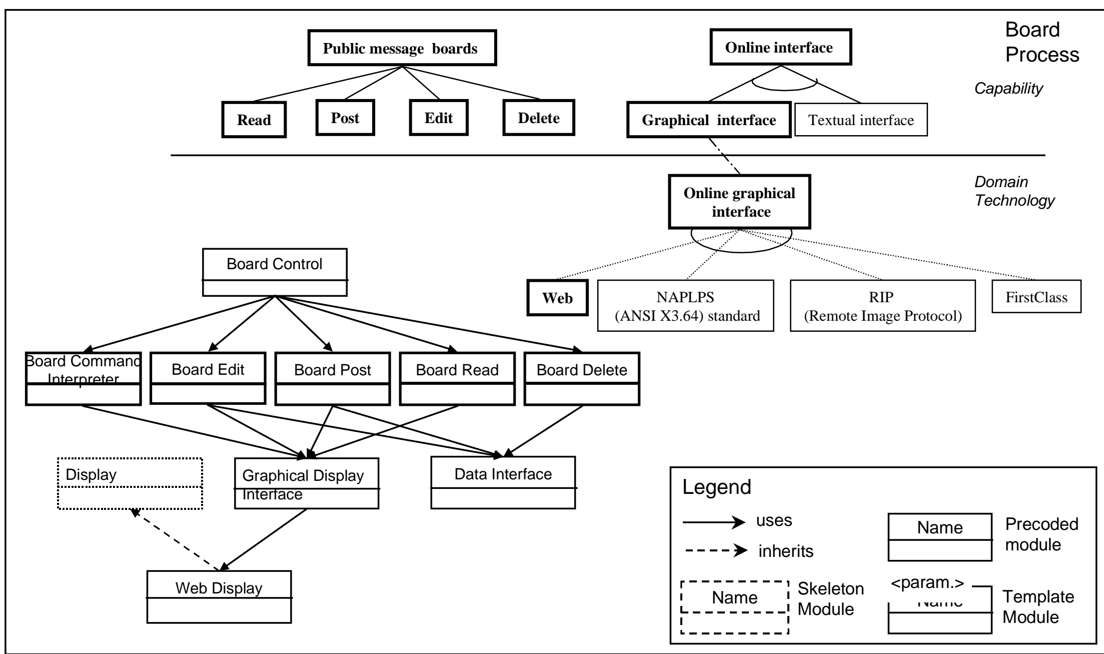

Figure 4-6: An Implementation of the Board Process of the LION EBBS.

4.2 Architecture Model Selection and Application Development

Through the feature selection process described above, a corresponding reference architectural model is immediately obtained in the artifact space. Once a reference architecture is identified, reusable components are easily found by following the specifications and methods (e.g., selection of pre-coded components, filling in skeletons, or instantiation of parameterized templates) encoded in the modules.

Figures 4-1,2,3 and Figures 4-4,5,6 show two different reference architectures mapped from two slightly different feature selection specifications. They both represent electronic bulletin board applications, although one was "larger" than the other in terms of the maximum possible users and the maximum content size of postable bulletins. In the respective subsystem configuration, by the difference in non-functional capability feature values ("Max # of Users" with a value of 500,000 vs. 300, and "Max Size of Contents" with a value of 100GB vs. 1GB), the HITEL-POP[3] is instantiated with a distributed system with "Board", "Chat", and "Gateway" as separate subsystems, while the smaller LION[4] is modeled as a single monolithic system. As for the process level architecture model, the HITEL-POP's two operating environment features, "Database" and "Fault Tolerance", have resulted in a separate DB Client Daemon process (a mediating process between the board and the data base processes), while the simpler LION system connects the board process directly to the file system. Similarly, at the module level, HITEL-POP's requirement for a "Textual display", is further implemented by providing both "ANSI" and "VT-

[3] HITEL-POP is a commercial EBBS system widely used in Korea.

[4] LION is a small student run EBBS system used at our university.

