Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

A Classification System for Software Reuse

by E.J. Ostertag
Advisor: J.A. Hendler

Ph.D. 92-18

A Classification System for Software Reuse[1]

Eduardo J. Ostertag
Computer Science Department
University of Maryland
College Park, Maryland 20742
e-mail: ostertag@cs.umd.edu

Abstract

Software reuse has been claimed to be one of the most promising approaches to enhance programmer productivity and software quality. One of the problems to be addressed to achieve high software reuse is organizing databases of software experience, in which information on software products and processes is stored and organized to enhance reuse.

This dissertation presents a system to define and construct such databases called the Extensible Description Formalism (EDF). The formalism is a generalization of the faceted index approach to classification in the sense that it provides facilities to define facets, terms, and object descriptions. Unlike the faceted approach, objects in EDF can be described in terms of different sets of facets and in terms of other object descriptions. This allows a software library to contain different classes of objects, to represent various types of relations among these classes, and to refine classification schemes by adding more detail supporting a growing application domain and reducing the impact of initial domain analysis.

EDF incorporates a similarity-based retrieval mechanism that helps a reuser locate candidate reuse objects that best match the specifications of a target object. Similarity between two objects is quantified by a non-negative magnitude called *similarity distance*, which represents the estimated amount of effort required to construct one given the other. Because of this, similarity distances are not necessarily symmetric.

EDF was designed to overcome the limitations of software reuse library systems based on controlled vocabularies. In particular, EDF provides a specification language based on concepts of set theory capable of representing a rich variety of software and non-software domains; it provides a retrieval mechanism based on exact matches and similarity metrics which can be customized to specific domains; and it provides a mechanism for defining and ensuring certain semantic relations between attribute values. A prototype application of this system has been implemented in ANSI C.

[1]This research was funded in part by a scholarship provided by ORDEN S.A, by the University of Maryland Systems Research Center (Engineering Research Center: NSFD CDR-8803012), and by NASA Goddard Space Flight Center grant NSG-5123 to the University of Maryland.

A Classification System for Software Reuse

by

Eduardo Jenkins Ostertag

Dissertation submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1992

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Complex computer programs such as large communications network controllers and command and control systems have placed a growing demand on the talents of software engineers as well as on existing technologies for software development. In order to keep up with the increasing complexity of today's software systems, productivity must be increased and costs reduced in all phases of the software construction process [6]. An important aspect of the projected solution to this growing demand for new software is the development of support technologies to help increase *software reuse*, that is, the reapplication of knowledge about one system to other similar systems [14]. Rather than starting from scratch in new development efforts, the emphasis must be placed on using already available software assets (e.g., processes, documents, components, tools). This approach avoids the duplication of work and lowers the overall development costs associated with the construction of new software applications.

Software reuse techniques have been divided into two major groups, depending on how new software components are constructed [16]. The first group places an emphasis on *composition*: new components are built by combining other components, hopefully without modification. Libraries of subroutines, object-oriented class hierarchies, and the UNIX pipe mechanism are all examples of composition techniques. The second group places emphasis on *generation*: new components are built by instantiating existing templates or by applying given transformation rules. The UNIX Yacc application and the Draco system [69] are examples of generation techniques.

One important characteristic common to most approaches to software reuse is that they rely, either explicitly or implicitly, on some kind of software repository or library from where the "basic building blocks" are extracted. This is most obvious in the case of composition techniques which require libraries of well organized reusable components in order to locate the proper parts to construct new components. In some way, this is also true of generation techniques, where the parts needed are not necessarily code, but templates or transformation rules. Even in the case of application generators, which include these templates and rules as part of the application, it is necessary to define large sets of parameters. The task of defining this set of parameters for a new component would be facilitated if we could start from the set of values used to generate a similar component. This ability to locate a starting point implies the existence of a library of components and their associated parameters.

The fact that software libraries are such an important aspect of most reuse systems, has made *software reuse library systems* (i.e., systems for designing, building, using, and maintaining software libraries) a very important research topic in the area of software reuse. Although many such systems have been proposed (see Chapter 2), they all suffer from one or more of the following problems:

- Restricted domain. Some reuse library systems have been designed with the sole purpose of improving reuse at the code level. Their representation language usually does not have the "expressive power" to model more abstract or complex software domains (e.g., software projects, defects, or processes).

- Poor retrieval mechanism. One essential characteristic of any software reuse library system is to allow the retrieval of candidate reuse components based on partial or incorrect specifications. This functionality requires the ability to perform similarity-based comparisons, but most systems only provide retrieval based on partial keyword matches or predefined hierarchical structures.

- Not flexible. Software reuse library systems must evolve as the level of expertise in an organization evolves. Because of this, a software reuse library system must be flexible enough to allow the incorporation of new classification schemes or new retrieval patterns, yet this is not the case in most systems.

- No consistency verification. Most software reuse library systems are based on representation models which must satisfy certain basic predicates for the library to be in a consistent state. Yet, most of these systems do not provide a mechanism for ensuring this consistency.

This dissertation proposes a classification system for software reuse called the Extensible Description Formalism (EDF) which addresses the limitations of current software reuse library systems. EDF is based on the principles of *faceted classification* which have proven to be an effective mechanism for creating such systems [75]. EDF is capable of representing a rich variety of software (and non-software) domains; provides a powerful and flexible similarity-based retrieval mechanism; and provides facilities for ensuring the consistency of the libraries.

1.1 Definitions and Framework

Software reuse library systems such as the Extensible Description Formalism (EDF) provide a very specific service to a software *reuser*: they facilitate the process of finding a set of *candidate* objects in a *software library* whose descriptions best match that of a required *target* object. Depending on the environment, a software reuser may be a person (e.g., software designer or programmer) or some other software tool that requires the services of the reuse library system. From

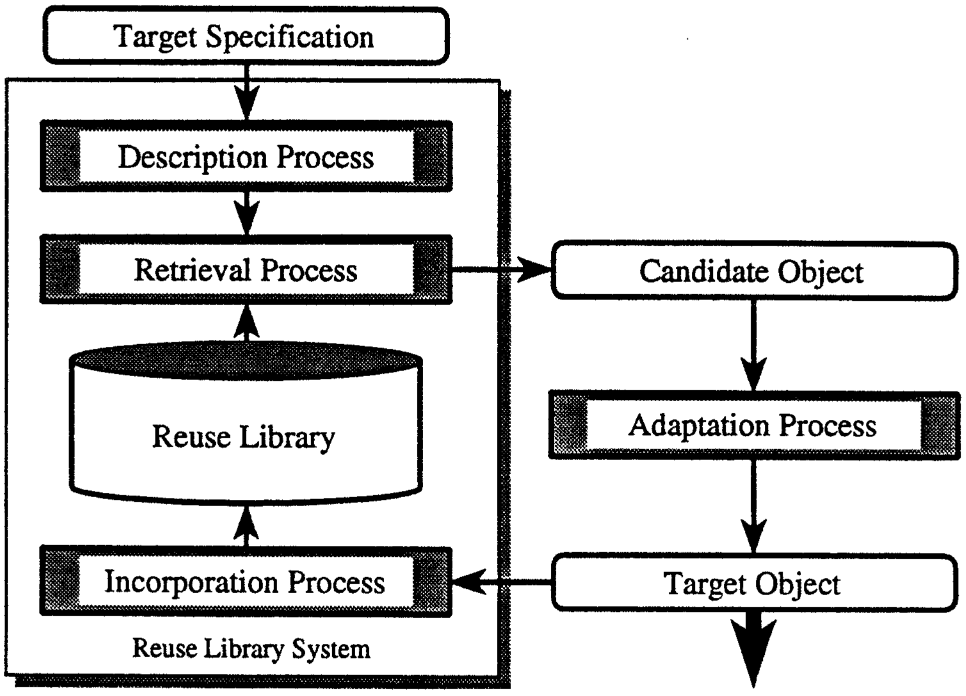


Figure 1.1: Main processes of a reuse library system

the point of view of a reuser, such a system must support three basic processes: *description*, *retrieval*, and *incorporation*.

The *description process* takes as input a specification of the required target object and produces a description of the object in terms of the language defined by the software reuse library system. Normally, the specifications of the target object are a product of the requirement of design steps of a standard software development life-cycle process. The target description produced by the description process depends on the type of representation model supported by the reuse library system. In the case of EDF, the representation scheme used is a generalization of the *faceted classification* system proposed by Prieto-Díaz [75], and is introduced in Section 1.2.

The *retrieval process* takes as input the target description produced by the description process, and retrieves from the software library a list of candidate objects that best match the characteristics of the target object. Depending on how the system is used, retrieval can be either a one-step process or an iterative one. In the case of EDF, retrieval is an iterative process which refines the original target description until the reuser is satisfied with the candidate objects produced by the system. This process is supported in EDF by a *similarity-based retrieval mechanism* which estimates the amount of reuse effort required to transform a candidate object into the required target object. The retrieval process and the mechanism for estimating similarity are introduced in Section 1.3.

The *adaptation process* generates the target object, usually by modifying the candidate objects. Although the adaptation process is not directly supported by EDF, its task is facilitated by EDF's similarity-based retrieval mechanism and the existence of a rich software library. The more similar the candidate and target objects are, the less modification effort should be involved. The adaptation process is important due to its relation to the rest of the software development process and organization. First, the objects produced by this process are used directly on the software projects being developed, and, second, they represent new software assets with potential for reuse in future projects.

The *incorporation process* is in charge of facilitating the reuse of newly produced software objects in future projects. Part of its task is to analyze the level of reusability of new objects, and select those with high reuse potential. Those objects selected for future reuse must be integrated into the software library. EDF supports this aspect of the incorporation process by allowing objects of different software domains (e.g., code, documents, defects, processes) to be stored in a single library. These object can also be interrelated, which facilitates the construction of "software experience databases"—databases that contain a body of experience accumulated within a project environment or organization [11]. EDF also provides an "assertion" mechanism to help ensure the semantic consistency of software libraries (see Section 3.1.4).

EDF and the Software Development Process

EDF is applicable to any type of organization that requires the classification, storage, and retrieval of large amounts of objects with the purpose of reuse. For example, EDF can be used to create documentation catalogs or interactive help systems.

The particular emphasis in this dissertation has to do with applying EDF to software organizations to help reduce costs in the software development process. One particular model of a software organization has been proposed by Basili [7]. Its structure and its emphasis on reusability make it an ideal environment in which to use and integrate EDF. The proposed model separates project related activities from reuse related activities in two suborganizations (see Figure 1.2). The first suborganization, called the *project*, is in charge of developing software products, taking advantage of all forms of packaged experience from prior and current developments in the organization. The second suborganization, called the *factory*, recognizes potentially reusable experience and packages it so it is easy for the project suborganization to use.

The project suborganization performs activities specific to the implementation of a software product. It analyzes the requirements and produces a high-level system design. This suborganization may follow various process models such as the Water Fall or Iterative Enhancement model. However, when the system components have been identified, they are requested from the factory suborganization and integrated into the program under development. After component integration, the project suborganization continues as usual with product quality control (e.g., system test and reliability analysis) and release.

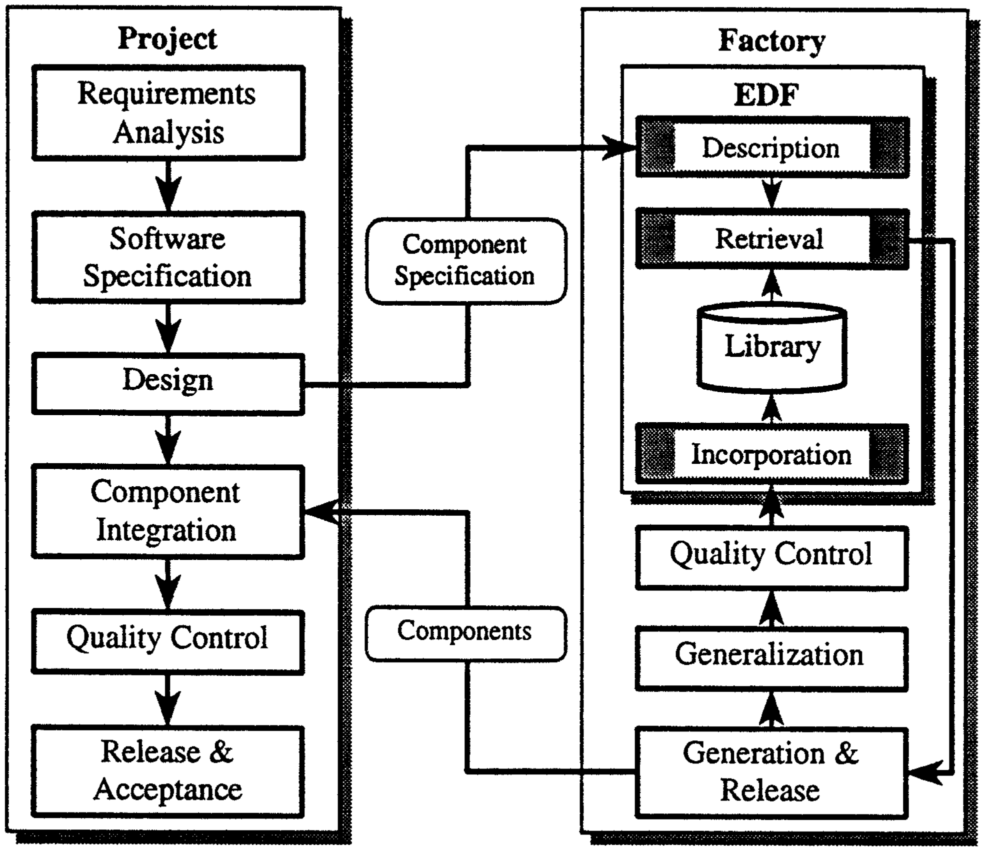


Figure 1.2: EDF and the software development process

The factory suborganization has two main tasks: to satisfy the requests for components coming from the project organization, and to prepare itself for answering those requests by producing and storing reusable software components. This enhancement process is accomplished by analyzing, generalizing, and packaging the components it generates.

The original proposal for the factory suborganization included a module called "component generation." This module has been replaced here by the Extensible Description Formalism (EDF). As Figure 1.2 shows, the EDF module provides a more detailed description of the original generation module, and it integrates naturally with the rest of the factory suborganization. What is more, EDF has been fully defined and implemented, so it could actually be used in systems that support this type of organization structure such as the CARE System developed at the Department of Computer Science of the University of Maryland [24].

1.2 Representation Model

The Extensible Description Formalism uses a generalization of the *faceted classification* approach proposed by Prieto-Díaz [75] to represent and classify software objects. The faceted index approach relies on a predefined set of *facets* defined by experts. Facets and associated sets of *terms* form a classification scheme for describing components. Component descriptions can be viewed as records with a fixed number of fields (facets), where each field must have a value selected among a finite set of values (terms).

Faceted classification has proven to be an effective technique to create libraries of reusable software components. Yet, it suffers from various shortcomings which limit its usefulness and applicability. The EDF approach to classification overcomes these limitations by extending the representation model as follows:

- Components are replaced by *instances* that belong to several different *classes*. Instances and classes are defined in terms of attributes and other classes, supporting multiple inheritance.
- Facets are replaced by typed *attributes*. Possible types are: integers, strings, enumerations, classes, and sets of the above. Having instances as attribute values allows a library designer to create relations among different instances (e.g., that `push` is a component of `stack`).
- The concept of similarity is extended to account for the richer type system, including comparisons of instances of different classes and comparisons of set values.
- Semantic attribute relations can be defined and checked using the `assertion` construct. This facility simplifies the process of maintaining the consistency of the definitions in a software library.
- An integrated language describes attributes, terms, classes, instances, distances, and their dependencies. Descriptions are type checked. The language is based on a formal mathematical model which makes it both coherent and analyzable.

A set of attributes in an EDF classification scheme defines a multidimensional space S. A point $p \in S$ may represent two different concepts: an *instance* or a *class*. An instance is an object description defined solely in terms of the attributes of S. A class defines the set of all instances whose projection onto S is p. Instances and classes are defined using logical expressions in a subset of propositional calculus. Expressions are composed of attribute-name attribute-value pairs denoted using assignments *name* = *value*. Expressions can be combined using the operators "`&`" and "`|`" which denote intersection and union of sets, respectively. There are two other basic propositions: "`in` *class*" means that the instance referred to belongs to *class*, and "`has` *name*" means that the attribute *name* has a defined attribute value. Instances are denoted by `instance(`E`)` and classes are denoted by `class(`E`)` where E is an expression.

For example, consider an EDF classification scheme for data structure packages and their operations. Each package is characterized by two attributes: "`language`" specifies the language that was used to implement the package, and "operations" lists the operations of the data structure. The class of all packages is defined in EDF as follows:

```
Package = class(has language & has operations);
attribute language   : {Ada,C,Fortran};
attribute operations : set of Operation;
```

Operations, on the other hand, are characterized by three different attributes: "function" indicates its functionality, "srcFile" indicates its source code file name, and "inPackage" indicates the package to which the operation belongs. The class of operations is defined in EDF as follows:

```
Operation = class(has function & has srcFile & has inPackage);
attribute function  : {insert,remove,traverse};
attribute srcFile   : string;
attribute inPackage : Package;
```

Given the previous definitions, we can create a small software library containing only the descriptions of one package, "stack", and one of its operations, "push".

```
stack = instance(in Package & language=Ada &
                 operations={newstack,push,pop,top});
push  = instance(in Operation & function=insert &
                 srcFile='push.ada' & inPackage=stack);
```

This example illustrates some of the advantages of EDF over faceted classification. First, an EDF library may contain descriptions of different types of objects (e.g., packages and operations). Second, attribute values are not limited to just terms; they can also include strings (e.g., `srcFile`) and instances (e.g., `inPackage`). Third, object descriptions may be interrelated, forming arbitrary hierarchical classification schemes. In particular, this example defines a one-to-many relation between packages and operations (via attribute `operations`), and a one-to-one relation between operations and packages (via attribute `inPackage`). There is no limit to the depth of this type of hierarchy. For example, we could add a third level by defining a new object class called "Application" to characterize application programs in terms of the packages used in their implementation.

A complete description of the syntax and semantics of the EDF specification language is given in Chapter 3. In addition, Chapter 4 presents full descriptions of various taxonomies for different software domains, showing EDF's representation power.

1.3 Similarity-based Retrieval

Having the ability to represent various types of objects within the same software library allows EDF to model a wide range of domains. Yet, this representation power would be of little value without a mechanism for retrieving objects from a library based on partial or (possibly) incorrect specifications.

Electronic library catalogs usually provide their users with applications that allow them to retrieve documents based on partial matches of keywords. That is, given a list of terms, the system retrieves all those entries whose associated terms match one or more of the terms of the query. The problem with these systems is that if an exact match does not exist in the library, the query fails. Entries whose description may be *similar* to that of the query are not consider by the system, no matter how close they are.

Similarity-based retrieval is particularly important in environments such as software organizations where reusability of old software components is an important cost reduction mechanism. The reason being that a successful query based on exact matches would only indicate the existence of identical components, which is a situation that seldom occurs in these types of environments.

Several approaches for doing similarity-based retrievals have been proposed in the literature (see Chapter 2), most of which are very domain specific. In particular, systems based on faceted representations of objects provide mechanisms for doing similarity-based retrieval which are mostly derived from Prieto-Díaz's work [75]. In his system, software components are retrieve based on the degree of similarity of facet terms. Similarity among some of the terms of a facet is encoded as a integer value called *conceptual distance*. These distances are arranged in *conceptual distance graphs* which are used by the system to compute the similarity of terms that have not been assigned an explicit distance.

EDF extends this approach to similarity-based retrieval. It computes the degree of similarity between two instances as a non-negative magnitude called *similarity distance*, which represents the estimated amount of effort (e.g., man-hours) required to obtain one given the other. As opposed to other faceted systems, EDF is capable of computing the degree of similarity between objects of different classes (i.e., with different sets of attributes). In addition, similarity distances in EDF are not necessarily symmetric. That is, given two objects, A and B, the transformation effort required to obtain A given B is not necessarily the same as the one required to obtain B given A. This simple, but important concept that transformation efforts are not symmetric is mostly ignored by other software reuse systems.

To deal with the situation of comparing two instances with different (and possibly disjoint) sets of attributes, similarity distances are computed as an aggregate of three different magnitudes called *transformation*, *construction*, and *removal*, all of which represent estimations of reuse effort. Given a source instance, S, and a target instance, T, the transformation distance measures the effort to transform the values of attributes common to both S and T. The construction distance estimates the amount of effort required to supply those attributes found in T but not specified in S. Similarly, the removal distance estimates the effort required to eliminate those attribute values in S that are not required by the specification of T.

The transformation distance is the only type of metric found in faceted systems because components in this system are described using the same set of attributes. EDF's construction distance estimates the effort required to build things from scratch, while the removal distance measures the effort required to eliminate unwanted or unsolicited component characteristics provided by the candidate instance. These similarity distances allow EDF to sort candidate instances by decreasing similarity (i.e., increasing distance) to a given target object description. The task of ordering candidate instances is accomplished using EDF's "**query**" command. This command receives as input a logical expression E (see previous section) describing the target component, and attempts to present the reuser with a list of the best reuse candidates to build the target.

EDF also provides facilities to integrate user-supplied distance computation functions to the system. These "foreign" functions allow a library designer to customize EDF's similarity-based retrieval mechanism to different environments, and to integrate alternative metrics for estimating similarity.

EDF's similarity model is formalized in Chapter 3. Chapter 5 describes the process of assigning and using similarity distances with the purpose of selecting software reuse components extracted from the NASA SEL database. Appendix A describes in detail the use of "foreign" distance functions.

Sample Retrieval Process

This section presents, by means of an example, a high-level overview of the main steps involved in the process of retrieving components using EDF. This example assumes the existence of an EDF library of packages and components such as those previously described in page 6. The attributes used here, though, are not the same.

Our sample retrieval process will select from a library a list of candidate components that best approximate the required properties of a target component T, which is described informally as follows.

> *Component* T *is required to print a spreadsheet on a printer. The spreadsheet is stored in computer* A, *and the printer is connected to computer* B. *Both computers use UNIX as their operating system.*

For this example, component T will be separated into two subcomponents: T1, which transfers a spreadsheet from one computer to another, and T2, which prints the spreadsheet. These informal descriptions are used as a basis to describe the functionality of each required component in terms of a predefined set of attributes.

```
T1 = instance                              T2 = instance
      (Function    = Transfer &                  (Function    = Print &
       Object      = SpreadSheet &                Object      = SpreadSheet &
      Source      = Computer-A &          Device     = Printer-P &
      Destination = Computer-B);          Controller = Computer-B);
```

EDF selects the best reuse candidate component for each target description (e.g., T1 and T2). The candidates are selected based on the degree of similarity between the target and existing library component descriptions. Assume that T1 and T2 are not in the software library, but that T1* and T2* have been selected as their best reuse candidates, respectively.

```
T1* = instance                        T2* = instance
       (Function    = Copy &                (Function    = Display &
        Object      = File &                 Object      = SpreadSheet &
        Source      = Computer-A &           Device      = Terminal-V &
        Destination = Computer-B);           Controller  = Computer-B);
```

The selected component T1* can "copy" (not "transfer") a "file" (not a "spreadsheet") between computers of different types. Component T2* can "display" (not "print") a spreadsheet on a "terminal" (not a "printer"). Each candidate component can be examined by either reading its documentation or obtaining implementation specifics. If it proves to be unsuitable, other candidates can be obtained by using alternative descriptions of the target.

The candidates selected (e.g., T1* and T2*) could be merged to construct the required component T, but it would be preferable if we could find in the library a single unit combining the functionalities of these candidates in a common environment or package (see page 6). In our example, we query the EDF library for a package that must group both T1 and T2 and which must also work in a UNIX environment. That is, we look for packages that are most similar to the following description.

```
            P* = instance(System=UNIX & operations={T1,T2});
```

Assume there is no package description in the library that matches P* exactly, so EDF suggests an alternative package P+ similar to P*. P+ works under UNIX but its member components deal with matrix operations instead of spreadsheets. Among these components, T1+ and T2+ are the most similar to T1* and T2*, respectively.

```
T1+ = instance                        T2+ = instance
       (Function    = Copy &                (Function    = Display &
        Object      = Matrix &               Object      = Matrix &
        Source      = Matrix-File &          Device      = Terminal-V &
        Destination = Matrix-File);          Controller  = Computer-B);
```

There are now two alternatives to construct the required component T. We can merge T1* and T2* into one software unit, requiring the overhead of creating joint data structures, definitions, etc. Alternatively, we can use T1+ and T2+ which are already part of the package P+, and therefore share certain properties (e.g., memory management or programming language) that may reduce the overall effort to construct T compared to using the unrelated components T1* and T2*.

1.4 Contribution of this Work

As explained earlier, current software reuse library systems based on the faceted index approach to classification suffer from one or more of the following problems: they are applicable to a restricted set of domains; they posses poor retrieval mechanisms; their classification schemes are not extensible; and/or they lack mechanisms for ensuring the consistency of library definitions. The primary contribution of this dissertation is the design and implementation of the Extensible Description Formalism which overcomes these problems.

- EDF is applicable to a wide range of software and non-software domains. The EDF specification language is capable of representing not only software components at the code level, but it is also capable of representing more abstract or complex software entities such as projects, defects, or processes. What is more, these software entities can all be made part of one software library and can be arranged in semantic nets using various types of relations such as "is-a", "component-of", and "members-of". The EDF representation model is described in Section 3.2.1.

- EDF has a powerful similarity-based retrieval mechanism. One essential characteristic of any software reuse library system is to allow the retrieval of candidate reuse components based on partial or incorrect specifications. EDF provides a retrieval mechanism that selects candidate components based on the degree of similarity of their associated library descriptions. This mechanism is based on an iterative refinement process in which components at different levels of granularity can be retrieved. It also includes facilities that allow a library designer to customize the retrieval process by including domain specific functions coded in standard programming languages such as ANSI C. EDF's similarity model is described in Section 3.2.2.

- EDF provides an extensible representation scheme. A software reuse library system must be flexible enough to allow representation schemes to evolve as the needs and level of expertise in an organization increases. The EDF specification language provides several alternatives to extend or adjust a taxonomy so as to allow the incorporation of new objects into the library without having to reclassify all other objects. The methods for adjusting a taxonomy are described in Section 3.1.

- EDF provides a consistency verification mechanism. Most software reuse library systems are based on representation models which must satisfy certain basic predicates for the library to be in a consistent state. The EDF specification language includes an "assertion" mechanism whose purpose is to help specify and ensure the consistency of the object descriptions contained in a library. This mechanism is introduced in Section 3.1.4.

In short, EDF addresses the main limitations of current faceted classification systems by extending their representation model and incorporating a retrieval mechanism based on asymmetric similarity distances.

1.5 Overview of the Dissertation

Chapter 2 presents a survey of research related with software reuse techniques. This survey covers two related areas: systems that provide functionalities similar to that of EDF, and alternative methods for computing similarity.

Chapter 3 presents a complete description of the concepts that compose Extensible Description Formalism. These concepts are first explained informally by constructing a sample classification taxonomy for software components. The chapter concludes by presenting a formal definition of the syntax and semantics of the EDF specification language.

Chapter 4 demonstrates EDF's representation power by presenting taxonomy definitions of various software domains. First, I include taxonomies for describing components of a commercial software library called the EVB GRACE library and a library for Command, Control, and Information System developed at Contel Technology Center. I also include a taxonomy for representing software defects, and explain how an EDF library of software defects can help a system tester. Finally, I present two taxonomies, one for describing software process models and the other for software evaluation models.

Chapter 5 develops a complete software reuse library based on information obtained from the NASA SEL database, which contains several hundred descriptions of projects, systems, and components developed at NASA Goddard Space Flight Center. The emphasis in this chapter is given to the process of designing a similarity model with the purpose of selecting suitable reuse candidates.

Chapter 6 contains concluding remarks. I present a summary of the contents of this document and describe the advantages of EDF over other reuse library systems. I finish by describing some areas of future research.

Appendix A describes the main aspects of a prototype application that implements the different concepts defined by the Extensible Description Formalism. This prototype was implemented in ANSI C and has been ported to several UNIX platforms.

Chapter 2

Related Work

In its most general form, *reuse* involves using previously achieved results in a new situation, and *reusability* denotes the degree of freedom with which previous results may be used in a new situation [85]. Two levels of software reuse are usually considered: reuse of ideas and knowledge, and reuse of software products and their components. That is, a *reusable software component* may be any product of the software development process—a unit of code, a design specification, a test case, etc. Use of a component more than once can mean anything from informal reuse of a design by its designer to the widespread use of a large software package. The component can be used unchanged, or it can be modified to fit the new application.

Recent research focuses on exploring new directions aimed at formalizing and standardizing the activities and procedures necessary for reuse. Significant contributions have been reported in the areas of software cataloging and retrieval, program synthesis from reusable components, reuse measurement, and domain analysis [14, 15]. Research in these areas is creating the foundation for development of tools and methods to make reuse practical and effective [87, 88].

Approaches to software reusability have been broadly classified in two categories according to the nature of the reuse components: *passive* and *active* [13]. Passive reuse components are almost immutable, ready building blocks. Their reuse involves interconnecting appropriate components so as to assemble the required program. Depending on the readiness of the reuse component, two extreme forms can be distinguished: direct reuse (black box), and reuse after modification (white box). Between these two extremes, there are reuse methods which require adapting the reuse component by some predefined mechanism such as parametrization, specialization, and others. The application of passive reuse components requires solving the following problems: identification, description, classification, storage, retrieval, and use of the component. It is also necessary to be able to evaluate, modify, and combine the components into larger systems.

Active reuse components, on the other hand, are software patterns that take malleable and diffuse forms. Both programs and results are reusable. They can be represented as simple and widely accessible formalisms (such as BNF), concepts (objects and methods in object-oriented languages), software models (denotational semantic models), etc. The emphasis in active software reuse, as opposed to passive reuse, is on the construction of reusable components, and not their manipulation. Sidorov [85] explains that operation of active reuse tools can be represented by a mapping of the form $M \times T \xrightarrow{I} P$, where M is the set of models of the application domain, T is the set of transformations, P is the set of new programs, and I is the set of instructions driving the generation process. In active reuse, the elements of all three sets, T, I, and M, can be reused.

The remainder of this section presents a review of the search work done in the area of software reusability. Emphasis has been placed on three main topics. First, Section 2.1 covers different methods and tools for representation of reusable components. Second, Section 2.2 deals with the problem of effort estimation and its relation to similarity computation. Finally, Section 2.3 describes a system called AIRS (AI-based Reuse System) that was used as the basis to design EDF.

2.1 Representation of Reusable Components

One of the problems affecting software reuse has to do with how to represent reusable software. Many methods for representing software components for reuse have been proposed, including traditional library and information science methods and knowledge-based methods. Comprehensive surveys of representation techniques have been presented in the literature [51, 14, 15, 39]. In this section I summarize these approaches and describe systems in which they have been used.

2.1.1 Library and Indexing Methods

A major goal of reuse library developers is to provide ways for users to search for software components that satisfy a set of requirements. Traditional libraries, catalogues and indexes are used for this purpose. One such catalog is the *ALA glossary of library terms* [93, 66] that records, describes, and indexes resources of a collection. Indexes, for example *Reader's Guide To Periodical Literature*, also provide information about documents and other items, yet traditionally library indexes are less descriptive and are usually designed to help users find documents in a collection.

Indexes make use of specialized indexing languages. For example, the *Dewey Decimal System* defines the location of an item in a library by describing its subject and content [33] numerically. Many indexing languages have been developed for library and information retrieval. They are often classified as either having a *controlled* or *uncontrolled* vocabulary. In controlled vocabularies, a limited set of terms describes information items, while in uncontrolled vocabularies there are no restrictions on term selection.

