EDF: A Formalism for Describing and Reusing Software Experience

Pablo A. Straub
Eduardo J. Ostertag
Computer Science Department
University of Maryland
College Park, MD 20742

Abstract
One approach to achieve high levels of reuse of software experience is to create a database with information on software products, processes, and measurements. This paper presents the Extensible Description Formalism (EDF), a language to describe these databases. EDF is a generalization of the faceted index approach to classification [9]. Objects in EDF can be described in terms of different sets of facets and other object descriptions. Classification schemes are easy to extend by adding new attributes or refining existing ones. EDF has been implemented and used to classify a large software library. In this paper, we present the EDF approach to classification; the development of a classification of data structure operations and packages; and a software defect classification scheme used to describe, explain, and predict defects in data structure packages.

1 Introduction
Software reuse has been claimed to be one of the most promising approaches to improve both productivity and quality in software development [2]. Reuse is not limited to software components: processes, experience, and lessons learned should also be consciously reused. To achieve high reuse we need an organized software library in which both reusable components and development histories are classified according to several domain-dependent criteria.

Different classification approaches have been proposed for organizing software libraries. The CATALOG System [5] extracts keywords from software documentation and creates an index using these keywords. This approach is simple, but conveys no semantic information. Semantic-net based systems [11, 4] are powerful, but they support a narrow application domain.

The faceted index approach [9] relies on a predefined set of facets or attributes defined by experts. Facets and associated sets of terms form a classification scheme for describing components. Component descriptions can be viewed as records where each field (facet) must have a value selected among a finite set of terms. Facets have associated conceptual graphs defining similarity of terms and in turn similarity between software components. The faceted approach is reported to be very effective in retrieving components, but the construction of conceptual graphs is labor intensive. The AIRS system [8] integrates the concepts associated with the faceted index and semantic-net based approaches to reduce the rigidity and the laborious creation of a semantic structure.

One of the main restrictions imposed by a faceted approach to classification is that each object must be described using all facets. This forces component descriptions to use facets that are meaningless to the particular component. It 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.

The Extensible Description Formalism (EDF) is based on an extension to faceted classification. It has facilities to define facets, terms, and component descriptions, but has none of the restrictions explained above. Descriptions may be defined in terms of other descriptions and in terms of different sets of facets. The formalism is extensible in the sense that characterizations that are initially very coarse can be refined to the point where they become specifications. EDF is a general classification language that can be useful in other domains (e.g., software defect classification).

This paper presents a method based on EDF that facilitates the representation of software defect experience. The following section explains the EDF approach to classification. Section 3 presents the EDF language by developing a database of data structure operations and packages. Finally, Section 4 defines a software defect classification scheme used to describe, explain and predict defects in data structure packages.
2 Foundations of EDF

This section presents the EDF approach to classification. A formal definition of the EDF language is presented elsewhere [10].

2.1 Extending Faceted Classification

The EDF approach to classification overcomes the disadvantages of the faceted approach by extending it as follows.

- Components are replaced by instances that belong to several classes. Instances and classes are defined in terms of some attributes and other classes.
- Facets are replaced by typed attributes. Possible types are: integer, string, term enumerations, classes, and sets of the above. Having instances as attribute values allows to relate different instances.
- 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.
- An integrated type-checked language describes attributes, terms, classes, instances, distances, and their dependencies. The language is based on a formal mathematical model.

2.2 Formal Model of EDF

A set of attributes in an EDF classification scheme defines a multidimensional space \( S \). A point \( p \in S \) represents two different concepts: (1) an instance defined solely in terms of the attributes of \( S \) and (2) the class of all instances whose projection onto \( S \) is \( p \).

In EDF points are described 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 \( \land \) and \( \lor \) for logical and and logical or, respectively (there is no negation in EDF). There are two other basic propositions: 'in class' means that the instance referred belongs to class, and 'has name' means that attribute name has a defined value.

In order to have a first order type system, we distinguish explicitly between classes and instances (since classes are possible types and instances are possible values, an association of classes with instances means that types can be values). To accomplish this, instances are denoted by instance\((E)\) and classes are denoted by class\((E)\) where \( E \) is an expression.

Because expressions can contain disjunctions, they can define a (finite) set of points. An instance denoted by instance\((E)\) is well defined if \( E \) defines a single point. An expression \( E \) defines a single point 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 (i.e., \( E \) can be simplified into a consistent conjunct of assignments) [10].