Series" terminal handling modules, both integratable into the application as template. LION's "Online graphical interface" and associated "Web" implementation features are used to call for a "Web Display" precoded reusable code.

Inter-module Communication Model / Architectural Model	Data Flow		Control Flow
	Message queue	Message/reply	Event
Distributed	Socket	RPC CORBA	RPC Socket CORBA
Centralized	Shared Memory	Local Procedure Call (LPC)	Signal

Figure 4-7: Architectural Component Connectors.

```
int createQueue(char *dst, int port)
{
   int res, cid;
   ...
   res = fcntl(cid, F_SETFL, fcntl(cid, F_GETFL, 0) | O_NDELAY);
   ...
   if (dst != NULL) setAddr(&saddrin, dst, port);
   return cid;
}

int writeQueue(int cid, int len, char *buf)
{
   int sent, nPort;
   char *szHost, *szPort;
      ...
   sent = sendto(cid, buf, len, 0, (const struct sockaddr *) &saddrin, sizeof(struct sockaddr));
      ...
   return sent;
}
```

Figure 4-8: An Implementation of the Stream Connector with Sockets.

Figure 4-7 shows examples of various types of reusable "connector" codes available for different architectural models and feasible feature specifications. For instance, for a particular data communication implementation of a "Stream" in a "distributed" architecture, one might choose to use reusable sockets (illustrated in Figure 4-8). FORM's architecture based software reuse paradigm, therefore, avoids and minimizes problems that often occur during integrating reusable components in application development.

5. Conclusion

The feature-oriented domain analysis method FODA is extended in FORM to cover the entire spectrum of domain and application engineering, including the development of reusable architectures and code

components. FORM is characterized by the construction of a domain model in terms of domain features, called a feature model, and use of this feature model in the engineering of reusable architectures and components.

The feature model has also proven effective in identifying what can/should be made reusable. At the service level, alternative features of the model generally indicated needs for parameterization of architectures or components, or the development of alternative architectures or components. Alternative features of domain techniques generally showed the need for the definition of a virtual machine to encapsulate specific techniques and develop reusable components.

By applying various engineering principles, particularly the abstraction of conceptual interfaces and separation of implementation details, the resultant reference models are amenable to easy re-configuration. ***In our experience with the EBBS, providing a rich set of architectural connectors (e.g., socket, LPC, etc.), was vital in creating an architectural model most flexible to changing operating environment and available technologies.***

Our experience with the application of the method to the electronic bulletin board systems (EBBS) domain, the private branch exchange (PBX) domain, and other smaller domains have demonstrated the effectiveness of the method in identifying the commonalties and also evaluating the maturity of the application domain. The degree of standardization of service description or technical terms in the EBBS domain indicates its maturity as an application domain, and the feature analysis was used effectively by analyzing terms and their meanings. Application programs and documents were used only for validation and making enhancements afterwards. A feasibility study [Kang *et al.* 1997] of applying FORM to the PBX domain has shown that the interactions among features are analyzed and problems could be identified early in the lifecycle. Also, features could be packaged in software modules and interactions between features could be made visible and traceable in the code. A full-scale reengineering of PBX software applying FORM is now on the way with our industry partner. In addition, we are currently performing a feasibility study of applying the method to the elevator controller domain with other industry partner.

FORM is currently semi-formal in that it only provides language constructs for modeling architectures and integrating architectural components, and rigorous analysis of models can not be performed with the method. We plan to formalize the language construct so formal analysis of the applications derived from the reusable architectures and components can be made. As the feature model captures rationale for feature selection, we plan to extend the method further to support development of an application optimal for a specific problem context.

References

Aho, A. V. and N. Griffeth (1995), "Feature Interaction in the Global Information Infrastructure," In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM SIGSOFT, Washington, DC., pp. 2-5.