Controlled Vocabulary

Controlled indexing vocabularies ensure that the terms used by indexers and searchers are the same. That is, a controlled vocabulary is based on a list of acceptable terms. Unacceptable terms (e.g., synonyms) are also usually listed. Terms are generally derived using one of two methods [58]: *literary warrant* and *user warrant*. In literary warrant, index terms are derived from examination of the subject area; a term is used only if it occurs often enough in the literature. In the case of user warrant, index terms are included if they are of interest to the user population. Two major forms of controlled vocabularies exist: *keyword systems* and *classification systems*.

Keyword Systems. Keyword indexing consists of natural-language words and phrases that are used as terms for describing an information item. As with other controlled systems, unacceptable terms are also made available (e.g., synonyms). Terms in keyword systems are arranged primarily in alphabetical order and not by class, and, therefore, these systems provide no information about relationships between terms.

The terms *subject heading* and *descriptor* are sometimes used as synonyms of *keyword*. Subject headings are like enumerated classification systems, where indexes do not synthesize classes from terms. Synthesis is not allowed because all terms are created before the system is used. (One exception to this is found in the systems developed by Coates for the British Technology Index and Precis (Preserved Context Indexing System) developed by Austin [38].)

Descriptors, on the other hand, are controlled keywords designed to allow searchers to synthesize terms using Boolean operations. Descriptors are not normally used to represent classes; it is only after Boolean operations that descriptors can be synthesized to form composite concepts. The main difference between subject headings and descriptors is in the way they are used. A descriptor is used in conjunction with other descriptors, while a subject heading is designed to stand alone [58].

Information about relations between terms has been usually more limited in keyword systems than in classification systems. One way to overcome these limitations has been the use of a *thesaurus*. A thesaurus presents searchers with guidelines for combining terms. This is done by providing an alphabetic list of acceptable terms and their synonyms, and descriptions of relationships between the terms. One example of this technique is the *DTIC retrieval and indexing terminology* thesaurus [31].

Subject headings and descriptors have been used to construct systems for software reuse, but the literature usually does not make a distinction between the two. One example is a system of controlled terms for searching software components by Arnold and Stepoway [4]. It is not clear what form their list of terms has or whether Boolean operations or other forms of synthesis are allowed in their system.

Classification Systems. Classification systems group items into classes which are associated to a set of terms. For example, if the term `sort` indexes a component, then this component is grouped with all other components described by the term `sort`. In this sense, indexing terms can be considered class labels. There are two types of classification systems: *enumerated* and *faceted*.

In enumerated classification, all possible class labels describing a domain area are listed in a "classification hierarchy". Classes in these systems must satisfy the following conditions: they must be mutually exclusive, they cannot be combined to form new classes, and they must have at least a partial hierarchical order. The Dewey Decimal System [33] is a well known enumerated classification scheme, as is the computer science classification scheme used by *Computing Reviews* [3].

The major advantage of enumerated systems is their hierarchical structure. A well defined structure makes it easy to interpret the relationships between terms. The disadvantage is that creating a well defined classification structure requires an exhaustive analysis of the domain area. Also, these systems only provide one view of the relationships, and modifications are difficult because changes cannot be made without totally restructuring the hierarchy. Another problem with enumerative schemes is traversing the hierarchical tree to find appropriate class. In the Dewey Decimal System, for example, the title "Structured System Programming" could be classified in any of the following classes: system analysis (001.61), software (001.642.5), systems (003), systems analysis (620.72), or systems construction (620.73). To compensate for such ambiguity cross references are established, but cross referencing is a cumbersome and error prone process.

Faceted classification, on the other hand, is a more flexible type of classification system aimed at overcoming the rigidity of enumerated classification. Faceted classification was introduced by Raganathan in the late 1930s [80, 38, 22] and is widely used in libraries throughout Europe and India. In this classification system, a domain area is decomposed into basic terms (also called foci or elemental classes). These terms are then conceptually arranged into *facets*. For example, a faceted classification for software [75] might group terms such as "sort" and "append" into a facet called "function", or terms such as "array" or "string" might be arranged in another facet called "object".

In faceted classification, classes may be derived by synthesizing basic terms from different facets. For example, a class "append string" may be created by combining the terms "append" and "string" from the facets "function" and "object", respectively. This ability to derive classes during indexing is what distinguishes faceted systems from enumerated systems. In enumerated systems, the class "append string" would have to be explicitly listed in the classification hierarchy before an item can be classified. Faceted classification was called *analytico-synthetic* classification [38] because it allows indexers dynamic analysis of domains and the synthesis of terms into classes.

One major advantage of faceted classification over enumerated classification is its ability to synthesize classes. This means that not all classes need to be determined when developing a system. Faceted systems are also easier to update and modify because facets can be arranged independently of others. A disadvantage of faceted classification, though, is that systems with large numbers of facets can be difficult to use efficiently. Users may be forced to search through many facets before finding the right combination of terms to describe an object.

Enumerated classification systems have been used to implement classification systems for software reuse. A comprehensive survey of early enumerated systems for reuse was done by Prieto-Díaz [75]. One of the first classification schemes for computer programs was the IBM SHARE System [75]. A modification of this scheme was proposed by Bolstad [19]. Bolstad's scheme has served as the basis for several other collections, such as the National Bureau of Standards' Guide to Available Mathematical Software (GAMS) [25], and the enumerated system used by the International Mathematics and Scientific Library (IMSL) [49]. Other examples of enumerative schemes for software libraries include NASA's COSMIC Software Catalog [23] which uses a flat list of 75 NASA subject categories for classifying its programs, and Toshiba's PROMISS system [64] which uses a list of function codes for categorizing its registered programs.

Faceted classification systems for software reuse are less common than enumerated systems. One of the earliest and best known faceted systems was developed by Prieto-Díaz [75, 76]. His system relies on a predefined set of keywords extracted by experts from program descriptions and documentation. These keywords are arranged by facets into a classification scheme and used as standard descriptors for software components. A thesaurus is derived for each facet to provide vocabulary control and to add a semantic component to retrieval. Keywords can only be used within the context of the facet they belong to and ambiguities are resolved through a thesaurus.

An important component of Prieto-Díaz's system is the use of a *conceptual distance graph*. Conceptual distances between items of each facet are used to evaluate their similarity, which is used in turn to evaluate the similarity between required software specifications and available components. Conceptual distances are assigned based on experience, intuition, and common sense. More recently, Gagliano *et al.* [42], have proposed a method to compute conceptual closeness based on statistical analysis. Frequencies of "perceived similarity" obtained by running experiments with controlled groups of individuals are used to compute a "dissimilarity coefficient". This coefficient is then used to build conceptual distance graphs for a set of users.

Prieto-Díaz's system is reported to be very effective in retrieving components, but the construction of conceptual graphs is labor intensive and has not been formalized yet. One of the main restrictions imposed by the system is that each description must be defined using all facets that belong to the classification scheme. This implies that components that are naturally characterized by different sets of facets must be defined in terms of a unique set, thus forcing component descriptions to use facets that are meaningless to the particular component. This also implies that, in order to add a new facet to a particular scheme, the definition of all components in the database must be extended to include this facet.

Uncontrolled Vocabulary

In an uncontrolled vocabulary, no restriction is placed on what terms can be used to describe an item. The vocabulary terms can be extracted from any source, usually from the indexed objects themselves. The vocabulary may include single words or phrases from the text of objects, and terms sometimes have weights to define their relative importance. Methods to derive these weights have been described [83].

Potential advantages of uncontrolled vocabulary include reduced cost and specificity. Because terms are usually extracted automatically from the text of objects, indexing is usually cheaper than if done by humans. Also, since terms are unrestricted, indexing can be done as specific as possible.

Uncontrolled vocabulary systems have been used to develop tools for software reuse. One example is Bell Labs' CATALOG system [40]. This system is based on information retrieval (IR) techniques for locating components in the library. Either the component itself or an associated component descriptor is indexed by the IR system, so that each (non-trivial) word can be retrieved in a search. There is no manual classification of the components into pre-defined categories. A component descriptor template is suggested that includes programmer defined keywords, descriptive text, and many of the same attributes described above (author, date, environment, etc.). A user can search for an arbitrary word, phrase, or fragment within specific fields or across the component or descriptor as a whole.

Other examples of an uncontrolled vocabulary system include the Intermetric's Reusable Software Library (RSL) [23] which uses free-text indexing to create a hierarchical category code for describing component functionality of Ada functions, procedures, packages, and programs. A KWIC index is another form of free-text index which was developed to overcome the problem that most free-text index systems have with loosing syntactic and semantic relationships among terms. In a KWIC index each significant index term is presented in the phrase it occurs in the original text.

Maarek, Berry, and Kaiser [63] have proposed the concept of *lexical affinities* among pairs of words as an effective means to extract both lexical and term frequency information from software documentation. Results from their experimental system (GURU) demonstrate that this approach may be able to substantially improve keyword based library systems.

An approach that borrows from both faceted and free-text method has been proposed by Embley and Woodfield [34]. They propose a library of abstract data-types (ADT) which are classified using special descriptors. A descriptor defines an ADT using keywords, facets, and list of aliases. The system allows the user to define explicit relations among different ADT's, and provides some built-in relations such as "depends-on", "close-to", and "generalizes" which can be derived automatically from the values of the ADT's, facets, and keywords.

2.1.2 Knowledge-based Methods

One alternative to library classification methods are various knowledge representation methods developed in the field of Artificial Intelligence (AI). Many of these methods [97] have been used for representing reusable components.

Two factors must be considered when evaluating knowledge representation:

adequacy and *heuristic power*. Representational adequacy refers to how much information about the represented object can be expressed in a formal way. Heuristic power, on the other hand, refers to the kinds of inferencing that can be achieved with the representation.

One important aspect of knowledge-based representations is that they offer powerful ways to express relations between components. On the other hand, one potential disadvantage with these methods is that the knowledge acquisition problem sometimes proves to be very difficult [47]. There are various approaches to knowledge representation, among them rule-based, semantic net, and frame-based systems.

Rule-Based Systems

Rule-based systems are well known representation systems mainly because of their use in Expert Systems. Rules are normally composed of two parts: an *antecedent* and a *consequent*, both of which take the form of logical expressions. Whenever the antecedent of a rule is true, its consequent part is activated or assumed true. For example, a "bubble sort" component could be represented by a rule of the form: "IF functionality needed is a sort AND required language is Pascal THEN use `bubbleSort.p`".

A system for selecting code modules for reuse has been described by Rosales and Mehrotra [82]. Their system is called MES, and is targeted for selection of components in transmission systems developed in PL/1. Another rule-based system has been proposed by Bollinger and Barnes [17]. They used a Prolog-based system for finding reusable parts from an electric load monitor program written in Ada.

Fitzgerald and Mathis [37] developed an expert system to help select Ada parts. Their first version used the GRACE (Generic Reusable Ada Components for Engineering) parts, which is a set of data structures similar to those developed by Booch [20]. They later developed a new version which added parts from previous projects in their organization (Contel). One important characteristic of their system is that it maintains a log about failed searches which can be later used by the library administrator to expand or adjust the knowledge base.

Semantic Net Systems

A semantic net is a directed graph whose nodes correspond to conceptual objects and the arcs represent relationships among these objects. It is usually easy to express knowledge using semantic nets, that is, they have a good representation adequacy. On the other hand, the heuristic power of some basic semantic nets is poor. This problem is sometimes overcome by adding first-order logic semantics.

One reuse system based on knowledge representation is described by Solderitsch *et al.* [86]. The system is called RLF and uses nodes to represent concepts from a domain, and two types of arcs: ISA arcs and role arcs. ISA arcs represent class membership while role arcs describe non-class relationships among concepts.

This system has been used, so far, to construct three knowledge bases of software components.

The use of semantic nets as a tool to help the domain analysis process and to represent complex taxonomies has also been proposed by Prieto-Díaz [76]. He presents an example of a semantic net representation of the different concepts of a parser for a compiler such as: variable names, symbol table, and parsing method, code generator, etc. Several types of relations are used: is-a, represented-by, uses-method, and others.

Frame-Based Systems

Frame-based systems make use of *frames* to represent conceptual objects. A frame is a data structure composed of slots and fillers, similar to "record" structures found in most traditional programming languages. The main difference is that frames can be arranged in hierarchical structures which allow frames to inherit slots, and optionally fillers. Because of this, frame-based systems are usually compared to Object-oriented systems [30, 89, 67, 50, 54]. One difference between these two approaches is that in most frame systems slots may be attached to procedures, thus allowing dynamic computation of slot values and assignment of default values [26].

Several systems for software reuse have as their foundation frame-based systems. Devanbu *et al.* [32] developed LaSSIE, a classification-based software information system. LaSSIE incorporates a large knowledge base, a semantic retrieval algorithm based on logical inference, a powerful user interface with a graphical browser, and a natural language parser. LaSSIE is intended to help programmers find useful information about large software systems like AT&T's System 75 PBX with over one million lines of C code. LaSSIE is very powerful but creating a knowledge structure of any significant size is extremely labor intensive. Another disadvantage is rigidity. They usually support a very narrow application domain.

Another frame-based system is described by Allen and Lee [1]. Their system is called Bauhaus and supports the description and retrieval of reusable parts and their composition. Manual and automatic indexing methods are used, the latter done by parsing Ada packages. Wood and Sommerville [98] have also used frames for indexing reusable components. In their system frames slots are nominals, actions, and modifiers from some domain. Nominals are simple concepts, actions are simple verbs, and modifiers specify attributes of nominals and actions. Wood and Sommerville have used their system to represent UNIX tools.

2.2 Similarity and Effort Estimation

It has been argued that effort estimation by humans has long been a problem in software engineering. Woodfield, Embley, and Scott [99] performed an experiment where programmers were asked to evaluate the degree of reusability of software components. They concluded, among other things, that programmers cannot assess the worth of reusing a candidate component to satisfy the implementation requirements, and that in their evaluations programmers are influenced by some unimportant features (size of the candidate, percent of additional operations required) and are not influenced by some important ones (percent of operators to be modified, estimates of effort based on software science and lines of code).

From the perspective of a software production staff, profitability depends largely on assessing the applicability of reusable components, and incorporating them into the software system being developed, all of which requires methods for estimating the amount of effort to reuse these components. Part of the solution to this problem of effort estimation is to automate this process via software tools, but this is easier said than done. Estimating the amount of effort required to transform a candidate reuse component into a required component is, at best, difficult because it involves analysis of the component's structure, semantics, and relation to its environment.

2.2.1 Similarity and Software Component Models

One method to estimate transformation efforts used by several software reuse systems is based on computing the degree of *similarity* among object representations. The basic assumption is that the transformation effort from a candidate to a target component is proportional to the degree of similarity of their representations. Examples of such systems include Prieto-Díaz's faceted classification [75], the GURU system by Maarek *et al.* [63], and the AIRS system described in this thesis.

Prieto-Díaz's faceted classification system makes use of *conceptual distance graphs* and fuzzy logic to estimate the degree of similarity among component descriptions. The AIRS system, on the other hand, uses the concepts of *closeness* and *subsumption* relations for this same purpose. Both these relations define a measure of similarity called *similarity distance*.

In the case of the GURU system, a concept of *dissimilarity index* is used to cluster similar components based on lexicographic order of extracted keywords. A dissimilarity index δ over a set of objects Ω is defined [63] as a function $\Omega \times \Omega \rightarrow R_+$ that satisfies the following two properties:

$$\begin{aligned} \forall x \in \Omega, \delta(x,x) &= 0 \\ \forall (x,x') \in \Omega^2, \delta(x,x') &= \delta(x',x) \end{aligned}$$

That is, a dissimilarity index is reflexive and symmetric, but not necessarily transitive. This index has been used to define several similarity measures between documents [95]. The particular one used in the GURU system is defined as $|X \cap Y|$, where X and Y are the profiles[1] of two documents, and represents the number indices common to both documents. It is computed as follows:

[1]The profile of a document is defined by a set $\{(i, \rho)\}$, where ρ is the value of the index i.

$$\delta(X,Y) = \sum_{i \in p(X) \cap p(Y)} (\rho_X(i) \times \rho_Y(i))$$

where $\rho_X(i)$ is the ρ value of the index i in the profile X, and similarly for Y.

In the area of Artificial Intelligence, knowledge is very commonly represented as graphs or semantic nets. Depending on the nature of the problem, the edges can represent physical links, time duration, or abstract relationships. There are various ways of assessing the similarity of concepts depending on the representation adopted for knowledge. A comprehensive review of these different approaches was done by Rada *et al.* [79]. One particular example can be found in the theory of spreading activation [28] where one of the assumptions is that the semantic network is organized along the lines of semantic similarity. The more properties two concepts share in common, the more links (i.e., "is-a" relations) there are between the concepts and the more closely related they are. Semantic distance is measured as the minimal path length between two nodes in the net. These concepts have been applied to systems based on semantic nets such as KL-ONE [21].

2.2.2 Similarity and Software Metrics

All previously described systems were designed explicitly with the purpose of computing the degree of similarity between pairs of component descriptions within the same system. That is, similarity is computed based on the descriptions (i.e., models) of the components, but not using the components themselves.

Several types of software metrics have been defined with the purpose of analyzing and comparing software components, and to assess their degree of reusability [29, 43, 12, 24]. Software metrics attempt to quantify a particular characteristic of a software component by producing normalized numeric values which can later be used to rank the component against standardized tables or models. Although most software metrics have not been designed explicitly with the purpose of estimating similarity and effort transformations, they can easily be used as a basis to construct models for this purpose.

One example of a set of software metrics designed to identify and extract reusable components is described by Caldiera and Basili [24]. They use four different metrics for estimating the levels of usefulness, reusability, quality, and cost of a component: *volume*, *regularity*, *reuse frequency*, and *cyclomatic complexity*.

- **Volume.** The volume of the component affects both its reuse cost and quality. If a component is too small, the combined costs of extraction, retrieval, and integration exceed its intrinsic value. If it is too large, the component is more error prone and has lower quality. A component's volume can be measured using the Halstead Software Science Indicators [29], which are based on the way a program uses the programming language.

The Halstead volume is based on the concepts of *operators* and *operands*: an operator is an active program element (e.g., arithmetic operators or functions), and an operand represents a passive element (e.g., constants or variables). The Halstead volume is computed as $V = (N_1+N_2)\log_2(\eta_1+\eta_2)$, where N_1 and N_2 are the respective total count of all usage of operators and operands, while η_1 and η_2 are the respective total count of operators and operands used in the program.

- **Regularity.** The regularity of a component measures the readability and the non redundancy of a component's implementation. It can also be estimated by using the Halstead Indicators as $r = \hat{N}/N$, where $N = N_1 + N_2$ is the actual length of the document, and $\hat{N} = \eta_1 \log_2(\eta_1) + \eta_2 \log_2(\eta_2)$ represents the estimated length.

- **Reuse Frequency.** The reuse-specific frequency is an indirect indicator of the functional usefulness of a component, as long as the application domain uses some naming conventions to ensure that different names do not represent the same functionality. The basic idea to estimate reuse frequency is to compare the number of static calls addressed to a component with the number of calls addressed to a class of components known to be reusable. Let C be a component, $n(X)$ be the number of calls addressed to a component X, and M be the number of components $S_1, \ldots, S_M$ defined in the standard environment. The reuse frequency of C is computed as follows:

$$v_\sigma(C) = \frac{n(C)}{\frac{1}{M}\sum_{i=0}^{M} n(S_i)}$$

- **Cyclomatic Complexity.** The complexity of a component's control flow affects both its reuse cost and quality. Reuse of a component with very low complexity may not repay the cost, whereas high component complexity may indicate poor quality—low readability, poor testability, and a high possibility of errors. Complexity can be estimated using the McCabe measure [29], defined as the cyclomatic number of the control-flow graph of the program $v(G) = e - n + 2$, where e is the number of edges in the graph G, and n is the number of nodes.

Other examples of software metrics have been proposed. For example, Gannon *et al.* [43] used the "package visibility" metric to study usage of Ada programs. Similarly, Basili *et al.* [12] used the number of "data bindings" to represent the degree of connectivity among component pairs. A data binding is defined by Hutchens and Basili [48] as a triplet (p, x, q), where x is a global variable and p and q are program components, and where p assigns x and q references it.

2.3 AIRS: An AI-based Reuse System

The origins of the Extensible Description Formalism (EDF) are found in a system called AIRS (AI-base Reuse System) [73]. This system was designed to reuse Ada packages, and allow a software developer to browse a software library in search of components that best approximate some design specifications. AIRS relies heavily on several AI related data structures, techniques, and algorithms. The internal representation of the software library is constructed using a frame system with multiple inheritance [26], while the procedures used to find reuse candidates are based on A*-like search algorithms [72].

AIRS is essentially a combination between the faceted index and frame-based approaches. The domain information inherent in the facets is used largely to reduce the rigidity and the laborious creation of a semantic structure. A hierarchical frame system is used to maintain information on which of the objects in the reuse libraries have which features, how these objects are grouped, and how the features are related. Procedural attachment in the frame system is used to make the AIRS browsing system more efficient. In addition the features of the frame system are used to facilitate the integration of new components into the AIRS system, allowing a programmer to bootstrap its knowledge structures from a basic set of existing components.

2.3.1 The AIRS Classification Model

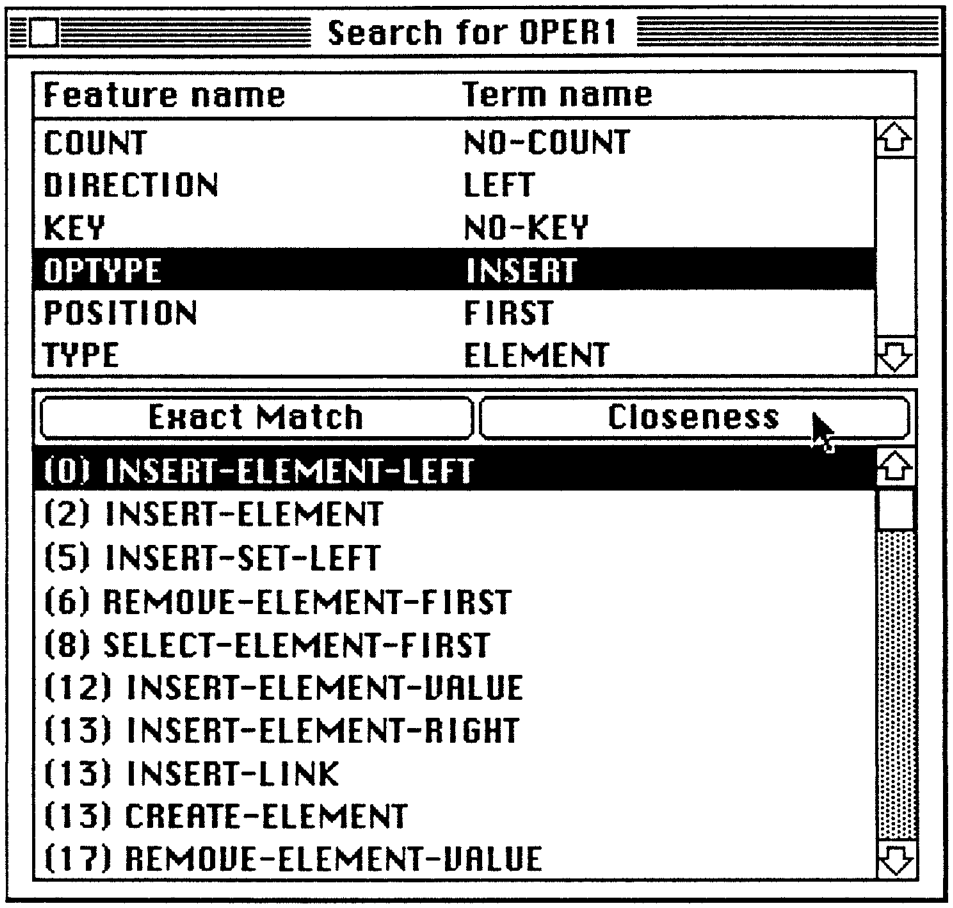
The AIRS model for classifying and computing similarity between software components is based on a two-level hierarchy. The base elements in the hierarchy are called *components*. The top elements are called *packages* and represent groups of related components. Both components and packages are described in terms of *features*. A feature is defined as a set of related values called *terms*. For example, a feature named `source-language` can be defined as the set of terms {`Pascal, Fortran, Ada, C`}. A component or package is modeled by a set of (f, t) pairs, where f is a feature and t is a term of f. A *package* represents a collection of components which are tightly coupled, that is, each component in the collection is defined to be used in conjunction with the others. This set of components are called *members* of the package. For example, abstract data types such as stacks and list can be represented in AIRS as packages.

AIRS uses two different methods for computing similarity between components and packages: the *closeness relation* and the *subsumption relation*. The closeness relation is based on the idea that two components are similar if one can be obtained by adapting a small number of parts of the other. The degree of closeness of two components (or packages) is computed by summing up the *closeness distance* of their corresponding feature terms. The closeness distance between two terms is quantified by a non-negative magnitude inversely proportional to their similarity. The subsumption relation is based on the idea that two components are similar if one is a subpart (e.g., subfunction) of the other. Components that have this relation are arranged in a directed acyclic weighted graph, and their degree of similarity is measured as the weight of the shortest path in this graph.

2.3.2 Applications of AIRS

The AIRS system has been used to construct two different software libraries. The first is the EVB GRACE library (part I) developed by EVB corporation, which contains a collection of Ada packages that implement data structures such as stacks and undirected graphs. The second is the CTC CCIS software library developed at Contel Technology Center, which contains a collection of C modules for implementing the basic functionalities of Command, Control, and Information Systems.

A highly interactive prototype of the AIRS system was implemented using Common Lisp in a Macintosh computer. This prototype supports an iterative refinement retrieval process for finding candidate reuse components and packages. This semi-automatic process takes the initial specifications of target components and/or packages and refines them until the reuser is satisfied with the candidate reuse components found using the similarity-based retrieval mechanism provided by AIRS. Figure 2.1 shows one of the panels provided by the prototype system for the purpose of retrieving candidate reuse components. The names shown in this figure form part of the AIRS GRACE reuse library.



Feature name	Term name
COUNT	NO-COUNT
DIRECTION	LEFT
KEY	NO-KEY
OPTYPE	INSERT
POSITION	FIRST
TYPE	ELEMENT

```
(0) INSERT-ELEMENT-LEFT
(2) INSERT-ELEMENT
(5) INSERT-SET-LEFT
(6) REMOVE-ELEMENT-FIRST
(8) SELECT-ELEMENT-FIRST
(12) INSERT-ELEMENT-VALUE
(13) INSERT-ELEMENT-RIGHT
(13) INSERT-LINK
(13) CREATE-ELEMENT
(17) REMOVE-ELEMENT-VALUE
```

Figure 2.1: Search panel of the AIRS system

2.3.3 Shortcomings

Although AIRS provides the basis for a useful software reuse library system, it has significant limitations and, therefore, can only be regarded as a first step towards a more complete system.

- **Fixed Set of Features.** One important restriction of AIRS is that all components and packages in a library must be defined in terms of a fixed set of features. What is more, an object in a particular class (i.e., components or packages) must be described in terms of all the features of the class.
- **Fixed Class Hierarchy.** In addition to the limited number of classes imposed by AIRS, objects of these classes can only be related in a fixed hierarchy. This arrangement does not allow a library designer to incorporate additional levels to the hierarchy—for example, to add a third level describing *systems* of related packages.
- **Fixed Comparators.** Similarity among object descriptions is quantified using either the *closeness* or the *subsumption* relation. The system does not provide any facilities for incorporating new methods for computing similarity.
- **No Verification of Consistency.** The semantic interdependencies among feature values defined by the underlying classification model must be maintained in order to have consistent object descriptions in the library. Yet, AIRS considers all features to be independent of each other, and provides no mechanism to express and ensure relations among their values.
- **Restricted Feature Types.** AIRS characterizes objects in terms of features and terms, in other words, the only *type* of features supported by AIRS are *enumerations*. This restriction limits the kinds of objects that can be described. For instance, basic properties such as "number of lines of code" and "source code file name" of an operation are very difficult (if not impossible) to express with AIRS.

2.4 Summary

The EDF system is essentially a combination between the faceted index and semantic network approaches. On one hand, objects are described in terms of a controlled vocabulary defined by the names of attributes and values. On the other hand, objects can be arranged in semantic networks based on relations such as "isa-a", "component-of", and "contains".

One important aspect of EDF is the ability to reason heuristically about the similarities between desired components and components residing in the existing software library. EDF is similar to the case-based reasoning approach currently being explored as a solution to many AI problems. In this approach, memory of previous solutions is used as heuristic knowledge to guide the processing of a new problem. An important aspect of case-based reasoning is the ability to find the information in memory most similar to the current situation. To retrieve relevant memories, these systems typically use a "domain theory"—a knowledge-base of information about the particular situations the system is expected to handle.

Chapter 3

The EDF Classification System

The Extensible Description Formalism (EDF)[1] is a general system for creating, using, and maintaining libraries of object descriptions with the purpose of improving reusability in software and non-software organizations. EDF is based on a generalization of the faceted index approach for creating Reuse Library Systems such as the AI-based Reuse System (AIRS) described in Section 2.3, and Prieto-Díaz's Faceted Library System (FLS) [78]. EDF overcomes the limitations of these systems by extending their representation model and incorporating a retrieval mechanism based on asymmetric similarity distances.

Reusable objects (e.g., software components) are described in terms of a set of attributes and types which define a classification taxonomy. Unlike other Reuse Library Systems on which objects must be defined in terms of all the attributes of the taxonomy, EDF can have different classes of objects in a library, each defined in terms of a particular set of attributes. One important feature of EDF is that object descriptions can themselves be used as attribute values, therefore allowing the definition of hierarchical classifications in which values are defined in more primitive terms. This feature, plus the fact that attributes can be assigned *sets* of values, can also be used to define semantic relations between different types of objects (e.g., "package-of" or "components-of").

In addition to the extensions to the faceted representation model, EDF incorporates a similarity-based retrieval mechanism that helps find suitable reuse candidate components in a library. Similarity is quantified by a non-negative magnitude called *similarity distance*, which is used as an estimator of the amount of effort required to transform one object into another. Distances between objects are computed using the values of the attributes of the their EDF descriptors. The basic assumption is that transformation effort can be estimated based on the EDF descriptions of the objects. One particular characteristic of this model is that distances between two object descriptions, A and B, are not symmetric, because the effort required to transform A into B is not necessarily the same as the one required to transform B into A.

Most Reuse Library Systems do not define a formal language for representing

[1]The initial version of EDF was designed by Pablo A. Straub and the author [90]. Later, the author extended and implemented EDF, and applied it to specific software domains.

objects and specifying the criteria for similarity. In the case of EDF, all the concepts that form its representation and similarity models can be expressed using the EDF Specification Language, whose syntax and semantics have been formally defined. In addition, a prototype application system which implements this language has been constructed and used to build several reuse library systems (see chapters 4 and 5). This prototype system is described in Appendix A.

The remaining of this chapter presents a detailed definition of the EDF system. Section 3.1 introduces the concepts behind EDF's representation and similarity model by developing a sample software reuse library. These concepts are then formalized in Section 3.2. Finally, Section 3.3 defines the syntax and semantics of the EDF Specification Language.

3.1 Developing a Reuse Library

To create and organize an effective reuse library, an extensive domain analysis must be performed beforehand. This analysis must produce a classification scheme (including attributes and their types) as well as an approximate measure of similarity between objects.

This section develops a small software library to classify operations to manipulate data structures consisting of repeated elements (e.g., stacks, trees, hash tables). The purpose of this section is to introduce the concepts underlying EDF's representation and similarity models. For presentation purposes we start with a trivial library and enhance it as more features of EDF are introduced.