Expressions are also used to characterize sets of instances defined in the database; \( set(E) \) denotes \( \{i \mid i \in D \cap class(E)\} \), where \( D \) is the set of instances in the database. This is used both to define set-valued attributes by intention, and to make queries to the database.

2.3 Comparing Components

The goal of a classification database system is to facilitate the selection of objects based on some criterion. EDF supports two criteria for selecting candidate objects given a description of a target object: by exact match and by distance between descriptions, where 'distance' is a measure of the dissimilarity of objects. For exact matches, the construct \( set(E) \) already described is used. For distance queries, the construct \( query(E) \) denotes a list of instances in the database sorted by increasing distance to the target.

The concept of distance used in EDF is based on the work on the AIRS software classification system [8], where the distance is an estimate of the effort required to transform a candidate component into the target component.

Distances between terms within a term enumeration are defined by a weighted directed graph, containing one node for each term. The weights represent distances. Distances are assumed to be transitive, in the sense that the distance from term \( t_1 \) to term \( t_2 \) is bounded by the weight of the minimum path from \( t_1 \) to \( t_2 \).

For set values, the distance from a candidate set \( A \) to a target set \( B \) is computed by choosing for each element in \( B \) the element in \( A \) that is more similar; all these distances are added together. Formally, if \( A \) and \( B \) are sets, their distance is

\[
D(A, B) = \sum_{b \in B} \min_{a \in A} D(a, b).
\]

The distance from one instance \( i_1 \) to an instance \( i_2 \) is computed by adding the distances of the corresponding attribute values, plus the creation distance of the values of those attributes defined for \( i_2 \) but not for \( i_1 \). (For attributes not defined for \( i_2 \) any value—or no value—is equally acceptable.) We formalize the above as follows.
\[
D(i_1, i_2) = \sum_{a \in dom(i_2)} \min(C(i_2.a), D(i_1.a, i_1.a))
\]

Where \(\text{dom}(i_2)\) is the set of attributes defined for \(i_2\) and \(i_1.a\) denotes the value of attribute \(a\) for instance \(i_1\). 'C' represents the creation distance function, an upper bound on the distance from any other value to a target. The creation distance for an instance is simply defined as the sum of the creation distances of its attribute values. The creation distance of a set is defined similarly. For term enumerations, its associated graph has an extra node that represents the unspecified value. Edges from this node to other nodes are explicitly declared; edges from other nodes to this node have an implicit weight of zero.

3 Developing a 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.

In this section we develop a small software library to classify operations to manipulate data structures consisting of repeated elements (e.g., stacks, trees, hash tables). For presentation purposes we start with a trivial instance containing only one element.

3.1 Classifying Operations

Booch [3] classifies operations over a data structure in three classes, based on how the structure is accessed: constructors (alter the data structure), selectors (evaluate the data structure), and iterators (visit all elements of the data structure). We can describe this classification scheme by defining the \text{funct} (functionality) attribute.

\text{attribute funct : \{construct, select, iterate\};}

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

\text{attribute timing : \{const, log, linear, quadratic, slow\};}

Using these attributes, we describe four operations for stack manipulation.

\text{push = instance(funct=construct & timing=const);}  
\text{pop = instance(funct=remove & timing=const);}  
\text{top = instance(funct=select & timing=const);}  
\text{newstack = instance(funct=construct & timing=const);}  

3.2 Extending the Taxonomy

The characterization of the functionality of operations presented above is too coarse: we cannot even tell push from pop! We want to refine this characterization by adding more classes. There are at least three approaches to do this: (1) add or replace terms in the type of an attribute, (2) add more attributes, or (3) describe all terms of an attribute using more primitive attributes. We explain these three approaches in turn.

Adding Terms to a Type. We add new terms to attribute \text{funct}. Constructors are further classified into create, insert, and remove operations. The term construct is removed, and its uses are replaced by one of the new terms. Now we can tell push from pop and tell those from newstack. The new library is:

\text{attribute funct : \{create, insert, remove, select, iterate\};}  
\text{attribute timing : \{const, log, linear, quadratic, slow\};}

\text{push = instance(funct=insert & timing=const);}  
\text{pop = instance(funct=remove & timing=const);}  
\text{top = instance(funct=select & timing=const);}  
\text{newstack = instance(funct=construct & timing=const);}  

The drawback of this approach is that instance definitions have 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.