Batory, Don, Lou Coglianese, Mark Goodwin, and Steve Shafer (1995), "Creating Reference Architectures: An Example from Avionics," In *Proceedings of the ACM-SIGSOFT Symposium on Software Reusability*, Seattle, WA.

Biggerstaff, Ted J. (1989), "Design recovery for maintenance and reuse.," *IEEE Software 22*, 7, 36-49.

Buschmann,T., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal (1996), *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, New York, NY.

Bryant, Alan D. (1994), *Creating Successful Bulletin Board Systems*, Addison-Wesley Publishing Company, Reading, MA.

Cameron, E. J. and H. Velthuuijsen (1993), "Feature Interactions in Telecommunications Systems," *IEEE Communications Magazine* 31,18-23.

Fekete, Alan (1993), "Formal Models of Communication Services: A Case Study," *IEEE Computer 26*, 8, 37-47.

Floch, Jacqueline (1995), "Supporting evolution and maintenance by using a flexible automatic code generator," In *Proceedings of the 17th International Conference of Software Engineering*, Seattle, WA., pp. 211-219.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Design*, Addison-Wesley, Reading, MA.

Gomaa, Hassan (1993), *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley, Reading, MA.

Gomaa, Hassan, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tarakoli (1994), "A Prototype Domain Modeling Environment for Reusable Software Architectures," In *Proceedings of the Third International Conference on Software Reuse: Advances in Software Reusability*, Rio de Janeiro, Brazil, pp. 74-83.

Griffeth, N. D. and Y. Lin (1993), "Extending Telecommunications Systems: The Feature-Interaction Problem," *IEEE Computer* 26, 8, 14-18.

Kang, Kyo C., Sholom G. Cohen, James A Hess, William E. Novak, and A. Spencer Peterson(1990), "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Kang, Kyo C., Jaejoon Lee, Sajoong Kim, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin (1997), "Feature-Oriented Engineering of PBX Software for Adaptability and Reusability,", Submitted to Special Issue on Managing Feature Interactions in Telecommunication Software Systems, IEEE Transactions on Software Engineering.

Luckham, David C., John J. Kenny, Larr M. Augustin, James Vera, Doug Bryan, and Walter Mann

(1995), "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering, Special Issue on Software Architecture* 21, 4, 336-335.

McKenny, Paul E. (1996), "Selecting Locking Primitives for Parallel Programming," *Communications of ACM 39*, 10, 75-82.

Mettala, Erik and Marc H. Graham (1992), "The Domain-Specific Software Architecture Program," Technical Report CMU/SEI-92-SR-9, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Moriconi, Mark, Xiaolei Qian, and R. A. Reimenschneider (1995), "Correct Architecture Refinement", *IEEE Transactions on Software Engineering 21*, 4, 356-372.

Neighbors, J (1984), "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering SE-10*, 5, 564-574.

Palmer, J. D. And Y. Linag (1992), "Indexing and Clustering of Software Requirements Specifications," *Information and Decision Technologies 18*,4, 283-299.

Perry, D. E. and A. L. Wolf (1992), "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes 17*, 4, 40-52.

Shaw, Mary and David Garlan (1996), *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Saddle River, NJ.

Sloane, A. M. And J. Holdswirth (1996), "Beyond Traditional Program Slicing," In *Proceedings of the 1996 International Symposium on Software testing and Analysis*, ACM SIGSOFT.

Tracz, Will (1990), "Three Cons of Software Reuse," In *Proceedings of the Third Annual Workshop on Methods and Tools for Reuse*, Syracuse University CASE Center, Syracuse, NY.

Tracz, Will (1993), "LILEANNA: A Parameterized Programming Language," In *Proceedings of the 2nd International Workshop on Software Reuse*, Italy, pp. 66-78.

Wolfe, David (1994), *The BBS construction Kit*, John Wiley & Sons, Inc., New York, NY.

Zave, Palmera (1993), "Feature Interactions and Formal Specifications in Telecommunications," *IEEE Computer 26*, 8, 20-28.