3.1.1 Creating a Taxonomy

Booch [20] classifies operations over a data structure in the following three classes, based on how the structure is accessed.

- **Constructors:** operations that alter the data structure.
- **Selectors:** operations that evaluate the data structure.
- **Iterators:** operations that visit all elements of the structure.

We can describe this simple classification scheme by defining an attribute called `functionality` as follows:

```
  attribute functionality : {construct,select,iterate};
```

Another attribute for classification of operations is execution time as a function of the size of the data structure.

```
  attribute
    timing : {constant,log,linear,loglinear,quadratic,slow};
```

Attributes `functionality` and `timing` define a simple classification scheme that can be used to describe four operations for stack manipulation. Each of these descriptions is called an *instance*.

```
push     = [functionality=construct & timing=constant];
pop      = [functionality=construct & timing=constant];
top      = [functionality=select    & timing=constant];
newstack = [functionality=construct & timing=constant];
```

This section has introduced two basic concepts of the EDF language: attributes and instances. The type associated with both attributes, `functionality` and `timing`, is an enumeration of terms. Each instance defines the attribute values of a particular data structure operation.

3.1.2 Extending a Taxonomy

The characterization of the functionality of operations presented above is too coarse. In fact, the descriptions of `push`, `pop`, and `newstack` are identical. This section refines this characterization by extending the classification scheme. There are at least three approaches to do this.

- Add or replace terms in the type of an attribute.
- Add more attributes.
- Describe attribute values in terms of more primitive attributes.

The first two approaches are common practice while designing a taxonomy and are the only alternatives a library designer has with other classification systems such as AIRS or Prieto-Díaz's system. The third approach is unique to EDF, and allows the construction of hierarchical classification systems. These approaches are explained in turn.

Adding Values to a Type

In this approach, the classification scheme is refined by including additional values to the type of an attribute. In particular, we add new terms to the `functionality` attribute. In the context of data structures consisting of repeated elements, the `constructor` term will be replaced by three new terms `create`, `insert`, and `remove`. With this new definition we can now tell `push` from `pop` and tell those from `newstack`. The updated definitions are as follows:

```
attribute
  functionality : {create,insert,remove,select,iterate};
```

```
attribute
  timing : {constant,log,linear,loglinear,quadratic,slow};

push     = [functionality=insert & timing=constant];
pop      = [functionality=remove & timing=constant];
top      = [functionality=select & timing=constant];
newstack = [functionality=create & timing=constant];
```

The drawback of this approach is that instance definitions had to be manually modified (e.g., changing construct by the corresponding new term in each instance). Moreover these extensions create flat taxonomies with few attributes and many terms, instead of hierarchies.

Adding Attributes

In EDF it is possible to define a new attribute and then use it to refine the classification of selected instances. Unlike other faceted classification systems, this new attribute does not have to be used in all instances. Hence, the addition of attributes requires modifying only those instances for which the new attribute is meaningful and important.

For example, we extend the taxonomy by adding a new attribute called exception. This attribute is used to describe those operations that can signal a fatal exception such as a stack overflow or underflow. The following definitions are added or modified in our library:

```
attribute exception : {underflow,overflow};

push = [functionality=insert & timing=constant &
        exception=overflow];
pop  = [functionality=remove & timing=constant &
        exception=underflow];
```

Only those operations that can generate an exception (i.e., push and pop) have been described using the attribute exception. The remaining instances in the library (i.e., top and newstack) were not modified and, therefore, have no defined value for the attribute exception.

It can be argued that the attribute exception could have been defined with an additional term called noException to describe those operations that do not generate exceptions. In this solution, all instances would have been defined using the same set of attributes and therefore a system like AIRS could still be used to model our taxonomy. Although this argument is valid in the current example, the fact that EDF can handle descriptions with different sets of attributes is particularly important in the case of libraries containing objects of different classes such as "projects", "systems", "packages", and "operations". The attributes of these sample classes are most probably disjunct, but they can all be classified in a single EDF library.

Describing Values of an Attribute

EDF provides a new approach to extend a classification scheme: describe all terms of an attribute using more primitive attributes. This process is illustrated by refining again the `functionality` attribute.

Within the domain of data structures consisting of repeated elements, the functionality is described in terms of three new attributes: `access` (whether the data structure is written or only read), `target` (which elements are affected), and `newsize` (how the number of elements varies).

```
attribute access  : {write,read};
attribute target  : {leftmost,rightmost,keyed,any,all,none};
attribute newsize : {increase,decrease,reset,same};
```

These new attributes are used to define each of the terms that belong to the attribute `functionality`.

```
create  = [in constructors & newsize=reset & target=none];
insert  = [in constructors & newsize=increase];
remove  = [in constructors & newsize=decrease];
select  = [in selectors];
iterate = [in iterators];
```

Where `constructors`, `selectors`, and `iterators` each define a *class* of instances. The class mechanism is used both as an abstraction mechanism and also as an abbreviation for expressions. These classes are defined as follows:

```
constructors = class(access=write);
selectors    = class(access=read & newsize=same);
iterators    = class(target=all);
```

The definition of the attribute `functionality` can now be changed, because its elements no longer belong to an enumeration type but to a class of instances, namely the class of instances defined in terms of one or more of the attributes `access`, `target`, and `newsize`.

```
attribute
  functionality : class(has access | has target | has newsize);
```

Since all former terms of the attribute `functionality` are defined, instances described using these values (e.g., `push`) do not need to be redefined. That is, this extension of the classification system does not affect the classification of objects already in the library.

This extended classification scheme allows us to define new categories of functionality. For example, we can define `modify` as a possible value of functionality, and also describe more specific iterators.

```
modify = [in modifiers];
passive-iterate = [in iterators & in selectors];
active-iterate  = [in iterators & in constructors];
modify-iterate  = [in iterators & in modifiers];
modifiers = class(access=write & newsize=same);
```

Where `modifiers` is the class of all operations that update elements in the data structure.

In summary, the process required to extend a classification scheme by redefining the terms of an attribute is as follows:

1. Select an attribute a whose terms are to be refined. Let T be the type of a. In the example, $a =$ `functionality` and $T = \{$`create, insert, remove, select, iterate`$\}$.

2. Perform a domain analysis on the domain of the terms of a. From this analysis, define a set A of new attributes that describe terms in T, and determine the type for each attribute in A. In the example, $A = \{$`access, target, newsize`$\}$ with their corresponding term enumerations.

3. Redefine attribute a. Possible values for a are not terms as before (type T is no longer part of the library), but instances that belong to a class defined using the attributes in A.

4. Define each former term $t \in T$ as an instance using the attributes in A, following the same procedure used to describe data structure operations.

5. If needed, other values for a can be described. This values can be specializations of former terms (e.g., passive-iterate) or they can represent new concepts (e.g., modify).

In principle, this process of refinement can be done indefinitely providing deep hierarchical taxonomies, but there is a point in which using this formalism is no longer useful (e.g., do not use EDF to describe detailed functionality, including pre- and post-conditions).

3.1.3 Creating Object Hierarchies

Reusable software usually consists of packages or modules, made from operations and other packages. We want to represent this modular structure, but we do not want to force any granularity of reuse. That is, we want to have a library consisting of packages and operations, assuming that both complete packages and isolated operations will be reused.

The following declarations define the kinds of reusable software components for a library of data structure packages. Because a package can have several subunits, the `subunits` attribute has a set type.

```
attribute subunits : set of components;
attribute parent   : packages;

components = class(in packages | in operations);
packages   = class(has subunits);
operations = class(has functionality | has timing);
```

Two other attributes for packages are defined: `maxsize` (whether there are limits in the number of elements of the structure) and `control` (whether concurrent access is supported) [20].

```
attribute maxsize : {bounded,limited,unbounded};
attribute control : {sequential,concurrent};
```

With these declarations, a stack package comprising the operations already described can be defined using one extra attribute (`parent`). The implementation has no preset bound on size and does not provide support for concurrency.

```
stack    = [subunits=set(parent=stack) & maxsize=unbounded &
             control=sequential];
push     = [parent=stack & functionality=insert &
             timing=constant & exception=overflow];
pop      = [parent=stack & functionality=remove &
             timing=constant & exception=underflow];
top      = [parent=stack & function=select & timing=constant];
newstack = [parent=stack & function=create & timing=constant];
```

Where the construct "`set(parent=stack)`" denotes the set of all instances defined in the library for which the attribute `parent` is equal to `stack`, in other words, the set {`pop`, `push`, `top`, `newstack`}.

3.1.4 Dependencies among Attributes

All classification schemes assume that certain semantic relations between attribute values are being maintained. For this purpose, EDF provides a mechanism that uses *assertions* to define semantic constrains between attribute values.

For example, consider the case of attributes describing the functionality of an operation. If the data structure is not written then there is no size change, and if the structure is reset then there is no specific target. These two relations can be expressed as follows:

```
assertion access=read => newsize=same;
assertion newsize=reset => target=none;
```

In addition, the attributes `maxsize` and `control` are only relevant for packages, and all units that declare a package as their parent must indeed be subunits of the package.

```
assertion has maxsize | has control => in packages;
assertion in packages => subunits=set(parent=self);
```

Where the keyword `self` denotes the instance being analyzed for compliance with the assertion.

3.1.5 Defining Synonyms

One of the difficulties of describing operations given our current taxonomy is remembering the precise terms used in the library. Besides, certain concepts can be given or referenced by more than one name. The introduction of synonyms for terms has been suggested as a partial solution to this problem [78].

One could declare that the distance between two terms is zero, making them synonyms from the point of view of queries based on similarity. However, queries based on exact matches will considered them different. In EDF is possible to declare an identifier i_1 to be a synonym of an identifier i_2 by simply declaring $i_1 = i_2$. For example:

```
update = write;
preserve = read;
```

These definitions introduce the synonyms `update` and `preserve` for the terms `write` and `read` of attribute `access`, respectively.

3.1.6 Queries and Comparing Objects

In order to find reusable software components in the library of packages and operations, it is necessary to define the distance values associated with the terms of enumerations types. This allows EDF to compute distances not only between these terms, but between instances defined using these terms.

Distances between terms are defined with a distance clause. For example, attributes access and newsize and their distance clauses are given below. The distances shown here are just sample values. Chapter 5 presents a detailed description of the process of assigning distances.

```
attribute access :
   {write,read}
distance
   {write -> read:4,
    read  -> write:6};

attribute newsize :
  {increase,decrease,reset,same}
distance
  {increase -> decrease:5,same:7,
   decrease -> increase:5,reset:3,
   reset    -> same:10,
   same     -> reset:10};
```

By transitivity we can determine other distances not explicitly given. For example, the distance from increase to reset is $5 + 3 = 8$, and the distance from decrease to same is 12. Note that a bigger value for this distance (13) can be obtained going from decrease to reset to same, but EDF always uses the smallest value.

Basically, the distance between two instances is computed by adding the distances of their corresponding attribute values. For example, the distance from remove to select is 16, given by the distance from write to read (4) plus the distance from decrease to same (12).

```
remove = [access=write & newsize=decrease];
                    4    +        12          =  16
select = [access=read  & newsize=same];
```

Distances between instances are used by EDF to select reuse candidates from a library. This selection is performed using the query command. For example, the following query finds components that are similar to an operation that retrieves an arbitrary element from a data structure in at most logarithmic time.

```
query functionality=[in selectors & target=any] & timing=log
```

Consider another example. Find a data structure with three operations: one to initialize, one to insert an element, and one to traverse the structure without modifying it; concurrent control is not needed, but the structure must be able to handle an unbounded number of elements.

```
query maxsize=unbounded & control=sequential &
      subunits={[functionality=create],
                [functionality=insert],
                [functionality=passive-iterate]}
```

In this query only the functionality of the operations have been specified. Attribute `timing` is not defined, meaning that any value for timing is equally acceptable in the retrieved operations.

3.2 Foundations of EDF

3.2.1 Representation Model

To understand the representation principles of EDF, it is useful to consider descriptions of objects of a particular class as points in a multidimensional space, were each dimension is represented by an *attribute*. Attributes have a name and a list of possible values defined by their associated *type* (i.e., set of values). If a is an attribute name, and v belongs to the a's type, the assignment "$a = v$" represents the set of all objects whose attribute a is v.

Assignments can be combined in expressions to define other sets of objects. In particular, if A_1 and A_2 are two assignments, the expressions "A_1 & A_2" represents the intersection of the sets A_1 and A_2. Similarly, "$A_1 \mid A_2$" represents the union of these sets. In addition, the set of objects that have been defined in terms of a particular attribute a, independently of the value associated with a, is denoted by "`has` a". The set of objects defined by the "`has`" operator is a short form of the expression "$a = v_1 \mid a = v_2 \mid \cdots \mid a = v_n$" where the values v_i are the elements of the type of a.

A set of objects is called a *class* in EDF. Classes can be given a name and they are denoted as `class(`E`)` where E is an expression; i.e., unions and intersections of other sets of objects. If c is a class name, the set of objects it represents is denoted by "`in` c", and can be combined with other sets of objects in an expression.

An object description is called an *instance* in EDF. Instances can be given a name and they are denoted as `instance(`E`)` or `[`E`]` where E is an expression. Semantically, an instance must have only one set of attributes, therefore we say that `instance(`E`)` is well defined if and only if (1) E is not a contradiction (i.e., `class(`E`)`$\neq \emptyset$), and (2) E defines a mapping from attributes to values, that is, E can be simplified into a consistent conjunct of assignments.

Expressions can also be used to characterize particular sets of instances defined in an EDF library. We denote by $\texttt{set}(E)$ the set $\{i | i \in D \cap \texttt{class}(E)\}$, where D is the set of instances in the library. In other words, the **set** operator defines the set of instances in the library that belong to the class defined by E.

3.2.2 Similarity Model

The goal of any Reuse Library System is to facilitate the process of finding suitable objects for reuse. EDF supports two criteria for selecting candidate objects: by exact match and by similarity. For exact matches the construct $\texttt{set}(E)$ already described is used. Similarity-based queries are performed using the construct "**query** E", which denotes the list of instances in the library sorted by decreasing similarity to the target object define by E. That is, the first element of the list "**query** E" is the best reuse candidate for $[E]$, the following element the second best, and so on.

As mentioned earlier, similarity is quantified by a non-negative magnitude called *similarity distance*, which is used as an estimator of the amount of effort required to transform one object into another. Because of this, distances between two object descriptions, A and B, are not symmetric, because the effort required to transform A into B is not necessarily the same as the one required to transform B into A. For this reason, whenever a distance is computed, it is important to define which object is the *source* and which the *target*.

Let Π be an object class defined by the set of attributes $\overline{\Pi} = \{A_1, \ldots, A_n\}$, and S and T be two instances in this class. Also, let $\overline{S} \subseteq \overline{\Pi}$ be the actual set of attributes used to define S, and similarly for $\overline{T}$. The distance from S to T is denoted by $D(S,T)$ and is computed as follows:

$$D(S,T) \;=\; \sum_{A \in \overline{S} \cap \overline{T}} k_A T_A(S.A, T.A) + \sum_{A \in \overline{S} - \overline{T}} k_A R_A(S.A) + \sum_{A \in \overline{T} - \overline{S}} k_A C_A(T.A)$$

where $I.A$ denotes the value of an attribute A of an instance I. The set $\overline{S} \cap \overline{T}$ represents the attributes shared by S and T, while $\overline{S} - \overline{T}$ is the set of attributes found in S but not in T, and similarly for $\overline{T} - \overline{S}$. These three sets are disjoint. In addition, each constant k_A is called the *relevance factor* of attribute A. Their values fall in the range 0 to 1, and must satisfy the relation $\sum_{A \in \overline{\Pi}} k_A = 1$. Functions T_A, R_A, and C_A are called *comparators*, and are explained later in this section.

The expression for distance $D(S,T)$ is based on the assumption that the overall transformation effort from S to T can be computed using a linear combination of the differences between their respective attributes. In other words, attributes are considered independent of each other when computing similarity. This is a strong assumption that limits the types of domains that can be handled by EDF's similarity model.

Comparators

As explained earlier, each attribute has three associated functions T_A, R_A, and C_A called *comparators*. T_A is the *transformation comparator* and is used to quantify the amount of effort required to transform one value of the attribute into another. R_A is the *removal comparator* and is used to estimate the amount of effort required to eliminate a source attribute value not required in the target specification. Finally, C_A is the *construction comparator* and estimates the amount of effort required to supply a target attribute value not specified in the source specification.

The set of all attribute comparators plus their associated relevance factors define a specific similarity model for a reuse library. These functions and values must be specified using a process called *domain analysis* [76] which, among other things, defines the criteria for similarity for objects in a particular domain. Nonetheless, EDF provides default comparators for each type of attribute. These default comparators can be used as a starting point from which to refine the similarity model of a library. This refinement is normally done by assigning attributes non-default comparators using "foreign" functions specified in some conventional programming language such as ANSI C. Appendix A explains "foreign" comparators in detail.

Numbers and Strings. Numbers and strings are the simplest type of value an attribute can have. In the case of numbers, the transformation distance from a source value N_S to a target value N_T is given by $|N_S - N_T|$, that is, by the absolute magnitude of their difference. The construction distance for a value N is N itself, while the removal distance is 0. The rational is that numeric values are normally used to measure things like "the number of lines of code of a component" which normally increase in value as the amount of work involved in the component increases. In the case of strings, the transformation comparator is defined as zero (0) if the text of the strings are identical, and infinity otherwise. Construction and removal comparators are both defined as infinity.

Enumerations. Distances between terms of an enumeration are defined by a weighted directed graph, where each node represents a term. The weight w associated to an arc connecting a node t_1 to a node t_2 is a non-negative magnitude that represents their associated transformation distance. The graph also includes an additional node called the *null term* and is denoted by Θ. The weight of an arc from Θ to a term t represents t's construction distance, while that of an arc from a term t to Θ represents t's removal distance.

Some pairs of nodes in the graph may not be connected by an arc, meaning there is no known method to directly obtain one from the other. Yet, to compare instances we need to estimate distances between all possible pairs of terms. We define the distance from term t_1 to term t_2 as the weight of the shortest path from t_1 to t_2 in the graph. If no such path exists, the distance is set to be infinity. If t_1 and t_2 are the same, the distance is zero.

Consider the enumeration {**assembler,pascal,common_lisp**} and its associated graph shown in Figure 3.1. According to the graph[2] the distance from **assembler** to **pascal** is 10, and the distance from **pascal** to **common_lisp** is 5, corresponding to the intuition that transforming an assembler source file to pascal is more difficult than recoding pascal to common_lisp. The creation distance for **assembler** is 30, given by the arc connecting Θ to **assembler**. The creation distance for **pascal** and **common_lisp** is 40 and 45 respectively, given by the weight of the shortest path from Θ. Similarly, the distance from **pascal** to **assembler** is 30, since the path **pascal** $\rightarrow \Theta \rightarrow$ **assembler** yields the smallest distance from pascal to assembler.

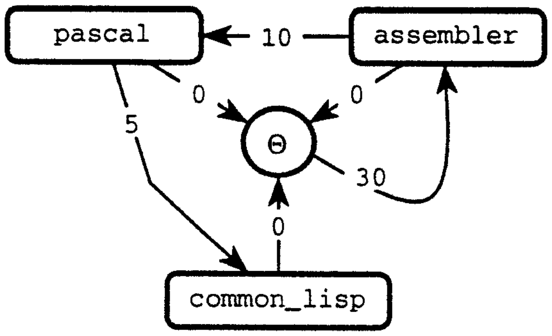


Figure 3.1: Distance graph for an enumeration type

For example, the definition of an attribute called **language** based on this enumeration and graph could be defined using the EDF specification language as follows:

```
attribute language :
  {assembler,pascal,common_lisp}
distance
  {
  ->assembler:30,assembler->:0,assembler->pascal:10,
  pascal->:0,pascal->common_lisp:5,common_lisp->:0
  };
```

Sets of values. The default transformation distance from a source set S_S to a target set S_T is computed by selecting for each value v_t in S_T the value v_s in S_S that is most similar to v_t (i.e., with the smallest distance to v_t). The overall transformation distance from the source set to the target set is then computed by adding the distances associated with each of these pairs of values.

[2]Arc weights were selected from a subjective scale in the range 0 (trivial transformation) to 30 (hard transformation)

More precisely, let A and B be two attributes such that the values of A are defined as sets of the values of B. If the type of S_S and S_T is that of attribute A, then the transformation comparator of A is defined as follows:

$$T_A(S_S, S_T) \;=\; \sum_{v_t \in S_T} \min_{v_s \in S_S} T_B(v_s, v_t)$$

where T_B is the transformation comparator of attribute B. In addition, the construction and removal comparators of attribute A are defined as follows:

$$\begin{aligned} C_A(S) &= \sum_{v \in S} C_B(v) \\ R_A(S) &= \sum_{v \in S} R_B(v) \end{aligned}$$

where R_B and C_B are the removal and construction comparators of attribute B, respectively. That is, the amount of effort required to construct a a set S is the sum of the construction efforts required to construct each of its elements. Similarly for the removal effort.

3.3 EDF Specification Language

3.3.1 Syntax of the language

This section presents a formal definition of the syntax of the EDF language. Syntax is presented in a variation of BNF using the following conventions: keywords and symbols occurring literally are written in `typewriter` typeface; nonterminals are written in *italics*; *type-name*, *attribute-name*, *instance-name*, *term*, and *class-name* all denote identifiers; *symbol*,... means one or more occurrences of *symbol*, separated by commas; and **keyword**$_{\text{opt}}$ means that the keyword may or may not occur, without affecting the semantics.

Declarations

An EDF library consists of a sequence of declarations. Each declaration either defines a name (of a type, an attribute, an instance, or a class) or describes an assertion that must be true of all instances in the library.

```
library ::=
        declaration

declaration ::=
        type-declaration
      | attribute-declaration
      | instance-declaration
      | class-declaration
      | assertion
```

Attributes and Types

Software components and other objects are described in terms of their attributes. We can think of attributes as fields of a record describing the object. The declaration of an attribute specifies the type of the values for the attribute. EDF supports the following types: `number`, `string`, term enumerations, object classes, and homogeneous sets of the above.

attribute-declaration ::=

  `attribute` attribute-name : type ;

type-declaration ::=

  `type` type-name = type ;

type ::=

  simple-type distance-clause
  | `set` distance-clause `of` type

simple-type ::=

  `number`
  | `string`
  | { term ,... }
  | class
  | type-name

where classes are defined on page 44 and the distance clauses are described below.

distance-clause ::=

  `distance`$_{opt}$
  | `no distance`
  | `distance` { triplet ,... }
  | `distance *` { triplet ,... }
  | `distance` { ctriplet ,... }

triplet ::=

  term$_{opt}$ `->` term$_{opt}$: number-literal

ctriplet ::=

  src-name$_{opt}$ `->` dst-name$_{opt}$: { C-code }

The keyword `distance` by itself is optional and assigns default distance functions. The case "`no distance`" indicates that the distance between values of the associated type is always zero.

In the third and fourth forms of the distance clause, the triplet t_1 `->` t_2 : n means that the distance from term t_1 to term t_2 is n. If t_1 is omitted the unspecified value is assumed (i.e., n is creation distance of t_2). If both t_1 and the arrow are omitted, the previous t_1 is assumed. If the keyword `distance` is followed by the character "*", then the distances between terms not mentioned in a triplet will be set to infinity. If "*" is not specified, distances between all terms will be adjusted by computing the shortest path between them.

The last form of the distance clause is a hook to distance functions programmed in a conventional programming language such a ANSI C. If both "src-name" and "dst-name" are specified, the triplet defines a transformation comparator. If "src-name" is omitted, the triplet defines a construction comparator. If "dst-name" is omitted, the triplet defines a removal comparator. These functions have read access to the library. This clause can be used with any type. For further details, refer to Appendix A.

Expressions

Expressions are formed from attribute assignments, the unary operators `has` and `in`, and the binary operators `&` (intersection) and `|` (union).

```
expression ::=
        attribute-name = value
      | has attribute-name
      | in class-name
      | expression & expression
      | expression | expression
      | ( expression )
```

The expression "*attribute-name* = *value*" means that the value of *attribute-name* for the instance being defined is *value*. The expression "`in` *class*" means that the instance defined belongs to *class*; it is similar to a macro-expansion of the expression that defines the class. The expression "`has` *attribute-name*" denotes the condition that the instance being defined has some value for *attribute-name*.

Values

Values are used in assignment expressions. Values are either simple values or set values. A simple value is either a literal (number or string), a term, an instance, or the value of an attribute of an instance. Set values must denote homogeneous sets; they are described either by extension or by intention, using the `set` construct. Only sets of instances can be described by intention.

```
value ::=
        simple-value
      | { simple-value ,... }
      | set ( expression )
      | set ( instance-name | expression )
```

```
simple-value ::=
        number-literal
      | string-literal
      | term
      | instance
      | instance . attribute-name
      | self
      | self . attribute-name
```

The construct `set(`*E*`)` represents the set of all instances in the library that satisfy the expression (i.e., that belong to `class(`*E*`)`). If the optional *instance-name* is used, then the name is bound within *E* to each instance in the library.

The dot notation "*instance . attribute-name*" is used to refer to the value of the attribute *attribute-name* of an instance. This notation is similar to that used in other languages to access record fields.

The keyword `self` is a reference to the instance defined by the expression in which the value is used. Within an `instance` construct, `self` is bound to the instance defined. Within an assertion, `self` is bound to every instance in the library in turn. Within nested `instance` constructs `self` is bound to the innermost instance.

Classes

A class is defined by giving the corresponding expression; the class denotes the set of all objects for which the expression holds. Classes are used to abstract properties of instances and also as abbreviations for the corresponding expressions. Classes are also used as types of attributes whose values are instances.

```
class-declaration ::=
        class-name = class ;

class ::=
        class ( expression )
      | class-name
```

Instances

Instances are defined in terms of an expression. As explained in Section 3.2.1, an instance defined by an expression *E* is a representative of the class of instances defined by "`class(`*E*`)`".

```
instance-declaration ::=
        instance-name = instance ;
```

instance ::=

```
    instance ( expression )
  | [ expression ]
```

An instance may not exist either because the class is empty (i.e., the expression is a contradiction) or because the class is not specific enough (i.e., it defines more than one valid set of attributes). A sketch of a possible simplification and verification algorithm is as follows.

1. Expand all "in" propositions with the expressions of the corresponding classes.
2. Transform the expression into disjunctive normal form, as follows:
   (a) Restructure the expression using associativity laws so that no disjunction occurs within a conjunction.
   (b) Represent each conjunct as a set of assignments and has propositions.
   (c) Represent the expression as a set of these conjuncts.
3. For each conjunction do the following:
   (a) Delete redundant assignments.
   (b) If there are still two assignments to the same attribute, or if there are unsatisfied has propositions, delete the conjunction.
   (c) Else, delete has propositions (not needed anymore).
4. Delete conjunctions that imply another conjunction.
5. If there are no conjunctions left, fail (E is a contradiction).
6. If there are more than one conjunction left, fail (E is not specific enough).

Assertions

An assertion specifies a semantic constraint that must be true of all instances in the library. Expressions are used to represent dependencies between attributes, to constrain data types and classes, and to enforce correct typing.

assertion ::=

```
    assertion expression => expression ;
```

The meaning of "assertion E_1 => E_2" is similar to set(E_1) $\subseteq$ set(E_2). This definition does not capture subtleties with respect to the binding of self; a precise definition is given on page 49. EDF signals false assertions.

Queries and distance computations

Queries are used to examine an EDF library; they are not part of the library itself. A query command computes a list of instances in the library sorted by decreasing similarity (increasing distance) to the implicit target instance define by an expression. The syntax of queries is:

query ::=
    query expression
  | **query** expression : identifier

If specified, *identifier* must be the name of an attribute or a type, and distances are computed using the distance functions associated with the type or the attribute. If *identifier* is not specified, distances are computed using the default distance functions provided by EDF.

The **distance** command is used to compute similarity distances between a pair of values. This command is useful for verifying the definition of distance functions and the results they produce.

distance ::=
    distance source-value$_{\text{opt}}$ **->** target-value$_{\text{opt}}$
  | **distance** source-value$_{\text{opt}}$ **->** target-value$_{\text{opt}}$: identifier

The *source-value* and *target-value* must be values of the same type (e.g., instance names). In case of terms, they must belong to the same enumeration. If both names are specified, the command computes their transformation distance. If only the source value is given, its destruction distance is computed. Finally, if only the target is specified, its construction distance is computed. The *identifier* has the same use as in the case of the **query** command.

3.3.2 Semantics of the language

We define the denotational semantics of EDF using Schmidt's notation [84]. We make minor extensions to the **cases** construct: patterns can include nested constructors, more than one pattern can match the tested value (the first one is chosen), and there is an optional **otherwise** pattern. In addition to λ-calculus we use standard mathematical notation (e.g., summations, set notation, predicate calculus).

To avoid considering many possible syntactical variations, the semantics is given for a subset of EDF in which all optional keywords are present and all types, classes and instances used are explicitly defined. We assume a preprocessor makes the transformation[3]. For instance, the library at the left column below is transformed into the equivalent library at the right column.

[3]In the implementation, the parser makes these transformations when it creates the abstract syntax tree.

```
attribute a : set of {i,o};    attribute a : a_type;
                               type a_type = set of a_base_type;
                               type a_base_type = {i,o};

attribute b : class(has a);    attribute b : b_type;
                               b_type = class(has a);

x = [b=[a={i}]];               x   = [b=x_b];
                               x_b = [a={i}];
```

The following subsections introduce and defined the semantic domains as well as the various valuation functions used in the semantic definition of the EDF language.

Domains

The meaning of an EDF library is given by an environment with the combined meaning of all declarations. We represent this meaning as a mapping from names to their denotations. The following naming convention is used: A, C, I, T, Z denote the set of names of attributes, classes, instances, terms and types, respectively and a, c, i, t, z denote names in their respective sets. Because EDF has only one name space, all these sets of names are pairwise disjoint.

$$Enviroment = Name \rightharpoonup Denotation$$

$$Name = A \cup C \cup I \cup T \cup Z$$

$$Denotation = Attribute + Class + Instance + Term + Type$$

The meaning of an attribute or term is simply the name of its type.

$$Attribute = Z$$

$$Term = Z$$

The meaning of an instance is a partial mapping from attributes to values, so no two assignments are allowed for the same attribute.

$$Instance = A \rightharpoonup Value$$

$$Value = I + \mathbb{R} + String + T + Setvalue$$

$$String = \text{set of all finite strings}$$

$$Setvalue = \mathbb{P}(I) + \mathbb{P}(\mathbb{R}) + \mathbb{P}(String) + \mathbb{P}(T)$$

The meaning of a class is an abstract syntax tree representation of the logical expression that defines the class.

$$Class = Self \rightarrow Expression$$

$$Self = Instance$$

$$Expression = And + Or + Has + Assign$$

$$And = Expression \times Expression$$

$$Or = Expression \times Expression$$

$$Has = A$$

$$Assign = A \times Value$$

The meaning of a type is a tuple $\langle v, t, c, r \rangle$, where v represents the set of values of the type, t is a function to compute the distance between two values of the type, c is a function to compute the construction distance of a value, and r is is a function to compute the removal distance of a value.

$$Type = Values \times TransformFunc \times ConstructFunc \times RemoveFunc$$

$$Values = C + Reals + Strings + Terms + Set$$

$$Reals = Unit$$

$$Strings = Unit$$

$$Terms = \mathbb{P}(T)$$

$$Set = Z$$

where the meaning of the distance comparators is as follows.

$$TransformFunc = Value \times Value \rightarrow \mathbb{R}$$

$$ConstructFunc = Value \rightarrow \mathbb{R}$$

$$RemoveFunc = Value \rightarrow \mathbb{R}$$

Declarations

The meaning of a declaration d is a function $\mathcal{D}[\![d]\!]$ that takes an environment and returns another environment with the updated declaration. The meaning of an EDF library is the least fixpoint of the $\mathcal{D}$ function applied to the library, that is

$\mathcal{M} : [\![Library]\!] \rightarrow Environment$

$\mathcal{M}[\![library]\!] = \text{fix}(\mathcal{D}[\![library]\!])$

The valuation function for declarations is

$\mathcal{D} : [\![Declaration]\!] \rightarrow Environment \rightarrow Environment$

$\mathcal{D}[\![\texttt{attribute}\ a\ :z]\!] = \lambda e.[a \mapsto inAttribute(z)]e$

$\mathcal{D}[\![c\ \texttt{=}\ \texttt{class(}E\texttt{)}]\!] = \lambda e.[c \mapsto inClass(\mathcal{E}[\![E]\!]e)]e$

$\mathcal{D}[\![i\ \texttt{=}\ \texttt{instance(}E\texttt{)}]\!] = \lambda e.[i \mapsto instantiate(\mathcal{E}[\![E]\!]e\ i)]e$

$\mathcal{D}[\![\texttt{type}\ z\ \texttt{=}\ type]\!] =$

$\quad \lambda e.\textbf{let}\ \langle v,t,c,r\rangle = \mathcal{T}[\![type]\!]e, e' = [z \mapsto inType\langle v,t,c,r\rangle]e\ \textbf{in}$

$\quad \textbf{cases}\ v$

$\quad isTerms(terms) \rightarrow \lambda id.(id \in terms \rightarrow z \,\square\, e'(id))$

$\quad \textbf{otherwise} \rightarrow e'$

$\mathcal{D}[\![\texttt{assertion}\ E_1\ \texttt{=>}\ E_2]\!] =$

$\quad \lambda e.\textbf{if}\ \forall i \in dom(e) \cap I : satisfies(e(i), \mathcal{E}[\![E_1]\!]e\ i)$

$\qquad\qquad \Rightarrow satisfies(e(i), \mathcal{E}[\![E_2]\!]e\ i)$

$\quad \textbf{then}\ e\ \textbf{else}\ \bot$

$\mathcal{D}[\![d_1\ d_2]\!] = \mathcal{D}[\![d_1]\!] \circ \mathcal{D}[\![d_2]\!]$

where $\mathcal{E}$ is the valuation function for expressions; *instantiate* creates an instance from a class; and $\mathcal{T}$ is the valuation function for types.

Expressions and Values

We want the meaning of an expression to be a syntax tree of the expression. To give adequate meaning to the syntax tree, it has to be evaluated in the context of an environment. Furthermore, because of the existence of the `self` value, the tree is evaluated in the context of a particular instance s. Hence, the valuation function for expressions $\mathcal{E}$ expects an expression and an environment and returns a class, that is, a function from an instance (self) to an expression. The valuation function for expressions is

$\mathcal{E} : [\![Expression]\!] \rightarrow Environment \rightarrow Self \rightarrow Expression$

$\mathcal{E}[\![a\ \texttt{=}\ value]\!] = \lambda e.\lambda s.inAssign(a, \mathcal{V}[\![value]\!]e\ s)$

$$\mathcal{E}[\![\texttt{has}\ a]\!] = \lambda e.\lambda s.inHas(a)$$

$$\mathcal{E}[\![\texttt{in}\ c]\!] = \lambda e.outClass(e(c))$$

$$\mathcal{E}[\![E_1\ \texttt{|}\ E_2]\!] = \lambda e.\lambda s.inOr(\mathcal{E}[\![E_1]\!]e\ s, \mathcal{E}[\![E_2]\!]e\ s)$$

$$\mathcal{E}[\![E_1\ \texttt{\&}\ E_2]\!] = \lambda e.\lambda s.inAnd(\mathcal{E}[\![E_1]\!]e\ s, \mathcal{E}[\![E_2]\!]e\ s)$$

Values are also represented as syntax trees, and again these trees are interpreted in the context of an environment and an instance. The valuation function for values is

$$\mathcal{V} : [\![Value]\!] \rightarrow Environment \rightarrow Self \rightarrow Value$$

$$\mathcal{V}[\![number]\!] = \lambda e.\lambda s.in\mathbb{R}(number)$$

$$\mathcal{V}[\![string]\!] = \lambda e.\lambda s.inString(string)$$

$$\mathcal{V}[\![i]\!] = \lambda e.\lambda s.inI(i)$$

$$\mathcal{V}[\![\texttt{self}]\!] = \lambda e.\lambda s.s$$

$$\mathcal{V}[\!["\{"v_1","\ldots","v_k"\}"]\!] = \lambda e.\lambda s.inSetvalue(\{\mathcal{V}[\![v_j]\!]e\ s | 1 \leq j \leq k\})$$

$$\mathcal{V}[\![\texttt{set(}E\texttt{)}]\!] =$$
$$\lambda e.\lambda s.inSetvalue(\{x \in dom(e) \cap I | satisfies(e(x), \mathcal{E}[\![E]\!]e\ s)\})$$

$$\mathcal{V}[\![\texttt{set(}i\,|\,E\texttt{)}]\!] =$$
$$\lambda e.\lambda s.inSetvalue(\{x \in dom(e) \cap I | satisfies(e(x), \mathcal{E}[\![E]\!]([i \mapsto e(x)]e)\ s)\})$$

where the predicate *satisfies* is true if the instance satisfies the expression (see details on page 53).

Types and Distances

The meaning of a type is a tuple $\langle v, t, c, r\rangle$, where v represents a set of values, t is a transformation function, c is a construction function, and r is a removal function. As with expressions, this meaning is interpreted in the context of an environment. Hence, the signature of the valuation function for types is

$$\mathcal{T} : [\![Type]\!] \rightarrow Environment \rightarrow Type$$

The valuation function for types is defined by cases. If the type includes the "`no distance`" clause, the transformation, construction, and removal functions are constant at zero. Here are all these cases

$\mathcal{T}[\![c\ \texttt{no distance}]\!] = \lambda e.\langle\ inC(c), \lambda\langle x,y\rangle.0, \lambda y.0, \lambda y.0\ \rangle$

$\mathcal{T}[\![\texttt{number no distance}]\!] = \lambda e.\langle\ inReals, \lambda\langle x,y\rangle.0, \lambda y.0, \lambda y.0\ \rangle$

$\mathcal{T}[\![\texttt{string no distance}]\!] = \lambda e.\langle\ inStrings, \lambda\langle x,y\rangle.0, \lambda y.0, \lambda y.0\ \rangle$

$\mathcal{T}[\![\text{"\{"}t_1\text{","}\ \text{"..."}, \text{","}t_n\text{"\}"}\ \texttt{no distance}]\!] =$
$\quad\lambda e.\langle\ inTerms(\{t_1,\ldots,t_n\}), \lambda\langle x,y\rangle.0, \lambda y.0, \lambda y.0\ \rangle$

For real numbers, the default distance from x to y is $y - x$ or zero if negative; the distance from an undefined value to a number is the number. The distance between two strings (or terms) is zero if the strings are equal and is unbounded otherwise; strings cannot be constructed and removal is zero. Terms cannot be constructed or removed.

$\mathcal{T}[\![\texttt{number distance}]\!] =$
$\quad\lambda e.\langle inReals, \lambda\langle is\mathbb{R}(x), is\mathbb{R}(y)\rangle.\max(y-x,0), \lambda is\mathbb{R}(x).x, \lambda y.0\ \rangle$

$\mathcal{T}[\![\texttt{string distance}]\!] =$
$\quad\lambda e.\langle inStrings, \lambda\langle x,y\rangle.(x = y \rightarrow 0 \,\Box\, \infty), \lambda y.\infty, \lambda y.0\ \rangle$

$\mathcal{T}[\![\text{"\{"}t_1\text{","}\ \text{"..."}, \text{","}t_n\text{"\}"}\ \texttt{distance}]\!] =$
$\quad\lambda e.\langle inTerms(\{t_1,\ldots,t_n\}), \lambda\langle x,y\rangle.(x = y \rightarrow 0 \,\Box\, \infty), \lambda y.\infty, \lambda y.\infty\ \rangle$

For attributes with enumerated types, a weighted directed graph is constructed from the distance clause of the attribute declaration. The distance between terms is the weight of the shortest path from the candidate to the target in the distance graph of the attribute.

$\mathcal{T}[\![\text{"\{"}t_1\text{","}\ \text{"..."}, \text{","}t_n\text{"\}"}\ \texttt{distance}\{triples\}]\!] =$

$$\begin{aligned}\lambda e.\langle\ & inTerms(\{t_1,\ldots,t_n\}),\\ & \lambda\langle isT(x), isT(y)\rangle.WSP(x,y),\\ & \lambda isT(y).WSP(Unspecified, y),\\ & \lambda isT(y).WSP(y, Unspecified)\ \rangle\end{aligned}$$

where function WSP computes the weight of a shortest path in the corresponding distance graph. The graph is a weighted digraph $G = (V, E, W)$, such that $V = \{t_1,\ldots,t_n\} \cup \{Unspecified\}$. The edges and weight function are given by the triples (i.e., triple x`->`y`:`n means $(x,y) \in E$ and $W(x,y) = n$), augmented with $(t, Unspecified) \in E$ and $W(t, Unspecified) = 0$ for all $t \in \{t_1,\ldots,t_n\}$.

The distance from a candidate instance to a target instance is computed by adding the distances of their corresponding attribute values. Extra attributes in the candidate are ignored; missing attributes in the candidate must be created; and missing attributes in the target must be removed.

$\mathcal{T}[\![$`c distance`$]\!] = \lambda e.\langle\ inC(c), tranFunct, consFunct, remFunct\ \rangle$

where *tranFunct*, *consFunct*, and *remFunct* are

$$tranFunct = \lambda\langle isI(x), isI(y)\rangle.\ \textstyle\sum_{a\in(dom(x)\cup dom(x))} D(a,x,y)$$

$$\begin{aligned} D(a,x,y) = \textbf{let}\ &\langle v,t,c,r\rangle = e(e(a))\ \textbf{in}\\ &\textbf{cases}\\ &a \in (dom(x)\cap dom(y)) \rightarrow t(x(a),y(a))\\ &a \in (dom(y) - dom(x)) \rightarrow c(y(a))\\ &a \in (dom(x) - dom(y)) \rightarrow r(x(a))\\ &\textbf{otherwise}\ \rightarrow \infty \end{aligned}$$

$$consFunct = \lambda isI(y).\ \textstyle\sum_{a\in dom(y)} \textbf{let}\ \langle v,t,c,r\rangle = e(e(a))\ \textbf{in}\ c(y(a))$$

$$remFunct = \lambda isI(y).\ \textstyle\sum_{a\in dom(y)} \textbf{let}\ \langle v,t,c,r\rangle = e(e(a))\ \textbf{in}\ r(y(a))$$

These functions, as stated, may not converge if there are circular dependencies between instances (e.g., `stack` depends on `push` and `push` depends on `stack`) and a query explicitly mentions these dependencies.

If the type includes an foreign `distance` clause, distance functions are external to EDF (user-supplied). These functions receive the environment, in addition to the corresponding values.

$$\mathcal{T}[\![\texttt{number distance}\{T,C,R\}]\!] = \lambda e.\langle\ inReals, T(e), C(e), R(e)\ \rangle$$

$$\mathcal{T}[\![\texttt{string distance}\{T,C,R\}]\!] = \lambda e.\langle\ inStrings, T(e), C(e), R(e)\ \rangle$$

$$\mathcal{T}[\![c\ \texttt{distance}\{T,C,R\}]\!] = \lambda e.\langle\ inC(c), T(e), C(e), R(e)\ \rangle$$

$$\begin{aligned}&\mathcal{T}[\![\text{"}\{\text{"}t_1\text{"},\text{"}\ldots\text{"},\text{"}t_n\text{"}\}\text{"}\texttt{distance}\{T,C,R\}]\!] =\\ &\quad \lambda e.\langle\ inTerms(\{t_1,\ldots,t_n\}), T(e), C(e), R(e)\ \rangle\end{aligned}$$

For set types, the transformation, construction, and removal function is computed from the functions of the corresponding base type. For every element in the target, the distance from every element in the candidate is computed, and the minimum is chosen; these distances are added.

$$\begin{aligned}&\mathcal{T}[\![\texttt{set of}\ z]\!] =\\ &\quad \lambda e.\ \ \textbf{let}\ \langle v,t,c,r\rangle = e(z)\ \ \textbf{in}\\ &\qquad \langle\ \ inSet(z),\\ &\qquad\quad \lambda\langle isSetvalue(X), isSetvalue(Y)\rangle.\ \textstyle\sum_{y\in Y}\min_{x\in X} t(x,y),\\ &\qquad\quad \lambda isSetvalue(y).\ \textstyle\sum_{y\in Y} c(y),\\ &\qquad\quad \lambda isSetvalue(y).\ \textstyle\sum_{y\in Y} r(y)\ \ \rangle\end{aligned}$$

Instances

An instance defines a mapping from attributes to values. This mapping is defined by an expression which is first converted into disjunctive normal form and then simplified. Type DNF represents an expression in disjunctive normal form as a set of conjuncts, where each conjunct is a set of assignments and **has** propositions.

$$DNF = \mathbb{P}(Conjunct)$$

$$Conjunct = \mathbb{P}(Assign + Has)$$

As explained in Section 3.2.1, an expression E defines a set of points. Function *instantiate* computes the set of points, where each point is represented as a mapping from attributes to values. If there is a mapping that is less defined[4] than all other mappings, that mapping is the instance; otherwise, there is no instance ('$\min_{\sqsubseteq}$' evaluates to $\perp$).

$$instantiate : Expression \rightarrow Instance$$

$$instantiate = \lambda E. \min_{\sqsubseteq}(exprToInstances(E))$$

Function *exprToInstances* creates the set of instances described in the expression (usually the expression will have no disjunctions '|', so this set will contain exactly one instance). To create the instances, the expression is transformed into disjunctive normal form, and each conjunct that is not a contradiction is used to create an instance.

$$exprToInstances : Expression \rightarrow \mathbb{P}(Instance)$$

$$exprToInstances = \lambda E.\{conjToInst(x) | x \in normalize(E) \wedge consistent(x)\}$$

Function *conjToInst* creates an instance (mapping) from a set of ordered pairs, by choosing for each attribute a in the corresponding domain the value that is most defined (usually the expression will have one value for each attribute).

$$conjToInst : Conjunct \rightarrow Instance$$

$$conjToInst = \lambda x.\lambda a. \max_{\sqsubseteq}\{v | \langle a, v' \rangle \in x\}$$

A conjunct is consistent if all attributes for which there is a **has** proposition have some value, and if whenever there are more than one value for the same attributes, there exists a maximal value.

[4]A partial function f is less defined than a partial function g, denoted $f \sqsubseteq g$ if $\forall x \in dom(f) : f(x) = g(x)$ [84, chapter 6].

$$consistent : Conjunct \rightarrow Boolean$$

$$\begin{aligned} consistent = \lambda x.\quad & (\forall a \in A : inHas(a) \in x \Rightarrow \exists v : inAssign(a, v) \in x) \wedge \\ & (\forall \langle a, v\rangle, \langle a, v'\rangle \in x : v' \sqsubseteq v \vee v \sqsubseteq v') \end{aligned}$$

Function *normalize* transform an expression in syntax-tree from into disjunctive normal form (a set of conjuncts). The definition given below is deceivingly compact.

$$normalize : Expression \rightarrow DNF$$

$$\begin{aligned} & normalize = \\ & \quad \lambda E.\ \textbf{cases}\ E \\ & \qquad isOr(a, b) \rightarrow normalize(a) \cup normalize(b) \\ & \qquad isAnd(a, b) \rightarrow \{x \cup y | x \in normalize(a) \wedge y \in normalize(b)\} \\ & \qquad \textbf{otherwise} \ \rightarrow \{\{E\}\} \end{aligned}$$

Functions $\min_{\sqsubseteq}$ and $\max_{\sqsubseteq}$ are defined below.

$$\min_{\sqsubseteq} : \mathbb{P}(Instance) \rightarrow Instance$$

$$\min_{\sqsubseteq} = \lambda S.\iota\ s \in S : \forall s' \in S : s \sqsubseteq s'$$

$$\max_{\sqsubseteq} : \mathbb{P}(Value) \rightarrow Value$$

$$\max_{\sqsubseteq} = \lambda S.\iota\ s \in S : \forall s' \in S : s' \sqsubseteq s$$

These functions depend on the definition of $\sqsubseteq$ for instances. In addition, $\max_{\sqsubseteq}$ requires $\sqsubseteq$ to be defined for values that are not instances. For set values, $A \sqsubseteq B$ iff $\forall x \in A : \exists y \in B : x \sqsubseteq y$. For other values $\sqsubseteq$ is equality.

Queries

We represent the meaning of a query command as a set of ordered pairs $\langle i, n\rangle$ where i is an instance name in the library and n the distance from i to the implicit instance defined by E.

$$\mathcal{Q} : [\![Expression]\!] \rightarrow Environment \rightarrow \mathbb{P}(I \times \mathbb{R})$$

$$\begin{aligned} & \mathcal{Q}[\![\texttt{query}\ E]\!] = \\ & \quad \lambda e.\{\ \ \langle x, n\rangle | x \in dom(e) \cap I \wedge \\ & \qquad\quad n = tranFunct(e(x), instantiate(\mathcal{E}[\![E]\!]e\ x))\ \} \end{aligned}$$

where *tranFunct* is defined on page 52 and *instantiate* is defined on page 53.

3.4 Summary

This chapter has introduced and formally defined the main aspects of EDF, a language-based classification system whose purpose is to facilitate the reuse of software knowledge in an organization. EDF is best described as a *software librarian* system, in which existing software components are cataloged and stored in libraries for future reference. Whenever a new component needs to be constructed, EDF's similarity-based retrieval mechanism helps retrieve from these libraries the set of reuse candidates that best match the specifications of the required new component.

The basic principles that support this methodology have all been explained in this chapter. In particular, we have learned about EDF's representation and similarity models, how these models are implemented by the EDF specification language, and the syntax and semantics of this language. This language forms the core of the EDF system for several reasons. First, it provides the facilities for designing taxonomies and creating libraries of object descriptions. Second, it allows to define similarity metrics which are used to query a library for candidate components. Third, it provides a mechanism to define semantic relations among the attributes of a class of objects.

The remainder of this dissertation emphasizes the uses of EDF. Chapter 4 shows the applicability of EDF to a wide range of domains by presenting taxonomies for "software defects", "software processes", and other domains. Chapter 5 describes the construction of an EDF reuse library based on information contained in the NASA SEL database. The main emphasis of this chapter is on the design and implementation of distance comparators. Appendix A describes the details of a prototype application that implements EDF.

Chapter 4

Sample EDF Taxonomies

EDF was initially designed as a tool to help increase reusability of software components at the *code* level (e.g., functions or subroutines). The goal of this chapter is to show that EDF can also be used effectively to represent and reuse other types of software knowledge. The emphasis here is on those properties of EDF that facilitate the representation of these objects, and *not* on how to define similarity distances for these objects. The problem of designing and using similarity distance comparators is addressed in Chapter 5.

This chapter is divided into three sections. Section 4.1 describes the representation of two software component libraries: the GRACE library of data structure operations, and the CCIS library which contains functions used to implement Command, Control, and Information Systems. Section 4.2 presents a taxonomy for software defects, and explains how this taxonomy can be used by system testers to predict the types of defects associated with software components. Finally, Section 4.3 describes how EDF can be used to characterize elements of Rombach's MVP-L language [81], as well as Basili's Goal/Question/Metric (GQM) paradigm for selecting software measurements [8, 9].

4.1 Software Components

This section deals with the representation of non-trivial software components such as data structure operations and functions for implementing Command, Control, and Information Systems. Both of the taxonomies presented here were initially design for the AIRS system (see Section 2.3) based on the documentation of actual code libraries developed by independent software organizations.

4.1.1 Data Structure Packages in Ada

The EVB GRACE library (part I) developed by the EVB corporation contains a collection of Ada packages that implement data structures such as stacks and undirected graphs. In this section we develop a taxonomy for classifying the components of the EVB GRACE library and creating an EDF reuse library with the purpose of facilitating their reuse.

The purpose of the EDF GRACE library is to allow a software developer to retrieve data structure packages and/or operations that best fit a set of design specifications. Given the specifications for a set of operations, EDF can propose a list of candidate components in the library for each of the members of this set. Once the candidate operations have been selected, EDF can be used to find a package whose operations best approximated the selected candidates. This was particularly important in the case where the selected candidates belong to different data structure packages.

Objects in the EDF GRACE library are classified in two different classes: *packages* and *components*, where each of these classes has a unique set of attributes. Packages are classified using a subset of the features of Booch's taxonomy [20], namely *controlled, managed, allocation* and *iterator*. These attributes capture the general *functional* behavior and the *time* and *space* characteristics of a data structure. The class of all packages is defined in EDF as follows:

```
Package = class(has pkName & has pkControlled &
                has pkAllocation & has pkManaged &
                has pkIterator & has pkOpers);
```

That is, the set of all data structure packages in the library is defined in terms of the attributes: pkName, pkControlled, pkAllocation, pkManaged, pkIterator, and pkOpers. These attributes are defined in EDF as follows:

```
attribute pkName       : string;
attribute pkControlled : {Controlled,unControlled};
attribute pkIterator   : {Iterator,nonIterator};
attribute pkAllocation : {Bounded,unBounded,Limited};
attribute pkManaged    : {Managed,unManaged};
attribute pkOpers      : set of Operation;
```

The attribute pkName is bound to a descriptive text of the package. The attribute pkControlled indicates whether the data structure can be accessed concurrently by more than one process, and the attribute pkIterator indicates whether the package provides functionality to traverse all the elements of the data structure.

The attribute pkAllocation describes the memory allocation scheme used to implement the package. In particular, if the package size is "unbounded", the attribute pkManaged indicates whether memory allocation is performed by the system (unManaged) or by a user-provided set of memory allocation functions (Managed). This semantic relation between the attributes pkControlled and pkManaged is expressed with the following EDF assertion:

```
assertion pkManaged=Managed => pkAllocation=unBounded;
```

That is, only "unbounded" packages can have their memory "managed" by user-provided memory allocation functions.

The attribute pkOpers is bound to the list of all operations of the package. Its associated type indicates that the elements of this list must all be members of the class of objects called "Operation" (described below).

The other class of objects in the EDF GRACE library are data structure operations. All operations in the library belong to the class called "Operation" defined as follows:

```
Operation = class(has opType & has opKey & has opCount &
                  has opTarget & has opRange &
                  has opDirection & has opPackage);
```

The attribute opType defines the functionality of the operation by describing how it interacts with the elements (nodes) that compose the data structure. The operation may "create" a data structure; "select", "insert" or "remove" a set of nodes and or links; "traverse" the structure; or "query" properties of the structure such as its size or length.

```
attribute opType : {Create,Select,Insert,Remove,Traverse,Query};
```

The attribute opTarget indicates the type of data structure elements affected or selected by the operation. This may be either a set of nodes, one node, or a link between nodes. The number of elements affected or selected is defined by the attribute opCount, and the attribute opKey indicates the type of key value used to select elements in the structure.

```
attribute opTarget : {NodeSet,Element,Link};
attribute opCount  : {All,One,Zero};
attribute opKey    : {Index,Pointer,Value,Size};
```

The attributes opRange and opDirection are used to define the relative location of the elements affected or selected by the operation. The former indicates a range of elements within the structure. The latter, defines a direction, relative to the value of opRange, on which the component will operate.

```
attribute opRange     : {FirstLast,FirstTo,FromLast,FromTo,
                         Rest,Floating,First,Second,Last};
attribute opDirection : {Left,Right,ToRight,ToLeft,
                         Breadth,Depth};
```

Finally, the attribute opPackage defines the package to which the operation belongs. It is defined as follows:

```
attribute opPackage : Package;
```

Having defined the classes for packages and operations, we can now present some samples of actual object definitions of the EDF GRACE library[1]. The following is the description of a data structure package that implements a circular doubly linked list. This data structure package can work in a concurrent environment and its maximum size is unbounded. It also provides functionality for traversing the structure, and it uses user-provided memory allocation functions for managing its memory structure.

```
CDLL_cmui =
  [
  in Package & pkName = 'Circular Doubly Linked List' &
  pkControlled = Controlled & pkAllocation = unBounded &
  pkIterator = Iterator & pkManaged = Managed &
  pkOpers = set(in Operation & opPackage=self)
  ];
```

The list of operations of the package are not given explicitly, but are computed automatically by EDF using the built-in "set" function. In this case, the set includes all those objects in the class "Operation" whose opPackage attribute value is equal to self, that is, to the name of the package being defined (CDLL_cmui). The definitions given below show two such operations: cll_InsertFirst and cll_CreateCopy. The operation cll_InsertFirst inserts one element in the head of the circular linked list given a pointer to this head. The operation cll_CreateCopy, on the other hand, creates a copy of the entire structure.

```
cll_InsertFirst =
  [
  in Operation &
  opType = Insert &
  opCount = One &
  opTarget = Element &
  opRange = First &
  opDirection = Left &
  opKey = Pointer &
  opPackage = CDLL_cmui
  ];
```

```
cll_CreateCopy =
  [
  in Operation &
  opType = Create &
  opCount = All &
  opTarget = NodeSet &
  opRange = FirstLast &
  opDirection = ToRight &
  opKey = Pointer &
  opPackage = CDLL_cmui
  ];
```

[1]The complete list of package and operation definitions of the EDF GRACE library is too long to include here.

4.1.2 Command, Control, and Information System

In this section we describe a taxonomy for classifying the different the modules and functions that compose the CTC CCIS library and creating an EDF reuse library with the purpose of facilitating their reuse. The CCIS library developed at Contel Technology Center (CTC) is composed of several modules implemented in C (see below). These modules are used to implement the basic functionalities of Command, Control, and Information Systems.

- **General** (GEN): general purpose functions that do not belong to any specific module. These functions are typically extensions to the ones contained in the standard C library.
- **Memory File** (MF): implements sequential files allocated in main memory (RAM). These files are created and exist only during the execution of a program.
- **Set Structure** (SET): implements unbounded sets of elements. The elements of a particular set must be of the same type.
- **Database Interface** (IDB): provides a simplified interface to the most commonly used operations of a relational database system.
- **Database File** (DBF): implements a specialized form of database files. These files are flat structures stored in a relational database processor.
- **Mail Service** (MS): implements the basic functionalities of an electronic mail system.
- **Man-Machine Interface** (MMI): implements a graphic user interface based on windows, predefined keys, and menus.
- **Free Text File** (FTF): implements a specialized form of text files which are stored in and retrieved from on a relational database.
- **Parametric Database Display** (PPD): collection of parametric functions used to retrieve and display information contained in a relational database.

As with the EDF GRACE library, the EDF CCIS library included two types of objects: *modules* and *functions*. The former represent the different C modules of the CTC CCIS library, and the latter represent their associated C functions. Modules are described using four attributes according to the following class definition:

```
Module = class(has mdAllocation & has mdIterator &
               has mdService & has mdOpers);
```

The attributes `mdAllocation`, `mdIterator`, and `mdOpers` have definitions similar to the ones given for the corresponding attributes in Section 4.1.1. The attribute `mdService` describes the services provided by the functions of the package (e.g., memory management, mail delivery, etc.). The definition of this attribute is given below.

```
attribute mdService : {General,Set,MemoryFile,DatabaseInterface,
                       DatabaseFile,MailService,FreeTextFile,
                       ManMachineInterface,DatabaseDisplay};
```

The functions of each package in the EDF CCIS library were described in terms of two attributes: `fnFunction` and `fnObject`. The `fnFunction` attribute describes the functionality of a component, and it is defined as follows. These terms were extracted from the documentation of the CTC CCIS library.

```
attribute fnFunction : {add,assign,clear,close,convert,copy,
                        count,create,delete,display,enable,
                        execute,find,goto,intersect,log,map,
                        measure,modify,open,parse,process,
                        read,rename,replace,retrieve,search,
                        suspend,terminate,test,transfer,
                        union,write};
```

The attribute `fnObject` describes the kind of object produced or consumed by the function, and is defined as follows. Again, these terms were extracted from the documentation of the CTC CCIS library.

```
attribute fnObject : {address,code,column,column_type,
                      control_variable,descriptor,directory,
                      element,event,file,function_key,group,
                      interface,keyboard,list,menu,name,offset,
                      owner,pdd_descriptor,pdd_page,permission,
                      pointer,ppd_table,printer,queue,subset,
                      queue_entry,record,set,sql_command,string,
                      substring,text,tuple};
```

One of the difficulties of posing queries in a library so rich in terminology is remembering the precise terms used to describe functions. To facilitate this situation, the EDF CCIS library included a list of synonym definitions for some of the terms of the attributes `fnFunction` and `fnObject`. The following are some sample synonym definitions:

```
update = write;                          sequence = string;
insert = add;                            location = address;
remove = delete;                         node = element;
```

These synonym definitions were made part of the EDF CCIS library by including them as terms of their respective attributes, and then defining the distance between them and their synonym terms as zero.

4.2 Software Defects

One obvious necessity of software systems is the ability to function without defects. Traditional software construction processes have specific subprocesses to detect defects (e.g., "unit test", "acceptance test"). However, detecting faults is not enough: to reduce the number of defects associated with a product and its development process requires the ability to explain and predict them. The ability to explain a defect helps to find its source, thus reducing the cost associated with its correction. In addition, being able to predict defects in a software system helps to select processes, methods and tools to avoid defects of a particular kind, reducing the need for later detection and correction procedures. Prediction also helps to improve the effectiveness of testing mechanisms by increasing the chances of finding defects.

In order to explain and predict software defects, we need to characterize the different kinds of defects associated with a particular software environment and project [10]. We also need to understand the relationships between defects associated with a product and its attributes. EDF can be used as an effective tool to model software defects. In particular, queries can help both to explain the cause of defects and to predict them in a particular software environment.

4.2.1 Characterizing Defects Using EDF

In this section we present an EDF classification made by Straub and Ostertag [91] which is itself based on IEEE standard terminology as presented by Basili and Rombach [10]. The reader should be aware that this classification scheme is specific to the particular environment we intend to use for our example.

Kinds of Defects

A software product can be defined by two distinct types of entities: data and processes. The first attribute we use to discriminate among defects is whether they are directly associated with processes or with documents[2] . If a defect is related to a document, it is called a fault. If it is related to a process, it is

[2]A "document" includes any form of representation of software information (e.g., books or electronic files) such as: requirements or specification documents, source code, or help files.

called either a failure or an error: failures are associated with processes that are performed automatically and errors are associated with human processes.

The attribute `entity` classifies the kind of entity (either data or process) in which the defect occurs. The attribute `creator` classifies the creator or agent of that entity (either computer or human). These attributes are used to define faults, errors, and failures.

```
attribute entity  : {data,process};
attribute creator : {computer,human};
defects  = class(has entity | has creator);
faults   = class(entity=data);
failures = class(entity=process & creator=computer);
errors   = class(entity=process & creator=human);
```

Cause of Defects

Failures, faults and errors are interrelated. Failures are caused by one or more faults[3]. For example, a failure during the execution of a program is caused by a fault in the program. Faults in a document are the consequence of defects in the processes that create the document or in the data used by these processes. For example, failure in a software tool can produce a fault in a document. The `cause` attribute describes these relationships. Because we do not model human processes, this attribute does not apply to errors.

```
attribute cause : set of defects;
assertion has cause => in failures | in faults;
```

Severity of a Defect

Another way to characterize defects is by their severity. This information helps prioritize activities aimed at correcting defects. We distinguish four levels of severity: fatal (stops production or development completely), critical (impacts production or development significantly), noncritical (prevents full use of features), and minor.

```
attribute severity : {fatal,critical,noncritical,minor};
```

Defects and the Lifecycle

We are interested in determining when and where a defect enters the system and when it is detected. Because the phases of the lifecycle are related to documents (e.g., the requirements phase is related to the requirements document), we use

[3]System failures are also caused by environmental accidents; here we only consider software-related failures.

phases to measure the time at which errors and failures occur as well as to determine the (kind of) document in which a fault occurs. The `occurrence` attribute relates a defect to the phase at which it enters the system. The `detection` attribute relates a defect to the phase at which it is detected. We explicitly declare the `phase` type that is used in these two attributes.

```
type phase = {requirements,specification,design,coding,
              unit_test,integration,integration_test,
              acceptance_test,maintenance,operation};
attribute occurrence : phase;
attribute detection  : phase;
```

So far we have defined attributes to characterize defects in general. The remaining analysis defines specific kinds of failures, faults and errors.

Kinds of Failures

A failure occurs during the execution of either the software product or a software tool. Our focus is on failures associated with the execution of a particular kind of software product: implementation of data structures.

```
attribute failure_kind :
      {overflow,underflow,illegal_access,
       wrong_output,infinite_loop,tool_failure};
assertion has failure_kind => in failures;
```

Kinds of Faults

Faults are defects in documents: they occur in executable documents (i.e., code) and also in other types of documents. Again, our focus is on documents interpreted by the computer, so we consider only faults on those documents.

```
attribute fault_kind :
      {control_flow,interface,algebraic_computation,
       data_definition,data_initialization,data_use};
assertion has fault_kind => in faults;
```

In general it is difficult to isolate defects in documents. However, if a particular area in a document contains a defect, one is interested in knowing whether something is missing (omission) or something is wrong (commission). We use the `fault_mode` attribute to distinguish between this two cases.

```
attribute fault_mode : {omission,commission};
assertion has fault_mode => in faults;
```

Kinds of Errors

Defects introduced by humans (i.e., errors) are ultimately the cause of most other types of defects in a software product, hence understanding their nature is critical. On the other hand, a complete characterization of errors involves modeling human processes which is out of the scope of this work. We simply characterize errors by the particular domain that is misunderstood or misused, using the error_kind attribute.

```
attribute error_kind :
      {application_area,problem_solution,syntax,
       semantics,environment,clerical};
assertion has error_kind => in errors;
```

4.2.2 Sample Descriptions

The following examples of defects and their characterization use the proposed classification scheme. The particular software project is the construction of a package to manipulate hash tables.

Case 1

Consider a programmer coding a particular function which according to the specifications must receive as input two integer arguments. The programmer understands exactly what must be implemented, but mistakenly declares the function with only one formal argument. This fault is detected while reading code during unit testing. These defects are classified as follows.

```
fault_1 = [in faults & occurrence=coding &
           detection=unit_test & severity=critical &
           cause={error_1} & fault_mode=omission &
           fault_kind=interface];

error_1 = [in errors & error_kind=clerical];
```

Case 2

Consider the case that deletions in a hash table do not always reclaim storage. This causes a system crash during operation due to an overflow in a hash table; the problem is corrected promptly by reformatting the table. The specific problem is that a code optimizer swapped two statements. These defects are classified as follows.

```
failure_2 = [in failures & severity=noncritical &
             occurrence=operation & cause={swapped_stmt} &
             failure_kind=overflow];

swapped_stmt = [in faults & severity=critical &
                occurrence=coding &
                detection=operation &
                cause={failure_op} &
                fault_kind=control_flow &
                fault_mode=commission];

failure_op = [in failures & occurrence=coding &
              detection=operation &
              failure_kind=tool_failure];
```

4.2.3 Explaining and Predicting Defects

Having a database with software components, software defects, and their interrelations is useful to explain and predict defects. These explanations/predictions are not automatic: they are done by a person who obtains relevant information using queries to the database. (We assume that distances between terms of all attributes are defined.)

The following is a description of a failure that has been diagnosed as an overflow in a data structure; this failure occurred during integration test.

```
overflow_fail = [in failures & severity=fatal &
                 occurrence=integration_test &
                 failure_kind=overflow];
```

We do not know the kind of fault that caused `overflow_fail`, so we query the database for faults that have caused similar failures using the following query command.

```
query in faults & occurrence=coding & cause={overflow_fail}
```

To predict defects in packages, defect descriptions must be integrated with package descriptions in a single database. We relate packages with their faults (and thus indirectly with errors and failures) by adding attributes to both packages and faults. The `docum` attribute for faults is the package in which the fault occurs; the `fault_set` attribute for packages describes the set of known faults.

```
attribute docum : packages;
assertion has docum => in faults;

attribute fault_set : set of faults;
assertion has fault_set => in packages &
                              fault_set=set(docum=self);
```

Assume we want to predict the kinds of defects that may be associated with the hashing data structure package. The following query retrieves packages that are similar to the hash package. The subunits are assumed to be already defined.

```
query maxsize=bounded & control=sequential
      & subunits={hash_create, hash_insert,
                  hash_lookup, hash_delete}
```

Assuming that similar packages will have similar defects, we can use the faults of the retrieved packages to predict the faults that may occur in the hash package.

4.3 Other Software-Related Domains

We show how EDF can be used to characterize the objects of two proposed models for describing and improving the process of software development. The first model is the Goal/Question/Metric paradigm for selecting software measurements. The second is the MVP-L language for describing software processes. It is important to note that the EDF taxonomies presented in this section are by no means complete, and should only be considered as "base proposals" for future research.

4.3.1 The Goal/Question/Metric Paradigm

The Goal/Question/Metric (GQM) paradigm consists of three major steps [8]: (1) identification of a set of *goals* of an organization relative to some software object (i.e., product or process); (2) definition of a set of *questions* that, if answered, will allow the organization to reach these goals; and (3) selection of *measurements* to answer these questions.

The description of these goals, questions, and measurements form, what I will call, an *object evaluation model* (OEM). The purpose of these models is to help analyze and improve the different products of a software organization and the processes it uses to produce them. Creating these models is complex and requires experience, yet, during the life time of a particular project, an organization may require a great number of different OEMs to evaluate things such as small functions and software modules, as well as design, coding and test procedures.

One way to help reduce the effort of creating new object evaluation models is by reusing previous ones. As we have already discuss, this approach would require the existence of a library of old OEMs, as well as a mechanism for retrieving them based on incomplete or (possibly) incorrect specifications. This type of scenario is exactly what EDF was designed for, that is, to help create reuse libraries and retrieve objects from these libraries based on similarity considerations.

The first step in the implementation of such a reuse scheme is the design of a taxonomy for OEMs. Object evaluation models will be characterized according to the following attributes: *purpose*, *perspective*, and *environment*. That is, an object evaluation model (OEM) belongs to the EDF class:

```
OEM = class(has purpose & has perspective & has environment);
```

These attributes are explained in the following sections, and are then applied to an example of an object evaluation model.

The purpose attribute

As its name implies, the purpose attribute describes the purpose of the study. This attribute has two dimensions: the *object* of the study (e.g., process or product), and the *goal* of the study (e.g., predict or improve). These two dimensions will be described in terms of the attributes targetObject and studyGoal, respectively. In other words, the purpose attribute is defined as:

```
attribute purpose : class(has targetObject & has studyGoal);
```

The targetObject attribute describes the relevant features that characterize the object that is the target of the study. This description focuses on the internals of the object, that is, those characteristics that differentiate it from the other potential objects of study, independent of its environment.

Target objects will be divided in two disjuncts classes according to the attribute targetType: *products* and *processes*. That is, the values of the attribute targetObject belong to the class of instances defined with the targetType attribute:

```
attribute targetObject : class(has targetType);
attribute targetType   : {product,process};
```

The actual attribute definition of a target instance will depend on its type. For example, if the target object is a software product such as an Ada package or operation, its description could be given using the taxonomies presented in Section 4.1. On the other hand, if the target object is a process, its description can be based on the taxonomy presented in Section 4.3.2.

As mentioned earlier, the attribute studyGoal defines the goal or objectives of the study, that is, whether we are interested on improving, assessing, or predicting certain properties of the target object. A simple definition of this attribute is as follows:

```
attribute studyGoal : {improve,assess,predict,motivate};
```

The perspective attribute

The attribute `perspective` is meant to define a particular angle for the evaluation. This attribute also has two dimensions: the *agent* who is going to make use of the results of the evaluation (e.g., manager or developer), and the *aspect* of the target object that will be improved or predicted (e.g., cost or reliability). These two dimensions are described using two attributes called `pointOfView` and `aspect`, respectively. The attribute `perspective` is then defined as follows:

```
attribute perspective : class(has pointOfView & has aspect);
```

The attribute `pointOfView` is important to filter the different features, models, measurements, and results that are important to the agent interested in the study. A simple definition of this attribute is as follows:

```
attribute pointOfView : {developer,designer,manager,customer};
```

The attribute `aspect`, on the other hand, complements the description given by the attribute `purpose` in that it pinpoints the particular aspect or characteristic of the target object we are interested in studying. A simple definition of this attribute is as follows:

```
attribute aspect : {cost,defects,reliability,changes};
```

The environment attribute

The attribute `environment` describes the relevant features that characterize the environment in which the object is going to be studied. This attribute complements the information given by the attribute `purpose`, and it explains the relation between the object and its environment.

The environment of an attribute is obviously domain dependent. As an example, we will characterize a simple context in terms of two attributes: `cpuType` and `applType`. That is, the attribute `environment` is defined as follows:

```
attribute environment : class(has cpuType & has applType);
```

Where attribute `cpuType` defines the type of CPU on which the target object will be executed, while attribute `applType` describes the type of functionality provided by the application the target object is part of.

```
attribute cpuType  : {VAX_780,VAX_8600,IBM_360};
attribute applType : {SatelliteSupport,DataStructures};
```

Example

The following definition of an Object Evaluation Model was taken from "The Goal/Question/Metric Paradigm" [9].

> *Analyze the system testing methodology for the purpose of assessment with respect to a model of defects from the point of view of the developer in the following context: the standard NASA/GSFC*[4] *environment; i.e., the process model is the SEL*[5] *version of the Water Fall model, the application is the ground support software for satellites, and the machine is a DEC/VAX 780 running VMS.*

This description can be characterized in EDF as follows:

```
NASA_GSFC = [
            purpose =
                [
                targetObject = SystemTestMethod &
                studyGoal = assess
                ] &
            perspective =
                [
                pointOfView = developer &
                aspect = defects
                ] &
            environment =
                [
                cpuType = VAX_780 &
                applType = SatelliteSupport
                ]
            ];
```

Where `SystemTestMethod` is the name of an instance with attribute `targetType` equal to `process`, and whose remaining attributes describe the "system testing methodology" in terms of, for example, the taxonomy presented in Section 4.3.2.

Reusing Object Evaluation Models

The goal of this section has been to present an EDF taxonomy for describing objects in the GQM paradigm as domain independent as possible. Although this

[4]GSFC stands for "Goddard Space Flight Center".

[5]SEL stands for "Software Engineering Laboratory".

presentation defines the basic elements of such a taxonomy, producing an actual library of OEMs would require tailoring the above definitions to a particular organization. Such an improved taxonomy could be used to facilitate the design of new OEMs by reusing similar OEM specifications. That is, given a partial description of a target OEM, EDF could be used to find similar descriptions which could help complete or refine the definition of the target OEM.

For example, consider an EDF library of Object Evaluation Models described using the attributes previously defined (i.e., purpose, perspective, and environment) in addition to a fourth attribute called `execution`. This new attribute would be used to describe the actual process and tools that were used to perform the evaluation. Given such a library of OEMs, it would be possible to use EDF to discover which processes and tools provide the best alternative to perform a new evaluation. For example, consider the following evaluation description similar to that of `NASA_GSFC` (see above):

> *Analyze the design process for the purpose of improvement with respect to a model of defects from the point of view of the designer in the context of the standard NASA/GSFC environment.*

That is, the target object is now the "design process" instead of the "system testing methodology"; the goal is improvement rather than assessment; and the point of view is that of the designer and not the developer. All the rest is identical to that of the `NASA_GSFC` model. The following `query` would find all those OEMs similar to the required one.

```
query purpose=[targetObject=DesignMethod & studyGoal=improve] &
      perspective=[pointOfView=developer & aspect=defects] &
      environment=[cpuType=VAX_780 & applType=SatelliteSuport]
```

If we assume that similar OEM descriptions imply similar executions, then the value of the `execution` attribute of the best reuse candidate proposed by this query should provide information of which processes and tools ought to be considered first to carry out this new evaluation.

4.3.2 Software Process Descriptions

This section presents an EDF taxonomy for describing elements of Rombach's process modeling language MVP-L[6]. This language was designed to help build descriptive process models, package them for reuse, integrate them into prescriptive project plans, analyze project plans, and use these project plans to guide future projects.

The MVP language describes software projects in terms of the following five entities: project plans, process models, product models, resource models, and attribute models. These entities are explained below:

[6]MVP stands for "Multi-View Process modeling".

- **Attribute model:** defines observable characteristics of software processes, products, or resources. Each attribute model binds an identifier to an elementary data types such as integer, real, or an enumeration. The model also defines the possible changes in value of this identifier.

- **Resource model:** describes the active and passive resources used to execute a process. Active resources are organizational entities or humans designated to execute processes. Passive resources are tools which are used to support the execution of a process (e.g., an editor).

- **Product model:** describes the structure of a class of products such as requirements or design documents. That is, a product model defines an object which is consumed or produced by a process within a project.

- **Process model:** describes the structure of a class of processes such as the design or coding phases of a software project. A process model is mainly defined by the set of resources it requires and the products it consumes and produces. Its activation is characterized by a boolean entry condition, and it terminates when a particular exit condition is satisfied.

- **Project plan:** combines process, product, and resource models to define an overall software project. That is, it defines a partial ordering of a collection of processes, the types of products that flow among these processes, and the resources required to consume and produce these products.

Although the MVP-L language was designed to help increase the reusability of software process descriptions and to help package this type of knowledge, the number of entities required to describe a project and their complexity is very high. One way to help reduce the effort of creating new models is by reusing previous ones. In particular, we propose a reuse mechanism based on EDF to create reuse libraries of MVP-L models with the purpose of retrieving them using similarity considerations.

According to the definition of MVP-L, a "project plan" model is defined by a name and three clauses: *imports*, *objects*, and *object relations*. The "imports" clause lists all elementary models used to declare the process, product, and resource objects making up the project plan. These objects are defined in the "objects" clause, and are interconnected according to their formal interfaces in the "object relations" clause.

The EDF characterization of "project plan" models will follow the same structure as in MVP-L, that is, they will be described in terms of three attributes called: `imports`, `objects`, and `relations`, which define the following EDF class:

```
ProjectPlan = class(has imports & has objects & has relations);
```

The imports attribute

The imports attribute is used to describe the "imports" clause of project plan models in MVP-L. As mentioned earlier, this clause lists all elementary models used to declare the process, product, and resource objects that make up the project. The following is an example of an MVP-L "imports" clause:

```
imports
  product_model  Requirements_document;
  product_model  Design_document;
  resource_model Design_group;
  process_model  Design;
```

This type of structure can easily be represented in EDF by a list of objects. That is, the imports attribute can be defined as follows:

```
attribute imports : set of ImportModels;
```

Where ImportModels is the name the class of all instances in the EDF library that can form part of the "imports" clause. These instances can be of one of three types: products, processes, and resources, therefore attribute ImportModels is defined as follows:

```
ImportModels = class(in Products | in Processes | in Resources);
```

That is, any instance in the EDF library which belongs to the class Products or Processes or Resources will also be a member of the class ImportModels, and, therefore, will be a valid member of the set defined by the attribute imports. The above classes are defined as follows:

```
Projects   = class(mType=project);
Products   = class(mType=product);
Processes  = class(mType=process);
Resources  = class(mType=resource);
Attributes = class(mType=attribute);
attribute mType : {project,product,process,resource,attribute};
```

It is important to note that the attribute mType does not fully characterize a model, it just defines its type. Additional attributes are necessary to describe the particular properties of the models, but, as mentioned earlier, only those of "project plans" will be presented here.

The `objects` attribute

The `objects` attribute is used to describe the "objects" clause of project plan models in MVP-L. As mention earlier, this clause defines the process, product, and resource objects that make up the project. The following is an example of an MVP-L "objects" clause:

```
objects
  rdoc: Requirements_document('complete');
  ddoc: Design_document('non_existing');
  dgrp: Design_group;
  dprc: Design(2000);
```

The "objects" clause is composed of a list of object identifiers. The type of each identifier is one of the models imported with the "imports" clause, and each model is instantiated by supplying values to its formal parameters. This structure can be described in EDF by a list of objects, where each object represents a mapping from identifiers to types. That is, the attribute `objects` is defined as follows:

```
attribute objects : set of PlanObjects;
```

Where `PlanObjects` defines the class of instances in the library defined in terms of two attributes: `objectModel` and `modelParams`. The former records the model type of the object, while the latter is used to record its instantiation parameters. The definition of these attributes is as follows:

```
PlanObjects = class(has objectModel & has modelParams);
attribute objectModel : ImportModels;
attribute modelParams : string;
```

The `relations` attribute

The `relations` attribute is used to describe the "object_relations" clause of project plan models in MVP-L. As mention earlier, this clause interconnects the objects defined in the "objects" clause according to their formal interfaces. The following is an example of an MVP-L "objects" clause which maps the formal interfaces `b`, `c`, and `d` of the process `dprc` to the objects `rdoc`, `ddoc`, and `dgrp` defined in the previous section:

```
object_relations
  dprc(b => rdoc, c => ddoc, d => dgrp)
```

As was the case with the attributes `imports` and `objects`, the `relations` attribute defines a list of objects. In this case, each object represents a mapping between process interfaces and objects defined in the project plan module.

```
attribute relations : set of ProcMappings;
```

Here ProcMappings is the name of the class of instances in the library defined in terms of the following attributes: processObject, mappedObjects. The former defines the name of the process object, while the latter lists the project plan object being mapped. These attributes are defined as follows:

```
ProcMappings = class(has processObject & has mappedObjects);
attribute processObject : Processes;
attribute mappedObjects : set of PlanObjects;
```

This representation does not include the names of the formal interfaces of the process. It simply assumes an ordering of these interfaces and lists the objects of the project plan that are being mapped. The other names mentioned here (i.e., Processes and PlanObjects) were defined in the previous sections.

Example

The following MVP-L process description was taken from Rombach's technical report "MVP-L: A Language for Process Modeling In-The-Large" [81].

```
project_plan Design_project_2 is
  imports
    product_model  Requirements_document;
    product_model  Design_document;
    resource_model Design_group;
    process_model  Design
  objects
    rdoc: Requirements_document('complete');
    ddoc: Design_document('non_existing');
    dgrp: Design_group;
    dprc: Design(2000)
  object_relations
    dprc(b => rdoc, c => ddoc, d => dgrp)
end project_plan Design_project_2
```

The EDF description of the project plan Design_project_2 is the following:

```
Design_project_2 =
  [
  imports    = {ReqDoc,DesDoc,DesGrp,Design} &
  objects    = {rdoc,ddoc,dgrp,dprc} &
```

```
    relations = {rel_2}
    ];

  ReqDoc = [in Products  & ...];
  DesDoc = [in Products  & ...];
  DesGrp = [in Resouces  & ...];
  Design = [in Processes & ...];

  rdoc = [objectModel=ReqDoc & modelParams='complete'];
  ddoc = [objectModel=DesDoc & modelParams='non_existing'];
  dprc = [objectModel=Design & modelParams='2000'];
  dgrp = [objectModel=DesGrp];

  rel_2 = [processObject=dprc & mappedObjects={rdoc,ddoc,dgrp}];
```

The definitions of the instances `ReqDoc`, `DesDoc`, `DesGrp`, and `Design` are missing some attribute values. This is denoted by an ellipsis symbol, and, for reasons explained earlier in this section, these attributes are not included here. The definition of `Design_project_2` was separated into several instance definitions with the purpose of improving clarity. It is also true that most of these "auxiliary" definitions would normally exist in the library, so the way they are being used here is representative of a typical project plan definition.

Predicting Failures in Processes

One interesting use of a library of process models is related to the prediction of defects associated with process. To predict defects in processes, their descriptions must include descriptions of the failures found during their execution. That is, by constructing a library whose taxonomy is the union of the software defect taxonomy presented in Section 4.2 and the one described in this section. We can relate processes with defects by adding attributes to both processes and failures. The `process` attribute for failures is the process in which the failure occurs; the `failure_set` attribute for processes describes the set of known failures.

```
  attribute process : ProjectPlan;
  assertion has process => in failures;

  attribute failure_set : set of failures;
  assertion has failure_set => in ProjectPlan &
                               failure_set=set(process=self);
```

Assume we want to predict the kinds of failures that may be associated with the process model `Design_project_2` described in our previous example. The following query retrieves processes that are similar to this process.

```
query imports={ReqDoc,DesDoc,DesGrp,Design} &
      objects={rdoc,ddoc,dgrp,dprc} &
      relations={rel_2}
```

Assuming that similar processes will have similar failures, we can use the failures of the retrieved process to predict the failures that may occur in the process `Design_project_2`.

4.4 Summary

This chapter has shown that EDF can be applied to a wide range of different software domains by presenting sample EDF taxonomies for: data structure packages, components for Command, Control, and Information Systems, software defects, GQM object evaluation models, and software process models.

The emphasis has been placed on the representation of different types of objects and their relations. Chapter 5 describes the process of designing and using similarity distances for retrieving objects from a reuse library. This is done by constructing a representation and similarity models for the NASA SEL database. This database contains hundreds of projects, systems, components, as well as recorded history of their reuse.

Chapter 5

Case Study: The EDF NASA library

This chapter presents a detailed description of the development of an EDF software reuse library. The contents and structure of this library are based on data extracted from the NASA Software Engineering Laboratory (SEL) database, which is used to store information regarding multiple aspects of the development of applications at NASA Goddard Space Flight Center.

The NASA SEL database contains all the necessary information required for the construction of an EDF reuse library. It provides great quantities of software engineering data such as descriptions of Ada and FORTRAN applications, their subsystems and components, as well as statistical information related to the construction of these applications.

One important aspect of the NASA SEL database is that it provides information regarding the reuse of application components. This information is called the Lineage Reuse Data (LRD), and describes the *origin* of components, as well as the amount of *effort* involved in their construction. The LRD is important because it helps design and construct similarity distance comparators, which are essential elements of EDF's similarity-based retrieval mechanism.

This chapter is organized as follows. Section 5.1 describes the contents and structure of the NASA SEL database, as well as the Lineage Reuse Data. Section 5.2 describes the EDF representation model used to characterize the database and also explains how to design distance comparators based on the LRD. Finally, Section 5.3 provides an evaluation of EDF's retrieval mechanism based on a set of tests performed using the resulting EDF NASA library.

5.1 The NASA SEL database

The Software Engineering Laboratory (SEL) was established in 1976 to support research in the measurement and evaluation of the software development process. Under its sponsorship, numerous experiments have been designed and executed to study the effects of applying various tools, methodologies, and models to software development efforts in flight dynamics applications. One of the major functions of the SEL is the collection of detailed software engineering data, and the archival of this information for future use. The SEL has created and maintained an online database for the storage and retrieval of software engineering data. This database was designed and implemented at NASA Goddard Space Flight Center using the ORACLE Relational Database Management System.

5.1.1 Overview of the Database

The fundamental entity of the NASA SEL database is called *project*. Project data compose the bulk of the information in the SEL database. A small portion of the database is also allocated to the storage of support data, such as computer, services, and personnel names.

Project data is logically divided into several entities, each associated with a different aspect of its development. Each project has an associated record on which basic information such as the project's name, type, development status, and identification code number is stored. Project data that become available during the different development phases of the project, are recorded in separate entity records (see below). For further details, refer to the "Software Engineering Laboratory (SEL) Database Organization and User's Guide" [68].

Schedules Project schedules divide the lifespan of a project into a series of non overlapping, contiguous time periods referred to by the SEL as phases. A phase corresponds closely to the primary type of development activity being performed at any given time.

Estimates At various points in the life of a project, estimates are made of certain project characteristics whose actual values do not become available until the end of development phases. They are made as part of the process of planning a project and monitoring its progress.

Resource Usage During the development stage of a project, the use of personnel and computer resources is measured and recorded on a weekly basis. Personnel resources are measured in terms of the number of hours spent working on a list of predefined activities.

Product Characteristics These describe the software product that is being generated during the development stage. There are two primary types of product data: those which capture the static composition of the system (i.e., the attributes and relations of the different subsystems and components of a project), and those which capture the dynamic properties of system growth and change.

Changes A change is viewed by the SEL as an update to one or more system components for a particular specific purpose. Typical purposes for changes include correcting an error, improving the efficiency of an operation, or implementing an enhancement.

Subjective Evaluations When a project completes its development stage, the retrospective subjective evaluations of personnel involved in the management of the project are collected and stored in the database. This information is later used as a basis to improve future development efforts.

Final Statistics When the development stage of a project is complete, measurements are recorded of the actual values of parameters that were estimated earlier and of additional parameters that were not estimated. These include, among others, total number of lines of code, number of pages of documentation, number of subsystems and components, etc.

Development Status The status of active projects is monitored throughout their development and recorded on a weekly basis. There are two types of status data: target data and measurement data. Target data represent the goals or target values (e.g., number of modules to be designed). The measurement data represent values measured during the process (e.g., number of modules that were actually designed at weeks end).

5.1.2 The Lineage Reuse Data

Project components in the NASA SEL database have one data field designed for the purpose of recording reuse information. This field is called the *origin* of the component, and it is used to store a code that indicates whether the component was constructed from scratch or by reusing parts of another component. These codes are assigned by the programmer that implements the component, and give a measure of the amount of code that was reused. The codes are the following.

OLDUC: indicates the component was constructed by copying another component with little or no modification. Both components are almost identical in structure and function.

SLMOD: indicates the component was constructed by slightly modifying another component. Both components share most of their code, but the structure and/or functionality may have been changed.

EXMOD: indicates the component was constructed by extensively modifying another component. Both components have very little code in common, and their structure and/or functionality may be very different.

NEW: indicates the component was created from scratch. The component is original, and shares no code with other components.

Codes `OLDUC`, `SLMOD`, and `EXMOD` define an implicit relation between two components: the one whose code was reused (the *source*) and that which was constructed using this code (the *target*).

Rush Kester at NASA performed a study [57] in which component source code files were examined to determine their origin and the amounts of reused code. The data collected from this study, combined with the reuse information in the SEL database and design documents, enabled Kester to trace the evolution of components of 9 different Ada projects. Part of the outcome of this study is called the *Lineage Reuse Data*, which consists of a list of approximately 2000 triplets (S, T, C), indicating that component T was constructed by reusing code from component S, with an amount of effort given by the *origin* code C.

The Lineage Reuse Data considers three types of components: those that were developed using components from other projects; those that were acquired from external sources to NASA; and those that were adapted from similar components within the same project. Only those components developed at NASA have their data stored in the NASA SEL database.

5.2 Characterization of the NASA database

The EDF NASA library was constructed by selecting a subset of the entities that compose the NASA SEL database. This subset includes the entities: *project*, *subsystem*, and *component*. The relations among these selected entities were preserved, but not all the record data fields they possess in the NASA SEL database were used. Our goal in constructing this EDF library was to show that EDF is capable of representing the type of information and relations contained in this database, and not to duplicate the database[1].

5.2.1 Representation Model

Figure 5.1 shows the relations among the entities of the EDF SEL Library. Single-arrow connections represent one-to-one relations, while double-arrows represent one-to-many relations. For example, the fact that a project is associated with an unbounded number of subsystems is represented by an arc connecting project to subsystem with a double-arrow. On the other hand, a subsystem is always associated to one project. This is represented by an arc connecting subsystem to project with a single arrow.

Each of the entities shown in Figure 5.1 represents an EDF class, which is characterized by a set of attributes. In most cases, these attributes have a one-to-one correspondence to fields used in the NASA SEL database, yet their names were changed for easy reading[2]. Attribute values are defined in terms of EDF type names, which refer to either numbers, enumerations, classes, or sets of the above. To help distinguish these types, enumeration names are written in *italics*, class names are written in **bold**, and the rest are in plain format.

[1]The mere size of the database would make this task almost impossible given the time and resources available for this research.

[2]Field names in the NASA SEL database are constrained to a maximum number of characters, which makes some of them very cryptic.

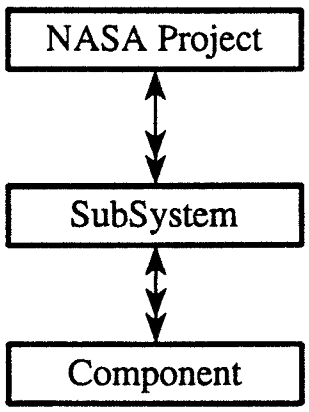


Figure 5.1: Logical view of the EDF NASA library

Project Class

The attributes used to characterize a project are described in Table 5.1. The attributes in the first block are used to characterize all projects in the database regardless of whether they have been completed or they are still being developed. Attributes in the second block define the final statistics for projects that have been completed.

Attribute	Type	Description
`proStatus`	*proStatus_t*	Project status
`proType`	*proType_t*	Project type
`proSubsystems`	set of **subsystem**	List of subsystems
`proSubscnt`	number	Number of subsystems
`proCompcnt`	number	Number of components
`proChanges`	number	Number of changes
`proDocumSize`	number	Number of pages of documentation
`proTechnical`	number	Number of technical hours
`proService`	number	Number of service hours
`proComments`	number	Number of comment lines
`proCodeLines`	number	Number of lines of code
`proExeModules`	number	Number of executable modules
`proStatements`	number	Number of statements
`proExeStmts`	number	Number of executable statements

Table 5.1: Attributes of the project class

Project data are collected only while a project is either under development or in maintenance. The status of a project is represented by the attribute `proStatus` as one of the alternatives in Table 5.2.

Term Name	Description
act_dev	Project is in development phase
act_maint	Project is in maintenance phase
discont	Data collection discontinued
inactive	Project has been completed

Table 5.2: Status codes of a project

Projects at NASA are all related in one way or another to solve problems related to the area of flight dynamics. The attribute **proType** is used to distinguish these different types of problem solutions based on the alternatives presented in Table 5.3. Projects with no defined area are characterized as otherProType.

Term Name	Description
agss	Ground support system
attitude	Attribute oriented
database	Database system
graphic	Graphic/User interface
mpa	Mission planning and analysis
orbit	Orbit orientation
realtimep	Real time processing
scientific	Scientific oriented
simulator	Simulator
tool	Software tool
otherProType	Other project type

Table 5.3: Type codes of a project

The attribute **proSubsystems** is used to characterize each subsystem of the project. Each element of this set must belong to the class **subsystem**, explained in the following sections.

Subsystem Class

The composition of the project is recorded as the project evolves. Projects are partitioned into subsystems and components, along with descriptive information about each. The attributes that define a subsystem are given in Table 5.4.

A subsystem defines a group of components (e.g., FORTRAN subroutines, or Ada functions) whose purpose is to solve a particular problem or provide some kind of service. The attribute subFunction differentiates subsystems based on the functionality they provide according to the alternatives presented in Table 5.5.

Subsystems belong to one particular project. The attribute subProject defines the subsystem's project. Its value must be an instance of the class **project**,

Attribute	**Type**	**Description**
`subFunction`	*subFunction_t*	Subsystem function
`subProject`	**project**	Subsystem's project
`subComponents`	set of **component**	List of subsystem components

Table 5.4: Attributes of the subsystem class

Term Name	**Description**
`cpexec`	Control processing/executive
`dpdc`	Data processing/data conversion
`graph`	Graphics and special device support
`mathcomp`	Mathematical/computational
`realtime`	Real-time control
`sysserv`	System services
`userint`	User interface

Table 5.5: Function codes of a subsystem

defined on page 82. The list of components that form a subsystem is defined using the attribute **`subComponents`**. Each element of this list must be an instance of the class **component**, defined in the following section.

Component Class

Subsystems define a group of one or more components that perform a particular function. Components can be active elements, such as subroutines, or they can be passive elements, such as FORTRAN data blocks. The attributes that define a component are given in Table 5.6.

Attribute	**Type**	**Description**
`comType`	*comType_t*	Component type
`comPurpose`	set of *comPurpose_t*	Component purposes
`comSubsystem`	**subsystem**	Component's subsystem
`comTotalLines`	number	Number of source lines
`comExeStmts`	number	Number of executable statements
`comComLines`	number	Number of comment lines
`comStmtLines`	number	Number of statement lines

Table 5.6: Attributes of the component class

Components are implemented either from scratch or by reusing the code of some old component previously implemented. In the latter case, we can differentiate between three cases depending on how the old component was reused: (1) reused with no modification, (2) reused with small modifications, or (3) reused with extensive modifications. Attributes `comType` and `comPurpose` are both used to capture the nature of the component in terms of its function and type of source code. The attribute `comPurpose` defines the functionality in terms of a list of keywords representing the main operations performed by the component. These keywords are shown in Table 5.7.

Term Name	**Description**
`adada`	Ada data abstraction
`adapr`	Ada process abstraction
`alcomp`	Algorithmic/computational
`cntrmod`	Control module
`datra`	Data transfer
`intop`	Interface to operating system
`iopro`	I/O processing
`lodec`	Logic/decision

Table 5.7: Purpose codes of a component

The attribute `comType` defines the source language and structure used to implement the component as one of the alternatives presented in Table 5.8.

The attribute `comSubsystem` defines the component's subsystem. Its associated value must be an instance of the class **subsystem** defined on page 83.

Contents of the EDF NASA library

The EDF NASA library contains the description of only those components mentioned in the Lineage Reuse Data, their associated subsystems, and projects. Certain component, subsystem, and project descriptions were not included in the library due to lack of information and other factors. To be included, descriptions had to satisfy the following restrictions.

Components: Components in the NASA SEL database usually have values for all their associated attributes. A small group had part of their data missing from the database. To avoid using partially described components, all those components whose attributes `comExeStmts`, `comComLines`, `comStmtLines`, and `comPurpose` were missing values at the same time were removed. If any of these attributes had a value, the component was included in the library.

Subsystems: Only those subsystems of components mentioned in the Lineage Reuse Data were considered for the EDF library. Those that did not have a value for the attribute `subFunction` were excluded from library. Subsystems that did not have any components in the library (i.e, with an empty

Term Name	Description
adagenb	Ada generic body
adagens	Ada generic specification
adapackb	Ada package body
adapacks	Ada package specification
adasubb	Ada subprogram body
adasubs	Ada subprogram specification
adataskb	Ada task body
adatasks	Ada task specification
adaunspec	Unspecified Ada source code
alc	Assembly language
blockda	Fortran BLOCKDATA
display	Display identification
fortran	Fortran source code
incl	Include header file
jcl	Job control language
mendef	Menu definition/help file
namelt	NAMELIST/parameter list
pascal	Pascal source code
refdata	Reference data file
otherComType	Other component type

Table 5.8: Type codes of a component

subComponents list) were also excluded from the library. This situation arises when all subsystem components are eliminated because none satisfies the restrictions imposed for components.

Projects: Only those projects of components mentioned in the Lineage Reuse Data were considered for the EDF library. Those that did not have any subsystems (i.e., with an empty proSubsystems list) were excluded from the library. This situation arises when all project subsystems are eliminated because none satisfies the restrictions imposed for subsystems.

After excluding all those elements that did not satisfy the above restrictions, the resulting EDF NASA library contained descriptions for 1998 components, 31 subsystems, and 8 projects. The number of pairs in the Lineage Reuse Data that had both components included in the library was 1178. Of these, 736 triplets were OLDUC, 352 were SLMOD, and 90 were EXMOD.

5.2.2 Similarity Distances

So far, I have described the EDF representation model used to construct the EDF NASA library. In order to complete the implementation of this reuse library it is necessary to assign *distance comparators* to each of the attributes of the different classes of this model. Without these comparators, EDF's similarity-based retrieval mechanism could not be used to find suitable reuse candidates in the library, and a reuser would be limited to using queries based only on exact matches.

Relevance Factors

In general, the distance between two EDF objects is given by the sum of the distances between their corresponding attributes. This default scheme gives equal importance to all attributes. In our particular situation, this is not a reasonable assumption. For example, one would consider that differences between component subsystems (i.e., difference between the values of their `comSubsystem` attributes) is more important than the difference between their number of lines of source code (i.e., `comTotalLines`). Therefore, the first step required to design distance comparators is to assign a *relevance factor* to each attribute in the representation model, that is, to define the amount of influence they have in the computation of similarity distances.

To understand how relevance factors were assigned, it is important to remember how components, subsystems, and projects are related. Each component description references its subsystem via the attribute `comSubsystem`, and each subsystem references its associated project via the attribute `subProject`. The overall distance between two components, A and B, is given by the difference between the values of their corresponding attributes. Given that one of these attributes is `comSubsystem`, then this distance will also include the difference between the subsystem of A and B. Similarly, one of the attributes of subsystems is `subProject`, therefore the distance between the components will also include the difference between their associated projects.

The attributes `subComponents` and `proSubsystems` require special attention because they involve recursive distance computations. Although EDF was designed and implemented to handle these kinds of recursive distance computations, the size of the EDF NASA library, combined with EDF's current implementation, makes this kind of computation too slow. To avoid this situation, these attributes were assigned null distance comparators (i.e., they always return a distance of zero). The effect of this is that distances between component descriptions include their own subsystems and projects, but do not consider other subsystems or components.

Table 5.9 shows the relevance factors used in the EDF NASA library. Each value falls in the range 0 to 1, and the sum of the factors within each class is 1, with bigger factors indicating more relevance. These values were assigned using certain rules based primarily on intuition and experience[3]. In particular, attributes of the same type in a class were assigned equal relevance factors, so, for example, all numeric component attributes contribute 6.25% to the overall distance, while each enumeration contributes 25%. Numeric attributes were assigned smaller relevance factors, but their combined contribution is equal or larger than those of other attributes.

Class Name	Attribute Name	Relevance Factor
Component	`comType`	0.25
	`comPurpose`	0.25
	`comSubsystem`	0.25
	`comTotalLines`	0.0625
	`comExeStmts`	0.0625
	`comComLines`	0.0625
	`comStmtLines`	0.0625
Subsystem	`subFunction`	0.50
	`subProject`	0.50
Project	`proStatus`	0.25
	`proType`	0.25
	`proSubscnt`	0.0454
	`proCompcnt`	0.0454
	`proChanges`	0.0454
	`proDocumSize`	0.0454
	`proTechnical`	0.0454
	`proService`	0.0454
	`proComments`	0.0454
	`proCodeLines`	0.0454
	`proExeModules`	0.0454
	`proStatements`	0.0454
	`proExeStmts`	0.0454

Table 5.9: Relevance factors of attributes

Subsystems and projects are important because they define the component's environment, yet they do not directly characterize a component's functionality. For this reason, their contributions were defined to be relatively small by assigning attributes `comSubsystem` and `subProject` relevance factors of 25% and 50% respectively. Note that the relevance factor of 50% for `subProject` affects distances

[3]This ad hoc procedure is weak and should be subjected to further research (see Chapter 6). Nonetheless, as we will see in Section 5.3, the values that were chosen were more than adequate for our needs.

between subsystems, therefore its influence on component distances is only 12.5% (50% of 25%).

Distances and the Lineage Reuse Data

Similarity distances are non-negative magnitudes proportional to the amount of effort required to transform one component into another. EDF retrieves candidate reuse components for a given target component T by first computing distances from each component description in the library to T, and then selecting those with the smaller values. Transformation efforts ought to be defined based on an analysis of these objects, but in our case there is no access to their source code nor to any documentation describing their specific functionalities[4]. Nonetheless, the Lineage Reuse Data (LRD) provides indirect information regarding the transformation efforts between certain pairs of components.

As explained earlier, the LRD is basically a list of triplets (S, T, C), where T and S are components, and C is one of three codes: `OLDUC`, `SLMOD`, and `EXMOD`. Each triplet indicates that component T was constructed at NASA using component S, with an amount of effort given by C. We will use this information to assign distance magnitudes based on the following two assumptions: (1) each source component S was selected because it was the best reuse candidate available at the time for constructing its corresponding target component T, and (2) the amount of effort required for this transformation is properly characterized by code C. If these assumptions are correct, then the LRD codes `OLDUC`, `SLMOD`, and `EXMOD` must each represent an average transformation effort measured in terms of some particular cost unit (e.g., number of man-hours).

Let D_o, D_s, and D_e be the average transformation magnitudes associated with codes `OLDUC`, `SLMOD`, and `EXMOD`, respectively. The semantics of these codes imply the following relation between the magnitudes: $0 < D_o < D_s < D_e$. Figure 5.2 presents a summary of our analysis regarding transformation efforts (distances) and their relation to the codes of the LRD triplets.

Distance Comparators

EDF computes the distance between two object descriptions as a weighted sum of the differences between their respective attribute values. The weights are defined by the relevance factors associated with each attribute, and the differences are quantified by non-negative magnitudes computed by distance comparators. As explained in Section 3.2.2, each attribute has three comparators: one for transformation, one for constructing, and one for removal. The last two are required when comparing objects with different sets of attributes.

Distance comparators were defined using the information contained in the LRD in combination with the attribute values of the components of the EDF NASA library. The general idea is the following: if $(S, T, C) \in$ LRD, the distance

[4]Even if we had access to the code, it would take an enormous amount of time to examine the approximate 2000 components that compose the EDF NASA library.

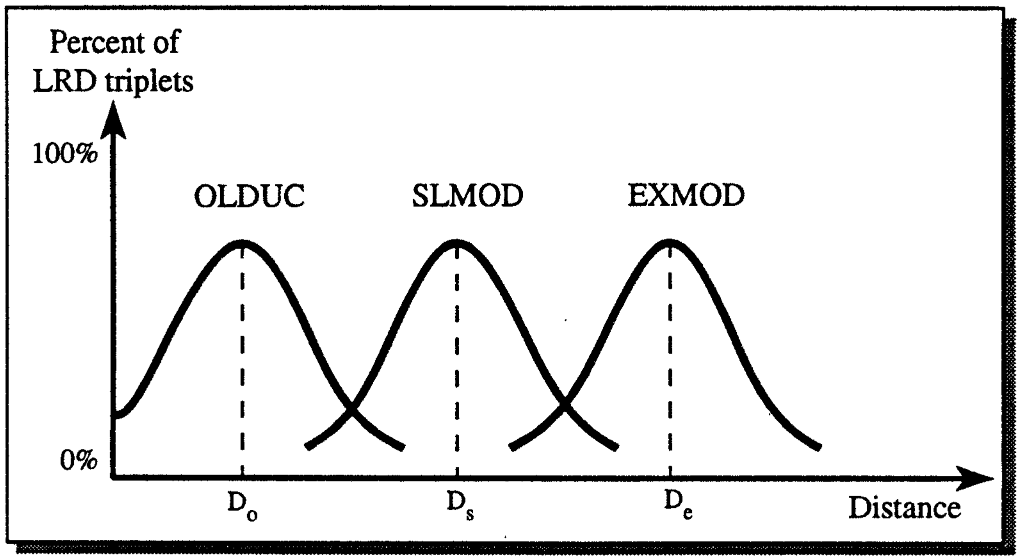


Figure 5.2: Distribution of distances for the Lineage Reuse Data

from S to T should be, on average, equal to the magnitude D_C assigned to C (i.e., D_o, D_s,or D_e). Given that this distance is computed as a weighted sum of the distances between the corresponding attributes of S and T, each attribute A should contribute, on average, with a magnitude equal to $R_A D_C$, where R_A is the attribute's relevance factor.

Specifically, let A be an attribute and $V_A = \{v_1, \ldots, v_n\}$ be the set of values that A can have. Also, let $(S, T, \mathtt{OLDUC}) \in$ LRD such that $S.A = v_s$[5] and $T.A = v_t$. If F_A is A's transformation comparator, then $F_A(v_s, v_t)$ should be equal to $R_A D_o$. Nonetheless, the same pair of values, v_s and v_t, may be used in the library to describe components that are not OLDUC but are, say, SLMOD. This would imply that $F_A(v_s, v_t)$ should be equal to $R_A D_s$ instead. The solution to this problem depended on the type of values of attribute A. If the values of A are enumeration terms, F_A returns the following average distance magnitude.

$$F_A(v_s, v_t) \quad = \quad R_A \frac{n_o(v_s, v_t)D_o + n_s(v_s, v_t)D_s + n_e(v_s, v_t)D_e}{n_o(v_s, v_t) + n_s(v_s, v_t) + n_e(v_s, v_t)} \tag{5.1}$$

where $n_o(v_s, v_t)$ is the number of times a triplet (S, T, C) in the LRD satisfies the following condition: $S.A = v_s$, $T.A = v_t$, and $C = \mathtt{OLDUC}$. Similarly for the parameters $n_s(v_s, v_t)$ and $n_e(v_s, v_t)$. When the values of the attribute A are numbers, F_A returns a distance magnitude proportional to the different between the values of v_s and v_t.

$$F_a(v_s, v_t) \quad = \quad R_A D_e \frac{|v_s - v_t|}{\max(v_s, v_t)} \tag{5.2}$$

[5] $P.A$ denotes the value of the attribute A of component P.

This equation returns a value between 0 and $R_A D_e$[6], and is based on the following two assumptions: (1) the bigger the absolute difference between two numeric attributes, the less similar their corresponding components are, and (2) this difference is limited to that of extensively modified components (i.e., EXMOD).

Given that the types of all attributes used in the representation model of the EDF NASA library are either enumerative or numeric, the equations 5.1 and 5.2 allows us to define all required distance comparators. For example, the actual definition of the attribute comPurpose is as follows:

```
attribute comPurpose :
  {
  adada,adapr,alcomp,cntrmod,datra,intop,iopro,lodec
  }
distance
  {
  ->adada  :732,adada->:851,datra:398,lodec:500,iopro:205,
            alcomp:669,adapr:493,intop:0,cntrmod:1000,
  ->adapr  :628,adapr->:0,intop:0,alcomp:553,adada:804,
            datra:491,iopro:499,lodec:800,cntrmod:831,
  ->alcomp :0,alcomp->cntrmod:546,lodec:730,iopro:266,
            intop:0,datra:723,adapr:686,adada:580,
  ->cntrmod:1000,cntrmod->iopro:205,adada:0,adapr:0,
            datra:194,alcomp:0,lodec:0,
  ->datra  :0,datra->:943,adada:728,lodec:824,iopro:652,
            alcomp:734,cntrmod:851,intop:0,adapr:473,
  ->intop  :1000,intop->adapr:0,adada:0,
  ->iopro  :1000,iopro->intop:0,adapr:424,adada:766,
            alcomp:318,cntrmod:500,datra:256,lodec:784,
  ->lodec  :1000,lodec->adada:908,alcomp:614,cntrmod:256,
            datra:591,iopro:854,intop:0,adapr:414
  };
```

The distances between the terms of the attribute comPurpose were all computed using equation 5.1, and rounded to the closest integer value. Numeric attributes, on the other hand, were defined in terms of foreign distance comparators. For example, the actual definition of the attribute comTotalLines is the following:

```
attribute comTotalLines :
  number
distance
  {
  s -> t :
      {
      double svalue = EDF_number(s);
      double tvalue = EDF_number(t);
      double factor = abs(svalue - tvalue) / max(svalue,tvalue);
      RETURN(R_comTotalLines * D_EXMOD * factor);
      }
  };
```

where R_comTotalLines and D_EXMOD are globally defined constants that represent the values R_A and D_e in equation 5.2, and EDF_number is one of the built-in support functions provided by EDF to obtain the value of an argument. For further details on foreign distance comparators refer to Appendix A.

[6]For any pair of positive values, x and y, the value of the expression $\frac{|x-y|}{\max(x,y)}$ is always between 0 and 1.

5.3 Evaluation and Results

In order to verify the representation model and distance comparators that define the EDF NASA library, it is necessary to test whether EDF is capable of properly predicting similarity distances among components in the library. Since the only information available regarding similarities is given by the Lineage Reuse Data, a test was performed to check whether the distances computed by EDF from source to target components in the LRD coincide with the analysis of distributions presented on page 89 and summarized on Figure 5.2.

The test performed is depicted on Figure 5.3. For each triplet (S, T, C) in the LRD, EDF was used to compute the list of candidate reuse components for T. This list contains all the components in the EDF NASA library sorted by increasing distance (decreasing similarity) to T. Upon completion, component S was searched in the list, and both its associated distance and relative position in the list were recorded. In this particular test, only the distance value is needed. The use of the position value is explained later in this section.

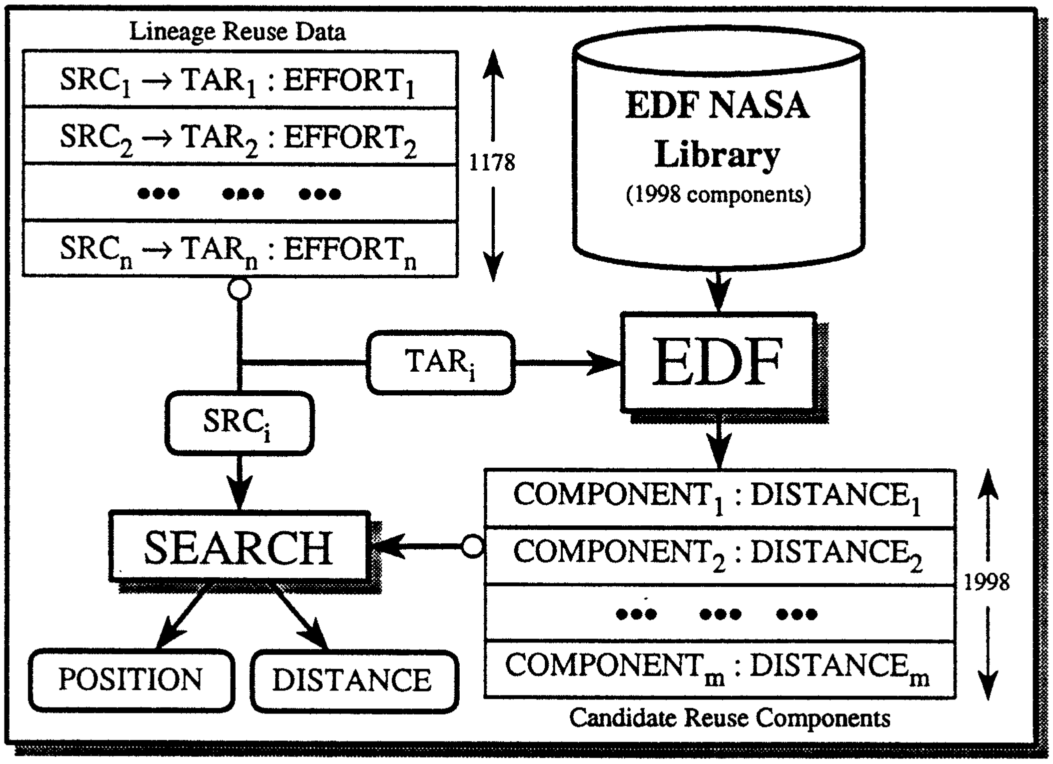


Figure 5.3: Process for obtaining distance and position values

As explained earlier, the EDF NASA library contains all the components mentioned in the LRD, therefore S will always be found in the list of candidate reuse components produced by EDF. Also, component T will have an associated distance of zero, and hence it will always be found in the first position of the list. Component S, on the other hand, may appear anywhere in the list with some arbitrary distance magnitude.

The values used for the average transformation efforts D_o, D_s, and D_e were defined as 100, 1000, and 2000, respectively. As explained earlier, these values plus the relevance factors shown in Table 5.9 define all the distance comparators that form the similarity model for the EDF NASA library. The statistics regarding the resulting similarity distances is shown in Table 5.10, while their distribution is shown in Figure 5.4.

Code	Average	StdDev	Minimum	Maximum
OLDUC	430	623	46	2310
SLMOD	791	973	4	3391
EXMOD	1703	1853	295	4000

Table 5.10: Statistics for distance values

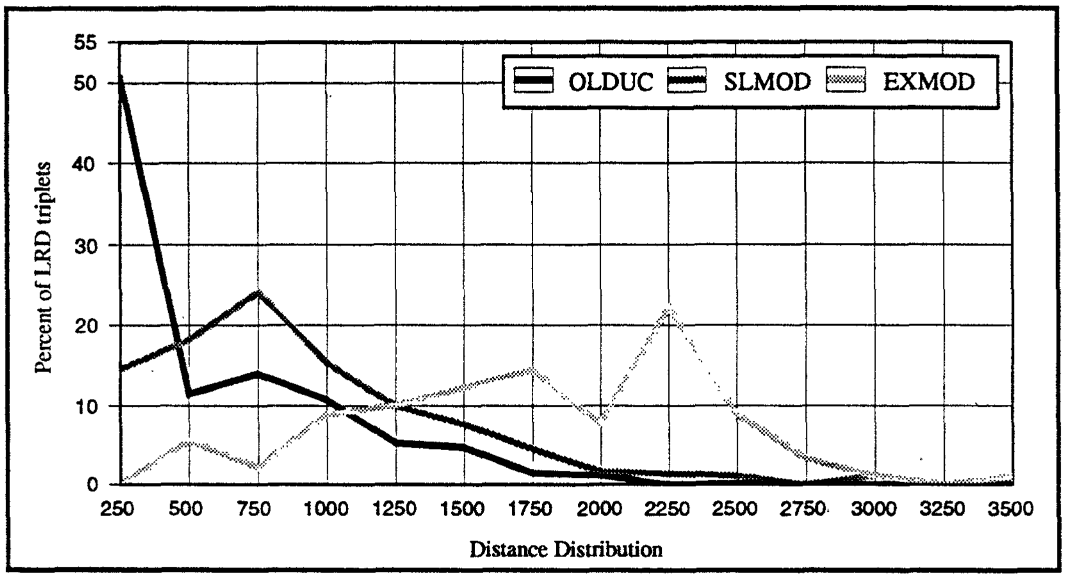


Figure 5.4: Distribution of distance values

These results confirm our expectations (see Figure 5.2) by showing a relation between the average similarity distance among components and the amount of code these components have in common. They also show that the deviation from this average increases as the amount of shared code decreases, that is, the more similar the components the less specific the distance becomes.

Nonetheless, the level of specificity is lower than desired, specially in the cases of SLMOD and EXMOD. Part of the reason is that the codes OLDUC, SLMOD, and EXMOD are assigned at NASA by programmers of different levels of expertise based on subjective evaluations. If two components are almost identical, we would expect all programmers to correctly assign a code of OLDUC. This is not so clear in the cases of SLMOD and EXMOD. Their semantics are not precise, so programmers with more experience would tend to assign code SLMOD to reuse tasks that programmers with less experience would otherwise classify as EXMOD.

5.3.1 Predicting Reuse Candidates

One of the basic assumptions about the Lineage Reuse Data is that each source component was selected to construct its associated target component because it was judged at the time to be the best reuse candidate available at NASA for this task. Our hypothesis is that these pairs can be predicted using EDF. That is, given the description of a target component T, its associated source component S in the LRD will be chosen by EDF as its best reuse candidate.

As explained earlier, EDF was used to compute the list of reuse candidates for each target component in the LRD and the position of its associated source component in this list was recorded (see Figure 5.3). Table 5.11 shows the statistics of positions values, and Figure 5.5 shows their distribution. Position values have been normalized so the maximum value (i.e., 1998) is mapped to one hundred percent (100%).

As we can see in Figure 5.5, EDF's retrieval mechanism was able to correctly predict over 60% of OLDUC type triplets. In addition, over 87% of the source components of this type of triplets could be found within the first 10 positions of the list of candidate reuse components. That is, the selection of reuse components performed at NASA and the one performed by EDF were very similar when the amount of transformation was small.

Code	Average	StdDev	Minimum	Maximum
OLDUC	86	211	1	1837
SLMOD	268	363	1	1917
EXMOD	758	548	1	1867

Table 5.11: Statistics for position values

The results for triplets of type SLMOD and EXMOD are not as clear. The distribution for SLMOD triplets shows a strong tendency towards smaller position values, while that of EXMOD triplets is almost uniform across the entire range. These results would imply that the quality of the predictions of EDF decreases as the differences between the source and its target component increases.

An alternative explanation for this behaviour is the following: when computing the position of a source component S in the list of reuse candidates for a target component T, EDF considered all the components available in the library as valid reuse candidates, and not just those that were actually available at NASA at the time T was constructed. What the results would be showing in the case of SLMOD and EXMOD is that the library contains components that were probably developed after T which make better reuse candidates than S. If this is the case, then the position of these components in the list of reuse candidates of T would

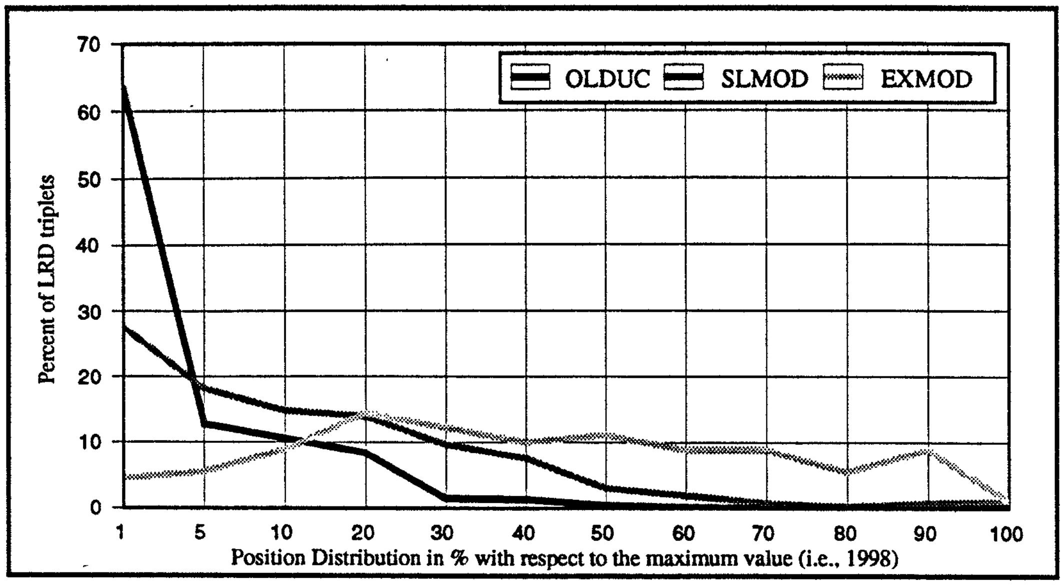


Figure 5.5: Distribution of position values

be smaller than that of S, and therefore we would obtain results like the ones shown in Figure 5.5.

In order to verify this hypothesis, we would need to know the dates of development of each component at NASA. The only relevant dates available in the NASA SEL database are those that indicate the starting and ending dates of the different phases of a project, and those that indicate when a component description was entered in the database. This information does not allows us to determine the dates we require for two reasons. (1) The date on which a component is entered in the database may or may not be related to the time the component was developed. Their descriptions are initially recorded on paper forms, and are entered in the database at random times thereafter. (2) An analysis of the number of hours spent on development shows that this activity took place during all phases of the projects contained in the EDF NASA library. That is, components could have been developed at any time during a project's life cycle.

It has been suggested that one method to approximate the date of a component is to use either the starting or ending date of its associated project. Unfortunately, most of the projects in the EDF NASA library were developed in parallel, and their starting and ending dates are approximately the same.

5.4 Summary

This chapter describes the different steps that were required to develop the EDF NASA library, a reuse library based on information extracted from the NASA SEL database. This database contains great quantities of software engineering data such as descriptions of Ada and FORTRAN applications, their subsystems and components. In addition, it contains information called the Lineage Reuse Data (LRD) that provides an indirect measure of the degree of similarity of certain pairs of components in the database.

The emphasis on this chapter was placed on explaining the process of designing, implementing, and verifying similarity models. In particular, the attributes that compose the representation model of the EDF NASA library were assigned relevance factors, as well as distance comparators designed using the similarity measures provided by NASA's Lineage Reuse Data.

The resulting similarity model was tested by computing similarity distances between components in the EDF NASA library, and comparing the resulting values with the similarity measures provided by the LRD. In addition, an experiment was performed to check whether the reuse candidates selected by EDF matched those selected at NASA. The results of this experiment were not conclusive when the amount of code shared by a candidate and its target component was small. On the other hand, EDF performed adequately when the two components had a significant amount of code in common.

Chapter 6

Conclusion

In this dissertation I have proposed the Extensible Description Formalism (EDF), a system for designing, building, using, and maintaining software reuse libraries (i.e., repositories of reusable software assets such as code, documentation, processes, tools, etc.). I have argued that software reuse libraries and the systems that manage them (i.e., software reuse library systems) are important components of almost all methods and tools aimed at improving reusability in software development organizations. In particular, these systems provide a cost effective alternative to improve reusability in organizations where large amounts of software components have already been developed but no formal methods have been used to structure and store said components for future use.

The Extensible Description Formalism is based on a generalization of the faceted index approach proposed by Prieto-Díaz [75], which has proven to be an effective tool for creating software reuse libraries. The faceted index approach relies on a predefined set of *facets* defined by experts. These facets and their associated *terms* form a classification scheme for describing components. Although useful, the faceted index approach has a very limited representation model and a poor retrieval mechanism, both of which severely restrict its usefulness in software reuse library system.

EDF overcomes part of the limitations of current faceted system by extending the their representation model. Two main concepts form the core of EDF's representation model: *instances* and *classes*. Instances are descriptions of reusable objects, while classes represent collections of instances with a set of common properties. Objects are described in terms of *attributes* and associated values. Unlike faceted classification which is limited to having only terms as attribute (facet) values, EDF allows attribute values to be instances and even sets of instances. This generalization can be used to create one-to-one, one-to-many, and many-to-many relations between different object classes within a library. In other words, EDF's specification language is powerful enough to represent a wide variety of software (and non-software) domains, ranging from standard software components such as data structure packages and their operations, to more complex domains such as software defects and software process models (see Chapter 4).

The other way in which EDF overcomes the limitations of faceted systems is by providing a similarity-based retrieval mechanism based on asymmetric distances.

These distances are used by EDF to select candidate reuse components for a given target component description. Similarity is quantified as a non-negative magnitude called *similarity distance*, which estimates the amount of effort required to obtain the target given a candidate component. Distances can be computed using either built-in or user-supplied distance functions. I have shown the applicability of EDF's similarity model by constructing a library containing part of the information of the NASA Goddard Space Flight Center database for satellite projects, and designing and implementing distance comparators based on actual reuse data at NASA (see Chapter 5).

A prototype system that implements the representation and similarity models of the Extensible Description Formalism has been implemented, and is available in a wide range of computer platforms (see Appendix A). This prototype defines algorithms for the computation of similarity distances as well as solutions for complex interdependencies that can appear in library definitions. It also provides a preliminary user-interface definition for the system. The prototype system satisfies the following required characteristics of any reuse library systems [77]:

- It accommodates continually expanding collections, a characteristic of most software organizations.
- It supports finding components based on *similarity* considerations, and not just exact matches.
- It supports finding functionally equivalent components across domains.
- It is specific and has high descriptive power, which are both necessary conditions for classifying and cataloging software.
- It is easy to maintain, that is, adding, deleting, and updating the class structure and vocabulary can be done without reclassification.
- It is easily usable by both the librarian and end user.

In summary, this dissertation has presented a software reuse library system called EDF and shown how its representation and similarity models overcome the limitations of current reuse library systems based on faceted representations of objects.

6.1 Future Research Work

Although the software reuse system developed in this dissertation has proved to be an effective reuse tool, its performance and usefulness can be enhanced. I have identified several areas that need more research: (1) domain analysis, (2) semi-automatic classification, and (3) refinement of the similarity model.

Domain Analysis. In general, to create a library for software reuse it is necessary to perform a *Domain Analysis*, the process of identifying, collecting, organizing, analyzing, and representing a domain model and software architecture from the study of existing systems, underlying theory, emerging technology, and development histories within the domain of interest. Domain Analysis is currently done by human experts, but several proposals for formalizing and automating this process have been presented in the literature [53, 96].

In the case of an EDF software library, the process of Domain Analysis must define the classes and attributes used to describe software components (i.e., the classification model), and must also define the distance functions required to compute degrees of similarity. One particular product of the process of Domain Analysis is a *domain specific language*, that is, a language with syntax and semantics designed to represent all valid actions and objects in a particular domain. I believe the EDF language provides the representation power required for a domain specific language, but further research is needed in order to provide facilities for extending or adjusting its syntax and semantics to a particular domain.

Semi-Automatic Classification. A method is needed to classify components in terms of a given representation model. In general, this involves analysis of the different parts of a component (e.g., source code, documentation, etc.), and the use of heuristics to extract attributes based on this analysis. One approach that has proven successful has been proposed by Maarek *et al.* [63], which uses the concept of *lexical affinities* to create classifications of text documents. An alternative approach is to predict an object description based on the descriptions of its parts. For example, consider the following implementation of the function "minimum of a set of numbers".

```
min(numbers)=first-element(sort-ascending(numbers))
```

Function `min` is implemented by sorting in increasing order of magnitude the list of numbers it receives as argument, and then returning the first element of the sorted list. This implementation uses two subfunctions to accomplish its task, namely `first-element` and `sort-ascending`. If we assume we have access to the EDF descriptions of these two subfunctions, it may be possible to predict the EDF description of the overall function `min` using some kind of automatic or semi-automatic process.

Similarity Distances. A method is needed to test whether the reuse candidates proposed by the system are truly the best ones available in the software library. For example, if we classify a new component A known to be similar to a previously classified component B, we would expect the library system to propose B as a reuse candidate for A. Failure to do this could arise due to errors in classification of components A or B, or because of errors in the definition of relevance factors and/or distance comparators. If both A and B are classified correctly, then we need to adjust the factors and functions used to compute their degree of similarity (i.e., distance).

One approach for performing a semi-automatic refinement process of distance functions is based on feedback provided by the users of the software library. This process would work as follows: let T be the description of a test component, and E the description of a component in the software library known to be T's best reuse candidate. The EDF system is used to retrieve a proposed candidate P for T. If the distance from the proposed candidate (P) to the expected candidate (E) is greater than a certain value K, the descriptions of T, E, and P are used to adjust the relevance factors and/or comparator that define the similarity model. A value K of zero forces P to be equal to E. In general, K's value may need to be greater than zero to avoid undoing adjustments of the graphs done for other test components.

Appendix A

An EDF Prototype System

This appendix describes the main aspects of a prototype system that implements the syntactic and semantic specifications presented in Chapter 3. The EDF prototype system was programmed in ANSI C [56], and has been ported to several UNIX workstations including DECstations, Sun/3 and Sun/4, and NeXTstations. A Graphic User Interface (GUI) was programmed in Objective C [70, chapter 3] for a NeXT computer.

A.1 User Interface Module

The EDF prototype system has two versions of the Interface module. Section A.1.1 describes the command-driven interface version, and Section A.1.2 describes the graphic user interface version.

A.1.1 Command User Interface

The command-driven version of the EDF prototype system can be used with any regular terminal device in a UNIX environment. The shell command for activating EDF has the following format:

> **edf** *library-file-name*

where ">" is the shell prompt and "**edf**" is the program name of the command-driven version of EDF. The optional command argument "*library-file-name*" is the name of a regular text file containing EDF definitions. If specified, EDF will parse and load this file before accepting any commands.

The following list describes all valid EDF commands. Commands have specific formats, and they almost all start with a keyword which must be typed verbatim. Some require additional arguments. Those arguments enclosed in square brackets ([...]) are optional; alternatives are separated by a vertical bar (|) and enclosed within curly brackets ({...}); arguments shown in *italics* specify the general format and type of the argument.

- Command: *identifier*

  Unparses the internal memory representation of the EDF entity associated with the given *identifier* and displays its textual representation in the user's terminal.

- Command: `show` { *identifier* | *entity-type* }

  Allows the user to inspect the source definition of the given *identifier*. It activates the UNIX "vi" editor with the library file and line number position where the *identifier* was defined. In its alternative form, this command displays the names of all user-defined entity names of the type *entity-type*. The keyword *entity-type* must be one of the following: `attribute`, `class`, `value`, `instance`, or `type`.

- Command: `edit` { *string* | *identifier* }

  Allows the user to edit the library file specified in *string*. By default, the UNIX "vi" text editor is used. If the library file is changed, EDF will automatically reparse and check its contents. If no argument is given to this command, the last edited library file will be used. If an *identifier* is specified, the command edits the library file where the *identifier* was defined.

- Command: `query` *target-expression* [: *identifier*]

  Searches for *candidate* instance descriptions similar to the implicit *target* instance defined by *target-expression*. The output is a list of user-defined instance names sorted by increasing distance to the target expression. If the optional *identifier* is supplied, it must be the name of a user-defined *type* or *attribute*. In this case, distances between instances are computed using the comparators associated with *identifier*.

- Command: `focus` [*focus-expression*]

  Normally, the `query` command considers all user-defined instances as potential *candidates* to build the similarity list. The `focus` command is used to restrict `query` commands to those instances that satisfy the expression *focus-expression*. If this expression is "*", then all instances are selected. Finally, if no expression is specified, the `focus` command displays the current *focus-expression* and the list of selected instances.

- Command: `distance` *source-value* `->` *target-value* [: *identifier*]

  This command is used to compute estimated *removal*, *construction*, or *transformation* distance magnitudes according to the comparators associated with the entity *identifier*. If this optional argument is specified, it must be the name of a user-defined *type* or *attribute*. Both, *source-value* and *target-value*, must be names of user-defined values (e.g., instances or enumeration terms). If both values are given, the *transformation* distance from the source to the target is computed and displayed. If only the source value is specified, its associated *removal* distance is computed. Finally, if only the target value is specified, its associated *construction* distance is computed.

- Command: `quit`

  Terminates the execution of EDF and return to the shell program.

A.1.2 Graphic User Interface

The Graphic User Interface module[1] of the EDF prototype system provides the same functionalities as the command-driven interface module described in the previous section. Yet, because of its graphic nature and design, some of these operations are simplified and are made easier to use and understand.

Once the NeXT version of the EDF prototype has been activated, the user is presented with the menu of options shown in Figure A.1. This menu provides access to all the commands that are specific to the application such as the "File" and "Panels" submenus. It also provides access to standard commands shared by all NeXT applications such as the "Edit" and "Windows" submenus.

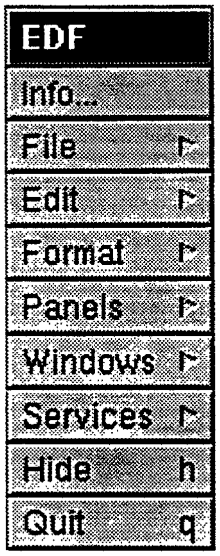


Figure A.1: The EDF menu

Normally, the first thing the user needs to do is to compile and load the definitions contained in an EDF library. This task is accomplished using the command "Load Library" found in the "File" submenu (see Figure A.1). If this command is selected, the user is asked for the name of an EDF library file. Once the file is compiled and loaded, a "Definitions" panel similar to that shown in Figure A.2 is activated.

[1]The Graphic User Interface module was developed in a NeXT computer using the "InterfaceBuilder" application [70] and the "Application Programming Interface" (API) [71].

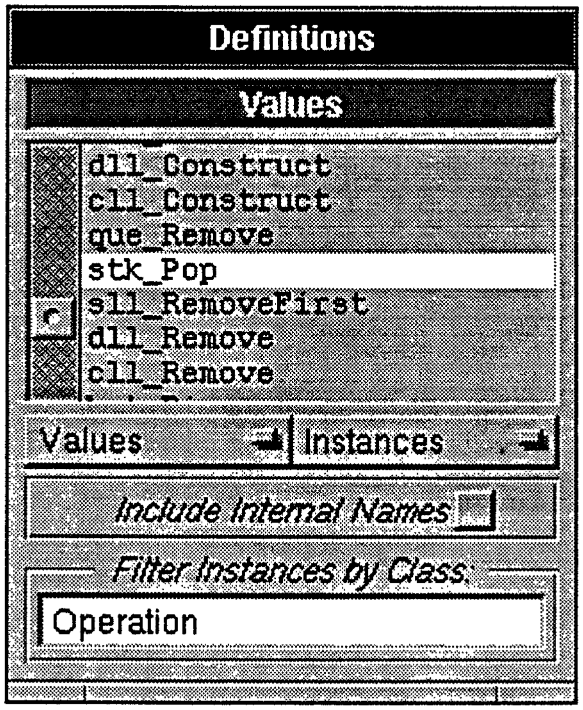


Figure A.2: The Definitions panel

The Definitions panel presents a list of user-defined library items of particular type. In this case, the list is composed of instances. The two pop-up menus underneath the list of names are used to select different item types: the leftmost pop-up menu selects the major item category (e.g., values, types, or attributes), and the rightmost selects the minor item category (e.g., instances, enumeration terms, or sets). By default, the panel shows only user-defined names; the "Include Internal Names" button can be selected to include the names generated by EDF for anonymous items. Finally, the field "Filter Instances by Class" at the bottom of the panel can be used to list only those instances that belong to a particular class (in this case, the class `Operation`).

The "File" submenu also allows the user to view the contents of an EDF library via the "Open Library" command. If this command is selected, the user is queried for the name of the library file and then a text window is activated. This text window allows the user to inspect the library by scrolling through its contents; no modifications to the text of the library are permitted.

Once an EDF library has been loaded, the user can start browsing its contents and searching for reuse candidates. All of this is accomplished via the "Panels" submenu shown in Figure A.3. The Panels submenu gives the user access to three panels specific to the EDF application: the Console, the Query, and the Inspect panels. The Console panel, shown in Figure A.4, allows the user to interact with the application using the commands described in Section A.1.1. Commands are written in the "Command" text field. Upon execution, the command and its output are appended to the end of the scrollable text field; commands are shown in bold face to differentiate them from their output.

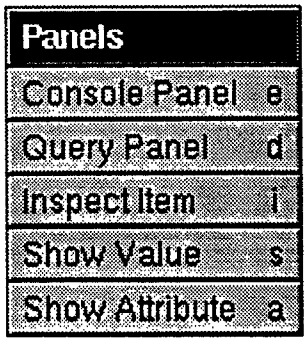
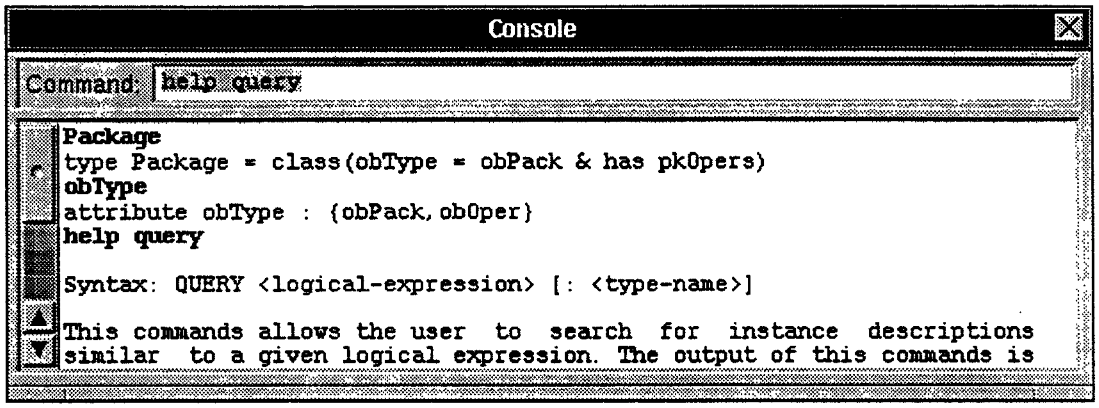


Figure A.3: The Panels submenu

Figure A.4 shows three sample commands and their output. The first command displays the definition of the user-defined class "**Package**", and the second the definition of the attribute "**obType**" (object type). The last command, "**help query**", is used to obtain information regarding the **query** command.

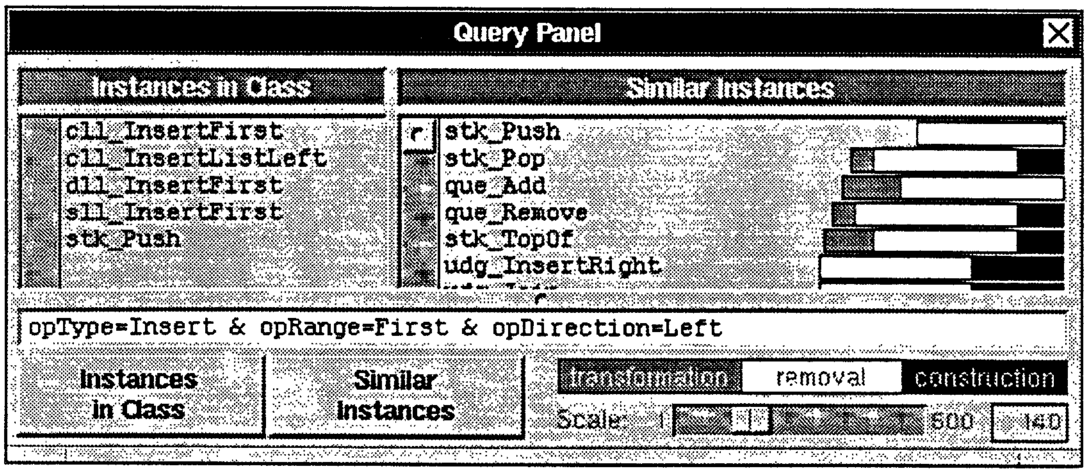


```
Package
type Package = class(obType = obPack & has pkOpers)
obType
attribute obType : {obPack, obOper}
help query

Syntax: QUERY <logical-expression> [: <type-name>]

This commands allows the user to search for instance descriptions
similar to a given logical expression. The output of this commands is
```

Figure A.4: The Console panel

The "Query Panel" command in the "Panels" submenu activates a panel that allows the user to retrieve instances using EDF's similarity-based retrieval mechanism. The Query panel, shown in Figure A.5, has a text field in which the user can type arbitrary EDF expressions. An expression is used to retrieve instances based on two different criteria: (1) instances that belong to the class defined by the expression, and (2) instances that are similar to the *target* instance defined implicitly by the expression.

The list of instances in the class defined by the expression is computed when the user selects the "Instances in Class" button, and is shown in the table by the same name. Likewise, the button and table named "Similar Instances" are used for similarity-based retrieval. Instance names in both lists can be selected using the *mouse* device, and their definitions can be inspected using an Inspect panel (explained below).



```
cll_InsertFirst
cll_InsertListLeft
dll_InsertFirst
sll_InsertFirst
stk_Push
```

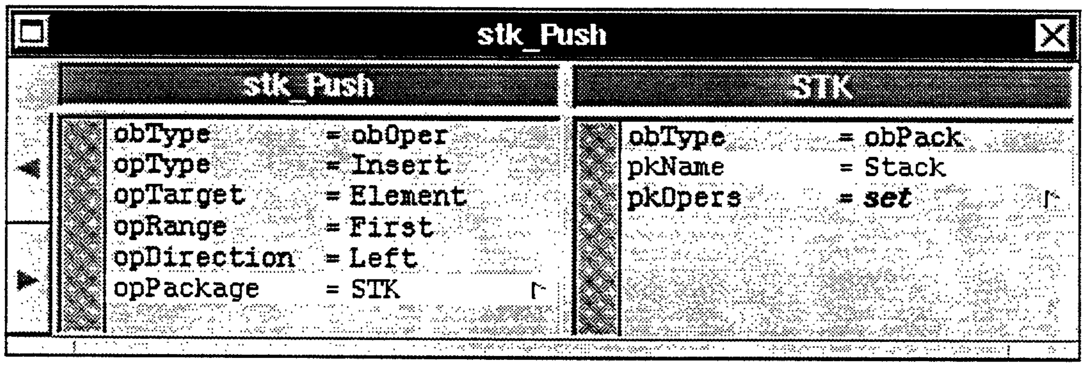
```
stk_Push
stk_Pop
que_Add
que_Remove
stk_TopOf
udg_InsertRight
```

```
opType=Insert & opRange=First & opDirection=Left
```

Figure A.5: The Query panel

The table "Similar Instances" displays instances sorted by increasing distance (i.e., decreasing similarity) to the *target* expression. The distance from each *candidate* instance to the *target* is proportional to the length of the horizontal bar displayed with each name in the list. Each bar is divided in at most three sections representing the transformation (dark-gray), removal (white), and construction (black) magnitudes that compose the overall similarity distance. The length of these bars can be adjusted using the horizontal scroll bar named "Scale".

The "Inspect Item" command of the "Panels" submenu is used to bring up a window with the definition of the last instance selected in the "Instances in Class" or "Similar Instances" lists of the Query panel. The user can have as many Inspector panels open as required.



```
obType       = obOper
opType       = Insert
opTarget     = Element
opRange      = First
opDirection  = Left
opPackage    = STK
```

```
obType       = obPack
pkName       = Stack
pkOpers      = set
```

Figure A.6: The Instance Inspector panel

Figure A.6 shows an Inspector panel with the definition of the stk_Push instance. The leftmost table of the panel displays the list of attribute values that define the instance; attribute names are separated by their respective value by an "=" sign. Compound attribute values, such as instances and sets of values, are shown with a triangle to the right of the value's name (e.g., opPackage and pkOpers).

If these entries are selected, their associated definition or members are shown in a separate table. For example, when the entry "opPackage = STK" was selected, the definition of the STK instance was shown in the table titled STK. Similarly, if the entry "pkOpers = set"[2] had been selected, a third table would be shown with the list of operations of the package.

The "Show Value" and "Show Attribute" commands of the "Panels" submenu (see Figure A.3) are used in conjunction with selected entries in an Inspector Panel. If the "Show Value" command is selected, it displays the source code definition of the value of the entry selected in the front most Inspector panel. Source code is displayed by activating a text window with the contents of the library where the value was defined, and then scrolling the window to show the value's first source line. The "Show Attribute" command displays the source code definition of the attribute of the selected Inspector entry.

A.2 Distance Computation Module

The Distance Computation module implements the default comparators for all EDF types (i.e., numbers, strings, enumerations, classes, and sets). It also implements a set of utility functions required to improve the efficiency of distance computations.

The main algorithm related to similarity computation is shown in Figure A.7. This algorithm computes the list of similar candidate instances for a given target instance. Most of the work in this algorithm is performed by the function `computeDistance`. This function receives as input references to two value entity structures of the same type, and returns one of the following values depending on which arguments are supplied: if the *candidate* and the *target* are both not NIL, it computes their transformation distance. If the *candidate* is NIL, it computes the construction distance for the *target*. Finally, if the *target* is NIL, it computes the removal distance for the *candidate*. In any event, if the magnitude cannot be computed, it returns the special value INFINITY.

The function `computeDistance` performs its tasks using the comparators associated with its arguments. These may be either the default comparators described in Section 3.3.2, or foreign comparators (explained in Section A.2.1). To improve efficiency, the implementation stores distance magnitudes involving instances in a Hash Table[3] indexed by two keys: the addresses of the *candidate* and the *target* arguments. In this way, once the distance between a pair of instances has been computed, it can be retrieved from the table in subsequent distance computations.

This Hash Table is also used to solve the problem of recursive references among instance descriptions. This problem is best understood by analyzing a

[2]Anonymous sets of values are given the generic name "*set*".

[3]Only instance values are inserted in the table because empirical measurements have shown that they are responsible for most of the CPU time used by the `computeDistance` function.

```
algorithm computeSimilarInstances (target:instance)
    listOfPairs := ();
    foreach candidate:instance in the library
        distance := computeDistance(candidate,target);
        if (distance not equal INFINITY) then
            listOfPairs := listOfPairs + [candidate,distance];
    end foreach;
    return listOfPairs sorted by increasing 'distance';
end algorithm;
```

Figure A.7: Algorithm for computing similar instances

specific example. Consider the following sample EDF definitions:

```
stack   = [size=bounded  & opers={push,...}];
push    = [package=stack & function=insert];
queue   = [size=bounded  & opers={dequeue,...}];
dequeue = [package=queue & function=remove];
```

If the function `computeDistance` is used to compute the transformation distance from `push` to `dequeue`, it must compute the distance from `stack` to `queue`. This requires the distance from the set "{`push,...`}" to the set "{`dequeue,...`}", which brings us back to computing the distance from `push` to `dequeue` in an infinite loop.

The method used to solve this infinite-loop problem relies on the Hash Table previously described, and on the fact that distances between instances are computed as the summation of the distances of their associated attribute values. The idea is the following: each time we need to compute the distance between two instance values, the system checks whether this pair has been inserted in the table. If it has not, the pair is inserted with an associated distance of zero (0), and then the computation is carried on as usual. If the pair is in the table, its associated distance is returned without further computation. Furthermore, distances between attribute values of the pair are added directly to their entry in the Hash Table as soon as they are computed.

If this method is applied to our previous example, the distance computation algorithm does not enter in an infinite loop because the second time the pair of instances `push` and `dequeue` is encountered, the system will return immediately with whatever distance value is currently stored in their table entry. Eventually, the algorithm will return to complete the original distance computation from `push` to `dequeue` and will leave the correct distance magnitude in their table entry.

A.2.1 Support of Foreign Comparators

EDF defines default *comparators* for each different kind of EDF type. Although default comparators are well suited for certain domains, sometimes it is necessary to define alternative comparators to be able to capture the semantics and relations of specific objects and attributes. For this purpose, EDF allows the library designer to define arbitrary comparators which can be assigned to any attribute or type using the "`distance`" clause.

For example, consider the attribute definition shown in Figure A.8. This statement defines an attribute called `linesOfCode` with an associated type of `number`. It also assigns the attribute three foreign comparators specified in ANSI C language.

```
%{
#define REMOVE_DISTANCE 0.0
#define MAXIMUM_LOC_DIF 100.0
%}

attribute
    linesOfCode : number
distance
    {
    sourceValue -> targetValue :   /* Transformation comparator */
        {
        double srcLOC = EDF_number(sourceValue);
        double tarLOC = EDF_number(targetValue);
        double difLOC = abs(srcLOC - tarLOC);
        RETURN(min(MAXIMUM_LOC_DIF,difLOC));
        },
    -> constructValue :            /* Construction comparator   */
        {
        RETURN(EDF_number(constructValue));
        },
    removeValue -> :               /* Removal comparator        */
        {
        RETURN(REMOVE_DISTANCE);
        }
    };
```

Figure A.8: Definition of a foreign comparator

The first comparator computes the transformation distance from `sourceValue` to `targetValue` as the minimum between `MAXIMUM_LOC_DIF` and the absolute difference of their magnitudes. The second comparator computes the construction distance for `constructValue` as its associated magnitude. Finally, the third comparator defines the removal distance as the value `REMOVE_DISTANCE`, independent of its argument `removeValue`.

Although the comparators in Figure A.8 are small, their structure is typical. First, none of the comparators has an explicit name; their identity is defined by the number of arguments they have and their disposition with respect to the "`->`" symbol. Second, these arguments can be used within the body of the comparator to retrieve their associated data value using utility functions such as `EDF_number` (see below). Third, the body of comparators may contain an arbitrary number of ANSI C statements, just as regular ANSI C functions do. Fourth, comparators must terminate their execution using the `RETURN` macro, which returns the computed distance. Fifth, global definitions common to all distance comparators in an EDF library can be placed within the delimiters "`%{`" and "`%}`".

A.2.2 Built-in Support Functions

Foreign comparators can make use of a set of built-in utility routines provided by the EDF prototype system. These routines provide access to all relevant internal EDF structures required for distance computations. All these definitions are contained in a file called "`edf.hdr`" used by the EDF prototype to compile foreign comparators.

- Function: `int EDF_ItemType (EDFItem item)`

  This function receives an item descriptor and returns an integer code (see Table A.1) indicating the type of the item. If the type of the argument is unknown, the function returns the code `UNDEFINED`.

Code	Description
`UNDEFINED`	Undefined entity structure
`V_NUMBER`	Argument is a Number value
`V_STRING`	Argument is a String value
`V_TERM`	Argument is an enum. Term value
`V_INSTANCE`	Argument is an Instance value
`V_SET`	Argument is a Set value
`T_NUMBER`	Argument is a Number type
`T_STRING`	Argument is a String type
`T_ENUM`	Argument is an Enumeration type
`T_SETOF`	Argument is a Set type
`T_CLASS`	Argument is an Class type

Table A.1: Return codes of the function `EDF_ItemType`

- Function: `EDFItem EDF_FindItem (char *identifier)`

  This function receives a string containing an identifier name and returns the object descriptor associated with that name. If no such descriptor exists, it returns `NIL`.

- Function: `int EDF_UserItem (EDFItem item)`

  This function receives an item descriptor and returns `TRUE` if the item was user-defined. It returns `FALSE` if the item was internally defined by EDF.

- Function: `char *EDF_ItemName (EDFItem item)`

  This function receives an item descriptor and returns a reference to a character string containing the name of the item. In case of error, it returns `NIL`.

- Function: `char *EDF_ItemFile (EDFItem item,int *fileLine)`

  This function receives an an item descriptor and returns a string containing the file name of the library where the item was defined. It also returns the file line position number of the item in the argument `fileLine`. In case of error, it returns `NIL`.

- Function: `int EDF_SetLength (EDFItem set)`

  This function receives a descriptor of an enumeration or a set of values and returns the number of elements in the set. Empty sets have length `0`. In case of error, it returns -1.

- Function: `EDFItem EDF_SetElement (EDFItem set,int index)`

  This function receives a descriptor of an enumeration or a set of values and returns the object descriptor in position `index` in the set. In case of error, it returns `NIL`.

- Function: `int EDF_Query (char *expression)`

  This function is equivalent to the `query` command. It receives a string containing an expression and computes a list of candidate instances that are similar to the target instance defined by the expression. It returns the length of the computed list. In case of error, it returns -1.

- Function: `EDFItem EDF_QueryElement (int index,EDFDist *dist)`

  This function is used after invoking the function `EDF_Query`. It returns a reference to the instance descriptor with position `index` in the list of similar instances computed by `EDF_Query`. It also returns its associated distance magnitude in the argument `dist`. In case of error, it returns `NIL`.

- Function: `EDFItem EDF_TermEnum (EDFItem term)`

  This function receives a descriptor of an enumeration term and returns a reference to the term's associated enumeration descriptor. If the argument is not an enumeration term it returns `NIL`.

- Function: `int EDF_InstLength (EDFItem inst)`

  This function receives a descriptor of an instance and returns the number of pairs (attribute,value) that define the instance. In case of errors, it returns -1.

- Function: `EDFItem EDF_InstAttribute (EDFItem inst,int index)`

  This function receives a descriptor of an instance and returns a the descriptor of the attribute with position `index` in the list of pairs (attribute,value) of the instance. In case of error, it returns `NIL`.

- Function: `EDFItem EDF_InstValue (EDFItem inst,int index)`

  This function receives a descriptor of an instance and returns the descriptor of the value with position `index` in the list of pairs (attribute,value) of the instance. In case of error, it returns `NIL`.

- Function: `int EDF_InClass (EDFItem class,EDFItem inst)`

  This function receives a descriptor of a class and an instance and returns `TRUE` if the instance is a member of the class. Otherwise, it returns `FALSE`.

- Function: `int EDF_Distance (EDFItem v1,EDFItem v2,EDFDist *dist)`

  This function receives two object descriptors of the same type and returns the distance from `v1` to `v2` in `dist`. In case of enumeration terms, both arguments must belong to the same enumeration. In case of error, it returns FALSE.

- Function: `double EDF_number (EDFItem number)`

  This function receives a descriptor of a number and returns its associated integer value. If the argument is not a number value it returns -1.

- Function: `char *EDF_string (EDFItem string)`

  This function receives a descriptor of a string and returns a reference to its associated string. If the argument is not a string value it returns `NIL`.

A.3 Verification and Resolution Module

The EDF language has certain properties that make it hard to implement. The first of these properties is that definitions in a library can make use of forward references to other entities, therefore it is generally not possible to verify the consistency of any particular definition until the entire library file has been parsed. The algorithm for solving forward references is described in Section A.3.1.

The other aspect of the language that is hard to solve has to do with computing the correct set of attribute values that define an instance. An instance definition is said to be "resolvable" if its associated expression can be reduced to a conjunction of non conflicting attribute values. For example, the following instance definitions are not resolvable:

```
I1 = [size=bounded | size=unbounded];
I2 = [size=bounded & size=unbounded];
I3 = [has color    & size=unbounded];
```

The definition of I1 cannot be resolved because its associated expression has two disjuncts, and none of them is a contradiction. The definition of I2 is a conjunction, but the attribute **size** has been assigned two contradictory values. In the case of I3, the expression is also a conjunction, but its **has** operator is not satisfied; i.e., the attribute **color** has no explicit value. Contradictions do *not* necessarily invalidate an instance definition. Consider the following:

```
I4 = [size=bounded & size=unbounded | color=red];
```

In this case the expression of I4 has two disjuncts, but the left one can be eliminated because it defines a contradiction, hence the final resolved definition of I4 is "[color=red]". To understand this, one has to remember that expressions describe sets of objects. Each attribute assignment defines the set of objects that have a particular property value, and the "&" and "|" operators represent unions and intersections of sets. In the case of I4, the sets "size=bounded" and "size=unbounded" are mutually exclusive so their intersection (&) is empty, therefore the expression is reduced to the union (|) of the empty set and the set "color=red", which is simply the latter.

In general, the process of reducing the number of disjuncts cannot be done on an instance-by-instance basis, but must be performed using an iterative algorithm. To understand why, consider the following definitions:

```
attribute size  : {small,big};
attribute elems : set of class(has size);
I1 = [size=big   & elems={I2} & elems=set(size=small)];
I2 = [size=small | elems={I2} & elems=set(size=big)];
```

To reduce the expression of I1 it is necessary to compute the set of instances "set(size=small)" and make sure it is equal to the set "{I2}"[4]. There are two instances that can be members of the set "set(size=small)", either I1 or I2. The instance I1 can not be a member, because it does not satisfy the expression "size=small". Instance I2, on the other hand, has two disjuncts, one of which satisfies this expression, but it cannot be made a member of the set "set(size=small)" until we eliminate the second disjunct by proving it is a contradiction. To do this, we need to compute the set "set(size=big)" and show that it is not equal to the set "{I2}". This brings us back to where we started, which is to reduce the definition of I1.

A.3.1 Forward References

As with other languages that allow forward references, the EDF prototype system uses essentially a two-pass compiler. In the first pass, it parses all the definitions

[4]If these sets are not equal, the entire expression would be a contradiction, and instance I1 could not be resolved.

in a library and transforms them into two internal structures: symbols and entities. References among definitions are always done via their associated symbol structures, and not directly between their entity structures.

Once a library has been completely parsed, the EDF prototype system performs a second-pass through all the definitions in memory. This second-pass checks that all symbols have been bound to an entity structure (i.e., they have been defined), and that all cross references among definitions are of the proper type. For example, consider a library with the following two definitions:

```
I1 = instance(package=I2);
package = class(has size);
```

In this case, the definition of the instance `I1` has a forward reference to the identifier `package`. Although `package` has later been defined as a class, it is being used incorrectly as an attribute. Also, this library has two undefined identifiers, namely `I2` and `size`. Because of the semantics of the `has` operator, the system is able to produce a specific error message indicating that `size` should have been defined as an attribute. The case of `I2` is not so clear cut; given that `package` is not an attribute, the only thing the system knows is that `I2` should have been defined as some kind of value entity.

The function that performs the second-pass is called `checkReferences`; it simply applies the auxiliary function `checkSymbol` (see Figure A.9) to each symbol in the Symbol Table. Basically, function `checkSymbol` obtains the symbol's entity structure and checks it by invoking the entity's `check` method. This method verifies the entity's structure and returns a status code of `TRUE` if the test is successful, and `FALSE` otherwise.

This algorithm is complicated by several factors. First, a symbol may be undefined, in which case an error message must be given. Second, the `check` methods use the function `checkSymbol` recursively to check the symbols referenced by their associated entity structure. To avoid infinity loops, each symbol is marked as visited before invoking `check`, and if it is already marked nothing is done with it. Third, the symbol's status code is stored in its structure so it can be immediately returned by `checkSymbol` if the symbol has already been visited.

A.3.2 Resolution of Instances

The semantics defined for instances require that they be placed in a "resolved" state, that is, their associated expressions must be reduced to a conjunction of non-conflicting attribute values. The first step of this resolution process is to transform each instance expression into its equivalent Disjunctive Normal Form (DNF). The next step involves computing all implicit attribute values (e.g., those defined with the `set` function) and then eliminating all contradictory disjuncts; each disjunct being a conjunction of attribute values. The process terminates once all instance DNFs have been reduced to one disjunct.

```
function checkSymbol (symbol)
    status := symbol.status;
    if (not symbol.visited) then
        symbol.visited := TRUE;
        entity := symbol.entity;
        if (entity is not NIL) then
            symbol.status := TRUE;
            status := entity.check();
        else
            status := FALSE;
            message 'Undefined symbol';
        end if
        symbol.status := status;
    end if
    return status;
end function
```

Figure A.9: Algorithm for checking a symbol definition

The EDF prototype system performs disjunct reduction and solves dependencies among attribute values using an iterative process that combines techniques for doing *lazy evaluation*[5] and heuristic rules for dealing with difficult situations and avoiding infinity loops. This iterative process is called the Resolution Algorithm, and terminates either with all instances resolved, or with an error message indicating that an unresolvable circular dependency has been discovered.

The main data structure used by the Resolution Algorithm is a singly linked list of nodes called `PNodes`, where each node represents a value entity structure requiring additional processing. Each `PNode` has four fields: `valueDef`, `selfInst`, `auxData`, and `nextNode`. The field `valueDef` references the value entity that must be resolved. The field `selfInst` is a reference to the instance being defined with the value `valueDef`. This field is required for those values that can be defined in terms of the `self` keyword, such as implicit or explicit sets of values. The field `auxData` is used to store "state" information regarding the resolution of the value. The particular information stored in this field is explained below. Finally, the field `nextNode` links the node to the next one in the circular linked list of

[5]Lazy Evaluation is a technique commonly used to implement functional languages such as "Miranda" [94] or "Lazy ML" [5]. A complete review of this technique is beyond the scope of this dissertation, and only those aspects that are used to resolve instances will be explained. For a full treatment of the subject, the interested reader is directed to "Functional Programming" by Field and Harrison [35].

pending values.

The EDF prototype system creates an initial list of pending values composed of all defined instances. Once this list has been created, it invokes the algorithm shown in Figure A.10. Basically, `resolveEntities` iterates through all nodes in the list `nodeList` over and over again until the list becomes empty, or until an unresolvable circular dependency is discovered. Each time it traverses the list, it invokes the `resolve` method associated with the entity of each node. This method performs whatever necessary actions, and returns one of the following three status codes: COMPLETED, PROGRESS, or NOPROGRESS.

```
algorithm resolveEntities (nodeList:PNode)
    while (nodeList not empty) do
        progress := 0;
        foreach pnode in nodeList do
            entity := pnode.entity;
            select (entity.resolve(pnode)) of
                case COMPLETED:
                    remove 'pnode' from nodeList;
                    progress := progress + 1;
                case PROGRESS:
                    progress := progress + 1;
                case NOPROGRESS:
                    continue;
            end select
        end foreach
        if (progress equals 0) then
            message 'Unresolvable circular dependency';
            abort algorithm;
        end if
    end while
end algorithm
```

Figure A.10: Algorithm for resolving definitions

- COMPLETED: Indicates the method has completed the resolution of its associated value, and hence it should not be invoked again. The algorithm simply removes the appropriate node from the list of pending nodes, and continues processing the remaining ones. It also records progress in the overall resolution algorithm.

- `PROGRESS`: Indicates the method was able to advance in the resolution of its associated value, but it still requires additional work to complete its task. The algorithm leaves the node in the list of pending nodes, thus ensuring that it will be processed in the next iteration. It also records progress in the overall resolution algorithm.

- `NOPROGRESS`: Indicates the method was not able to advance at all in the resolution of its associated value, and that it should be invoked again to attempt further progress. As with the previous case, the node is left in the list of pending nodes, but no progress is recorded.

The concept of "progress" is essential to this algorithm, because it allows detection of unresolvable circular dependencies. For example, consider the case of a pending node computing the list of instances defined by a `set` function. The `resolve` method for this node checks each defined instance in a library to see if it matches the expression. If it does not, it discards it, and checks the next one. If it does, it adds the instance to the list. In both of these cases the method returns the code `PROGRESS`, because it has reduced the amount of remaining work it needs to complete to compute the final list of instances. Now, if the method checks all candidate instances and it is unable to either discard or include any of them, it returns the code `NOPROGRESS`.

When a `resolve` method reports "no progress", it does *not* necessarily mean that it will be unable to accomplish its task. Normally, it just means that all the values required by the method are temporarily in an "unresolved" state, and therefore it has to wait for the next iteration of the algorithm to continue its task. Of course, if all the nodes that are pending report "no progress", then we have discovered an unresolvable circular dependency.

Resolving Instances

As explained earlier, the list of pending nodes contains initially one node for each instance defined in a library. When the Resolution Algorithm is activated, the `resolve` method associated with each instance is invoked until it returns a `COMPLETED` status code. After each invocation, the method may also return the codes `PROGRESS` or `NOPROGRESS` depending on the amount of progress it has been able to accomplish.

In general, the task performed by `resolve` methods is divided in two phases. The first is called the *initialization* phase, and is performed the first time the method is activated. The second phase is called *resolution*, and it is performed by subsequent invocations until the entity has been resolved. The particular steps performed by the *initialization* phase for instances is the following:

1. Transform the instance expression to its equivalent Disjunctive Normal Form (DNF).

2. Traverse each of the disjuncts obtained in the previous step, and add all attribute values that need resolution to the list of pending nodes of the Resolution Algorithm.

3. Return status code PROGRESS.

The *resolution* phase for instances basically attempts to eliminate all contradictory disjuncts from the DNF. These disjuncts are discovered performing the following two tests:

1. Make sure that all disjuncts with operators of the form "has *id*", also have at least one operator of the form "*id* = *v*". In other words, verify that all has operators are satisfied. If a disjunct has an unsatisfied has operator, then it is a contradiction, and is removed from the DNF.

2. For each pair of operators "$id = v_1$" and "$id = v_2$", verify that the values v_1 and v_2 are equal. If they are, remove one of the operators. If they are not, the disjunct is a contradiction, and is removed from the DNF.

The first step above can always be performed independently of whether the value v has been resolved, because this value is not required to verify the has operator. This is not the case in the second step, where either v_1 or v_2 or both may be in an unresolved state, and therefore it may not be possible to determine their equality. Although this is generally the case, there are some circumstances where the equality (or non-equality) can be established without having fully resolved values. The algorithm for doing this kind of test uses *lazy evaluation* techniques to accomplish its task, and is described later in the section "Lazy Comparator", page 120.

The Lazy Comparator returns one of three status codes: EQUAL, NOTEQUAL, and UNKNOWN. The meaning of the first two is obvious. The last code is returned when there is not enough information to determine equality or non-equality of the two values. In this case, the *resolution* phase must terminate and return the status code PROGRESS (or NOPROGRESS if nothing was accomplished) so it can continue its tasks in the next iteration of the Resolution Algorithm.

Resolving Implicit Sets

Implicit sets of instances are defined via the EDF built-in function `set`, and its members are all user-defined instances whose expression matches the expression associated with this function. In general, implicit sets cannot be computed without the instances being resolved, but, at the same time, instances that have implicit sets as attribute values cannot be computed until these values are resolved. Because of these potential circular dependencies, implicit sets must be added to the list of pending nodes and computed using the Resolution Algorithm.

As with instance values, the Resolution Algorithm activates the `resolve` method associated with each implicit set until it returns a COMPLETED status code. After each invocation, the method may also return the codes PROGRESS or NOPROGRESS depending on the amount of progress it has been able to accomplish.

To accomplish its task, the `resolve` method uses an auxiliary structure where it stores the state of its computation. This structure is called Pending Set (PSet), and contains three lists: `candidates`, `members`, and `nonMembers`. The basic idea behind this structure is that instances in the list `candidates` are tested for membership in the set. Depending on the outcome of the test, each instance is moved either to the list `members` or `nonMembers`. The `candidates` list represents the work that must still be done, while the `members` list is what we need to compute. The last list, `nonMembers`, is used by the Lazy Comparator to compare two unresolved sets.

The *initialization* phase for implicit sets is similar to that of instances, and is composed of the following steps:

1. Create a PSet structure and initialize the field `candidates` with the list of all user-defined instances, and the fields `members` and `nonMembers` to the empty list. Store the resulting PSet structure in the field auxData for future access (see page 115).

2. Transform the set's expression to its equivalent Disjunctive Normal Form.

3. Traverse each of the disjuncts obtained in the previous step, and add all attribute values that need resolution to the list of pending nodes of the Resolution Algorithm.

4. Return status code PROGRESS.

The *resolution* phase for implicit sets basically attempts to move instances from the list `candidates` to either the list `members` or `nonMembers`. It terminates with a status code of COMPLETED when the list of candidates becomes empty. The main task in this process is to determine whether an instance belongs to the set; to explain it we need the following definitions:

- Let N^s be the number of disjuncts in the Disjunctive Normal Form of the expression associated with the implicit set, and let D^s_k (for $1 \leq k \leq N^s$) represent each disjunct. Also, let N^s_k be the number of conjuncts in each disjunct D^s_k, and C^s_{kj} (for $1 \leq j \leq N^s_k$) represent each of the conjuncts of disjunct k.

- Similarly, let N^c be the number of disjuncts in the Disjunctive Normal Form of the expression associated with a candidate instance, and let D^c_k (for $1 \leq k \leq N^c$) represent each disjunct. Also, let N^c_k be the number of conjuncts in each disjunct D^c_k, and C^c_{kj} (for $1 \leq j \leq N^c_k$) represent each of the conjuncts of disjunct k.

For example, the implicit set "`set(size=big & color=red)`" has only one disjunct (i.e., $N^s = 1$). This disjunct has two conjuncts (i.e., $N^s_1 = 2$) where C^s_{11} = "`size=big`" and C^s_{12} = "`color=red`".

We say that a candidate instance, c, belongs to an implicit set, s, if and only if the following is true:

$$\forall\, i \in [1..N^c]\ \exists\, j \in [1..N^s] : dmatch(D^c_i, D^s_j) \tag{A.1}$$

where the predicate $dmatch(D^c_i, D^s_j)$ is true if and only if the disjuncts satisfy the following condition:

$$\forall\, k \in [1..N^c_i]\ \exists\, n \in [1..N^s_j] : cmatch(C^c_{ik}, C^s_{jn}) \tag{A.2}$$

and the predicate $cmatch(C^c_{ik}, C^s_{jn})$ is true if and only if the conjuncts satisfy one of the following two conditions:

1. If the conjunct C^s_{jn} is an operator of the form "**has** *id*" and the conjunct C^c_{ik} is of the form "$id = v$", for any value v.

2. If the conjunct C^s_{jn} is an operator of the form "$id = v_1$" and the conjunct C^c_{ik} is of the form "$id = v_2$" and the values v_1 and v_2 are equal.

Equation A.1 states that all disjuncts of the candidate instance must *match* at least one of the disjuncts of the implicit set. Similarly, equation A.2 states that all the conjuncts of the candidate disjunct must *match* at least one conjunct of the implicit set disjunct. The final two conditions to have a pair of conjuncts match are straightforward, and were explained earlier in this section.

We know from a previous discussion that comparing the values v_1 and v_2 will require the assistance of the Lazy Comparator. Given that this comparator may return the code `UNKNOWN` (as opposed to `EQUAL` or `NOTEQUAL`), it is not clear what the *truth* value of the above conditions should be in this case. The `resolve` method handles this situation by considering the condition to be *false*, but it does not remove the instance from the list of candidates. In this way, it will test its membership in the next iteration of the Resolution Algorithm.

Lazy Comparator

The Lazy Comparator is a three-valued function that compares two value entities and returns one of the following status codes: `EQUAL`, `NOTEQUAL`, and `UNKNOWN`. This function is able, in some circumstances, to determine the equality (or non-equality) of the values when one or both are in an unresolved state.

One obvious condition that does not require resolved values is that both of them must be of the same type. If this is not the case, the status code `NOTEQUAL` is immediately returned. In the case of enumeration terms, the function also tests that both terms belong to the same enumeration. Another test that does not require the values to be solved is to compare the addresses of the two values. If they both reference the same value structure, then they are by definition equal, and therefore the comparator returns the code `EQUAL` immediately without further analysis.

Of all the different value types, only two of them require special attention: instances and sets of values. The remaining ones (i.e., numbers, strings, and enumeration terms) are always in a resolved state, and therefore their equality (or non-equality) can be determined in a standard way.

Sets of Values. As explained earlier, each set of values has an associated structure called `PSet` which contains three lists of values: `candidates`, `members`, and `nonMembers` (see page 119). To determine the resulting status code for two sets `S1` and `S2`, the Lazy Comparator performs the following tests:

1. If both sets have been resolved, then the comparator returns `EQUAL` if their `members` lists contain the same elements. Otherwise, it returns `NOTEQUAL`.
2. If one of the sets, say `S1`, is resolved and the other is not, they are compared as follows:
   (a) If one of the elements of the `members` list of `S1` also appears in the `nonMembers` list of `S2`, the comparator returns `NOTEQUAL`.
   (b) If one of the elements of the `members` list of `S2` does not appear in the `members` list of `S1`, the comparator returns `NOTEQUAL`.
3. If both, `S1` and `S2`, are unresolved and one of the elements of the `members` list of one set appears in the `nonMembers` list of the other, the comparator returns `NOTEQUAL`.
4. If none of the above conditions is true, the comparator returns `UNKNOWN`.

When we say that an element E_1 of a set S_1 "appears" in another set S_2, it means either that the E_1's address appears in set S_2, or that there exists an element E_2 in S_2 that is equal to E_1. The latter condition requires the Lazy Comparator to call itself recursively.

Instances. When comparing two instance values, the Lazy Comparator distinguishes two different cases: (1) both instances are resolved, and (2) one, or both, instances are unresolved. In the first case, the comparator returns `EQUAL` if both are defined using the same set of attributes, and if their corresponding attribute values are equal. Otherwise, it returns `NOTEQUAL`. This situation can never generate an `UNKNOWN` status code, because resolved instances have all their associated attribute values resolved.

The case in which one or both instances are unresolved is more complicated, and can only generate a `NOTEQUAL` or `UNKNOWN` status code. Basically, the idea is that the comparator attempts to prove that the two instances will never be equal. If it succeeds, it returns `NOTEQUAL`, otherwise it simply returns `UNKNOWN`.

Two instances, s and c, are said to be not equal if the following condition is true (refer to the definitions presented on page 119):

$$\forall\, i \in [1..N^c]\, \forall\, j \in [1..N^s] : different(D^c_i, D^s_j) \tag{A.3}$$

and the predicate $different(D^c_i, D^s_j)$ is true if and only if one of the following two conditions is true:

$$\exists\, k \in [1..N^c_i]\, \forall\, n \in [1..N^s_j] : (C^c_{ik}.a \neq C^s_{jn}.a) \vee (C^c_{ik}.v \neq C^s_{jn}.v) \tag{A.4}$$

$$\exists\, k \in [1..N_i^s]\, \forall\, n \in [1..N_j^c] : (C_{ik}^c.a \neq C_{jn}^s.a) \vee (C_{ik}^c.v \neq C_{jn}^s.v) \tag{A.5}$$

Equation A.3 is very strong. It states that two instances can be guaranteed to be not equal if all possible pairs of disjuncts are different. Equations A.4 and A.5 are symmetric with respect to s and c, and they state that a disjunction, D_1, is different from another, D_2, if there exists at least one operator in D_1 whose attribute name does not appear in an operator in D_2, or if it does appear, their corresponding attribute values are different.

Bibliography

[1] B. Allen and D. Lee. A knowledge-based environment for the development of software parts composition systems. In *Proceedings of the 11th International Conference Software Engineering*, 1989.

[2] K.J. Anderson, R.P. Beck, and T.E. Buonanno. Reuse of software modules. *AT&T Technical Journal*, 67(4):71–76, July-August 1988.

[3] Anonymous. The full computing reviews classification scheme—1987 version. *Computer Review*, 29, January 1988.

[4] S. Arnold and S. Stepoway. The reuse system: Cataloging and retrieval of reusable software. In W. Tracz, editor, *Software reuse: Emerging Technology.* IEEE Computer Society Press, 1988.

[5] L. Augustsson. A compiler for Lazy ML. In *Proceedings ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984.

[6] B.H. Barnes and T.B. Bollinger. Making reuse cost-effective. *IEEE Software*, pages 13–24, January 1991.

[7] V.R. Basili. Software development: A paradigm for the future. Technical Report CS-TR-2263, UMIACS-TR-89-57, University of Maryland, Department of Computer Science, College Park, MD 20742, June 1989.

[8] V.R. Basili. Software development: A paradigm for the future. In *Proceedings of the 13th International Computer Software and Applications Conference (COMPSAC'89)*, pages 471–485, Orlando, Florida, September 20–22 1989. IEEE Computer Society Press.

[9] V.R. Basili. The Goal/Question/Metric paradigm. Unpublished document. University of Maryland, Department of Computer Science, July 5 1990.

[10] V.R. Basili and H.D. Rombach. Tailoring the software process to project goals and environments. In *Proceedings of the 9th International Conference on Software Engineering*, pages 345–357, Monterey, California, March 30–April 2 1987. IEEE Computer Society Press.

[11] V.R. Basili and H.D. Rombach. The TAME project: Towards improvement oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, June 1988.

[12] V.R. Basili, H.D. Rombach, J. Bailey, and A. Delis. Ada reusability and measurement. Technical Report CS-TR-2478, UMIACS-TR-90-73, University of Maryland, Department of Computer Science, College Park, MD 20742, 1990.

[13] T.J. Biggerstaff and A.J. Perlis. Foreword. *IEEE Transactions on Software Engineering*, SE-10(5):474–476, September 1984.

[14] T.J. Biggerstaff and A.J. Perlis, editors. *Software Reusability, Volume I: Concepts and Models.* ACM Press Frontier Series. Addison-Wesley, Reading, Massachusetts, 1989.

[15] T.J. Biggerstaff and A.J. Perlis, editors. *Software Reusability, Volume II: Applications and Experience.* ACM Press Frontier Series. Addison-Wesley, Reading, Massachusetts, 1989.

[16] T.J. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2):41–49, March 1987. Also in T.J. Biggerstaff and A.J. Perlis, eds., *Software Reusability*, Volume I, ACM Press, 1989.

[17] T.B. Bollinger and B. Barnes. Reuse rules: An adaptative approach to reusing Ada software. In *Proceedings of the AIDA '88*, Fairfax, Virginia, 1988. George Mason University.

[18] T.B. Bollinger and S.L. Pfleeger. Economics of reuse: Issues and alternatives. *Information and Software Technology*, 32(10):643–652, December 1990.

[19] J. Bolstad. A proposed classification scheme for computer libraries. *ACM SIGNUM Newsletter*, 10(2–3):32–39, November 1975.

[20] G. Booch. *Software Components with Ada.* Benjamin-Cummings Publishing Company, Menlo Park, California, 1987.

[21] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1986.

[22] B. Buchanan. *Theory of Library Classifications.* Clive Bingley, London, UK, 1979.

[23] B.A. Burton, R.W. Aragon, S.A. Bailey, K.D. Koehler, and L.A. Mayes. The reusable software library. *IEEE Software*, 4(4):25–33, July 1987.

[24] G. Caldiera and V.R. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, 24(2):61–70, February 1991.

[25] Center for Applied Mathematics, National Bureau of Standards, Washington D.C. *Guide to Available Mathematical Software*, 1980.

[26] E. Charniak, C.K. Riesbeck, D.V. McDermott, and J.R. Meehan. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1987.

[27] J.F. Cloarec, L. Bruneau, and A. Feroldi. Application of AI techniques to software reuse. *Annales des Telecommunications*, 44(5–6):317–323, 1989.

[28] A.M. Collins and E.F. Loftus. A spreading activation theory of semantic processing. *Psych. Rev.*, 82:407–428, 1975.

[29] S.D. Conte, H.E. Dunsmore, and V.Y. Shen. *Software Engineering Metrics and Models.* Benjamin/Cummings, Menlo Park, California, 1986.

[30] B.J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, Reading, Massachusetts, 1986.

[31] Defense Technical Information Center, Alexandria, Virginia. *DTIC Retrieval and Indexing Terminology*, 1987.

[32] P.T. Devanbu, R.J. Brachman, P.G. Selfridge, and B.W. Ballard. LaSSIE: A knowledge-based software information-system. *Communications of the ACM*, 34(5):34–49, 1991.

[33] M. Dewey. *Decimal Classification and Relative Index.* Forest Press Inc., Albany, New York, 19th edition edition, 1979.

[34] D.W. Embley and S.N. Woodfield. A knowledge structure for reusing abstract data types. In *Proceedings of the 9th International Conference on Software Engineering*, pages 360–368, Monterey, California, March 30–April 2 1987. IEEE Computer Society Press.

[35] A.J. Field and P.G. Harrison. *Functional Programming.* International Computer Science Series. Addison-Wesley, Reading, Massachusetts, 1988.

[36] G. Fischer. Cognitive view of reuse and design. *IEEE Software*, 4(4):60–72, July 1987.

[37] J. Fitzgerald and R. Mathis. Use of an expert system in software component reuse. In *Proceedings of the AIDA '88*, Fairfax, Virginia, 1988. George Mason University.

[38] A.C. Fosket. *The Subject Approach to Information.* Clive Bingley, London, UK, 1977.

[39] W.B. Frakes and P.B. Gandel. Representing reusable software. *Information and Software Technology*, 32(10):652–664, December 1990.

[40] W.B. Frakes and B.A. Nejmeh. An information system for software reuse. In W. Tracz, editor, *Software reuse: Emerging Technology.* IEEE Computer Society Press, 1988.

[41] J.E. Gaffney and T.A. Durek. Software reuse – key to enhanced productivity: Some quantitative models. *Information and Software Technology*, 31(5):258–267, June 1989.

[42] R.A. Gagliano, M.D. Fraser, G.S. Owen, and P.A. Honkanen. Issues in reusable Ada library tools. In *Proceedings of the 6th EFISS Symposium*, Atlanta, Georgia, 1989.

[43] J.D. Gannon, V.R. Basili, and E.E. Katz. Metrics for Ada packages – an initial study. *Communications of the ACM*, 29(7):616–623, 1986.

[44] A. Gargaro and T.L. Pappas. Reusability issues and Ada. *IEEE Software*, 4(4):43–51, July 1987.

[45] J.A. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16–28, February 1986.

[46] D.E. Harms and B.W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, SE-17(5):424–435, May 1991.

[47] A. Hart. *Knowledge Acquisition for Expert Systems.* McGraw-Hill, 1986.

[48] D. Hutchens and V.R. Basili. System structure analysis: Clustering with Ada bindings. *IEEE Transactions on Software Engineering*, 11, August 1985.

[49] IMSL Inc., Houston, Texas. *International Mathematics and Scientific Library*, 10th edition, 1984.

[50] R. Johnson and B. Foote. Designing reusable classes. *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1(2), 1988.

[51] G. Jones and R. Prieto-Díaz. Building and managing software libraries. In *Proceedings of the 12th International Computer Software and Applications Conference (COMPSAC'88)*, pages 228–236, Chicago, IL, October 1988. IEEE Computer Society Press.

[52] T.E. Cheatham Jr. Reusability through program transformations. *IEEE Transactions on Software Engineering*, SE-10(5):589–594, September 1984.

[53] K.C. Kang. Features analysis: An approach to domain analysis. In *Proceedings of the SEI Reuse in Practice Workshop*, July 11–13 1989.

[54] S.E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, Reading, Massachusetts, 1989.

[55] B.W. Kernighan. The Unix system and software reusability. *IEEE Transactions on Software Engineering*, SE-10(5):513–518, September 1984.

[56] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Software Series. Prentice Hall International, Englewood Cliffs, New Jersey, second edition, 1988.

[57] R. Kester. SEL Ada reuse analysis and representation. Technical Report SEL-90-006, NASA Goddard Space Flight Center, Greenbelt, Maryland 20771, November 1990.

[58] F.W. Lancaster. *Vocabulary Control for Information Retrieval.* Information Resources Press, Arlington, Virginia, 2nd edition, 1986.

[59] R.G. Lanergan and C.A. Grasso. Software engineering with reusable design and code. *IEEE Transactions on Software Engineering*, SE-10(5):498–501, September 1984.

[60] Carlos H. Lauterbach. *Hierarchical Organization of Data Types for Program Modularity.* PhD thesis, University of California at Los Angeles, 1977.

[61] M. Lenz, H.A. Schmid, and P.F. Wolf. Software reuse through building blocks. *IEEE Software*, 4(4):34–42, July 1987.

[62] S.D. Litvintchouk and A.S. Matsumoto. Design of Ada systems yielding reusable components: An approach using structured algebraic specification. *IEEE Transactions on Software Engineering*, SE-10(5):544–551, September 1984.

[63] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An information-retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, August 1991.

[64] Y. Matsumoto. SWB system: A software factory. In H. Hunke, editor, *Software Engineering Environments*, pages 305–318. North-Holland, 1981.

[65] Y. Matsumoto. Some experiences in promoting reusable software: Presentation in higher abstract levels. *IEEE Transactions on Software Engineering*, SE-10(5):502–513, September 1984.

[66] C.T. Meadow. *The Analysis of Information Systems.* Melville Publishing Company, Los Angeles, California, 1973.

[67] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[68] NASA Goddard Space Flight Center, Greenbelt, Maryland 20771. *Software Engineering Laboratory (SEL) Database Organization and User's Guide*, revision 1 edition, February 1990. SEL-89-101.

[69] J.M. Neighbors. The draco approach to constructing software for reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–574, September 1984.

[70] NeXT Computer Inc., 900 Chesapeake Drive, Redwood City, CA 94063. *NeXTstep Concepts*, 1990. Part of the "NeXT Developer's Library" collection.

[71] NeXT Computer Inc., 900 Chesapeake Drive, Redwood City, CA 94063. *NeXTstep Reference Volume I and II*, 1990. Part of the "NeXT Developer's Library" collection.

[72] N. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, California, 1980.

[73] E.J. Ostertag, J.A. Hendler, R. Prieto-Díaz, and C. Braun. Computing similarity in a reuse library system: An AI-based approach. Technical Report TR-91-6, University of Maryland, Systems Research Center, College Park, MD 20742, January 1991.

[74] F.J. Polster. Reuse of software through generation of partial systems. *IEEE Transactions on Software Engineering*, 12(3):402–416, March 1986.

[75] R. Prieto-Díaz. *A Software Classification Scheme.* PhD thesis, Department of Information and Computer Science, University of California at Irvine, 1985.

[76] R. Prieto-Díaz. Domain analysis for reusability. In *Proceedings of the 11th International Computer Software and Applications Conference (COMPSAC'87)*, pages 23–29, Tokyo, Japan, October 1987.

[77] R. Prieto-Díaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):88–97, 1991.

[78] R. Prieto-Díaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, January 1987.

[79] R. Rada, H. Mili, M. Blettner, and E. Bicknell. Development and application of a metric on semantic nets. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(1):17–30, 1989.

[80] S.R. Raganathan. *Prolegomena to Library Classification.* Asian Publishing House, Bombay, India, 1967.

[81] H.D. Rombach. MVP-L: A language for process modeling in-the-large. Technical Report CS-TR-2709, UMIACS-TR-91-96, University of Maryland, Department of Computer Science, College Park, MD 20742, June 1991.

[82] S. Rosales and P. Mehrotra. MES: an expert system for reusing models of transmission equipment. In *Proceedings of the 4th Conference of Artificial Intelligence Applications*, pages 109–113. IEEE Computer Society Press, 1988.

[83] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York, 1983.

[84] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., Boston, 1986.

[85] N.A. Sidorov. Software reusability. *Cybernetics*, 25(3):340–346, 1989.

[86] J.J. Solderitsch, K. Wallnau, and J. Thalhamer. Constructing domain-specific Ada reuse libraries. In *Proceedings of the 7th Annual National Conference Ada Technology*, 1989.

[87] Special Issue on. Software reusability. *IEEE Transactions on Software Engineering*, SE-10(5), September 1984.

[88] Special Issue on. Tools: Making reuse a reality. *IEEE Software*, 4(4), July 1987.

[89] M. Stefik and D. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40–62, 1986.

[90] P.A. Straub and E.J. Ostertag. Semantics of the Extensible Description Formalism. Technical Report CS-TR-2561, UMIACS-TR-90-137, University of Maryland, Department of Computer Science, College Park, MD 20742, November 1990.

[91] P.A. Straub and E.J. Ostertag. EDF: A formalism for describing and reusing software experience. In *International Symposium on Software Reliability Engineering*, pages 106–113, Austin, Texas, May 17–18 1991.

[92] Pablo A. Straub. *The Nature of Bias and Defects in the Software Specification Process.* PhD thesis, Computer Science Department, University of Maryland, 1992.

[93] E.H. Thompson. *ALA glossary of library terms.* American Library Association (1943), 1943.

[94] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the Conference on functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[95] C.J. van Rijsbergen. *Information Retrieval.* Stoneham, MA: Butterworths, 2nd edition, 1979.

[96] S. Wartik and R. Prieto-Díaz. Criteria for comparing reuse-oriented domain analysis approaches. *Software Engineering and Knowledge Engineering*, September 1992.

[97] P. Winston. *Artificial Intelligence.* Addison Wesley, 2nd edition, 1984.

[98] M. Wood and I. Sommerville. An information system for software components. *ACM SIGIR Forum*, 22(3), Spring/summer 1988.

[99] S.N. Woodfield, D.W. Embley, and D.T. Scott. Can programmers reuse software? *IEEE Software*, 4(4):52–59, July 1987.