Adding Attributes. In EDF it is possible to define a new attribute to refine the classification of selected instances. Unlike other faceted classification systems, the addition of attributes requires modifying only those instances for which the new attribute is relevant.

In the current example we could describe the implementation language, memory requirements, implementation constraints, number and type of arguments, etc.

Describing Terms of an Attribute. EDF provides a new approach to extend a classification scheme: describe all terms of an attribute using more primitive attributes. We illustrate the process by refining again the \text{funct} attribute.

Functionality is described using three new attributes: \text{access} (whether the data structure is written or only read), \text{target} (which elements are affected), and \text{newsize} (how the number of elements varies).

\text{attribute access : \{write, read\};}  
\text{attribute target : \{left, right, keyed, any, all, none\};}  
\text{attribute newsize : \{increase, decrease, reset, same\};}

These new attributes are used to define each of the terms that belong to the attribute \text{funct}.
create = instance (in constructors & newsize=reset & target=none);
n insert = instance (in constructors & newsize=increase);
remove = instance (in constructors & newsize=decrease);
select = instance (in selectors);
itrate = instance (in iterators);

Where constructors, selectors, and iterators are classes of instances. The class mechanism is used both as an abstraction mechanism and also as an abbreviation for expressions.

classifiers = class (access=write);
selectors = class (access=read & newsize=same);
ititrators = class (target=none);

We now change the type of the attribute funct, because its elements do no longer belong to an enumeration type but to a class of instances, namely the class of instances defined using the new attributes.

attribute funct : class (has access | has target | has newsize);

Since all former terms of the funct attribute 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 database.

Using the extended classification scheme we can define new categories of functionality. For example we can define modify as a possible value of functionality, and we can also describe more specific iterators.

modifiers = class (access=write & newsize=same);
modify = instance (in modifiers);
passive-iter = instance (in iterators & in selectors);
active-iter = instance (in iterators & in constructors);
modify-iter = instance (in iterators & in modifiers);

We define two other attributes for packages: maxsize (limits in the number of elements) and control (support for concurrent access) [3].

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

With these declarations we can define a stack package comprising the operations already described with an extra attribute (parent). The implementation has no preset bound on size and does not provide support for concurrency.

stack = instance (subunits=set (parent=stack) & maxsize=unbounded & control=sequential);

Where push, top, and newstack are similarly redefined. The construct set (parent=stack) denotes the set of all instances defined in the database for which the attribute parent is equal to stack, that is \{push, top, newstack\}.

3.4 Attribute Dependencies

Most faceted classification systems assume independence among facets (attributes), which is usually not the case. Certain dependencies can be explicitly declared in EDF using an assertion. The declaration assertion \(E_1 \Rightarrow E_2\) is a statement meaning \(\text{set}(E_1) \subseteq \text{set}(E_2)\). The EDF system checks that all assertions in the database are satisfied.

Our classification scheme for packages and operations has several semantic dependencies among attributes. For example, for the attributes describing the functionality of an operation, we know that 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.

assertion access=read \Rightarrow newsize=same;
assertion maxsize=reset \Rightarrow target=none;

We also want to specify that attributes maxsize and control are only relevant for packages and that all units that declare a package as their parent must indeed be subunits of the package.

assertion has maxsize | has control \Rightarrow in packages;
assertion in packages \Rightarrow subunits=set (parent=self);

Where the keyword self denotes the instance being analyzed for compliance with the assertion.
3.5 Queries and Distances

In order to query the database for similar objects, 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 actual values used here are based on our intuitive understanding of the domain; others have dealt with the problem of determining relationships between terms \[6, 7\].

```
attribute access = \{write, read\}
            distances = \{write -> write:10, read:8, write -> read:4, read -> write:6\};

attribute newsize = \{increase, decrease, reset, same\}
            distances = \{increase -> increase:10, reset:10, same:8, increase -> decrease:5, same:7, decrease -> increase:8, reset:3, same:6, 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 (minimum path through node decrease). The creation distance for decrease is 10 + 5 = 15 (path through increase). The distance from reset to same is not 10 as declared but 8, the creation distance of same (this inconsistency in the distance clause is checked by the system).

Distances between instances are computed by adding the distances of the corresponding attribute values. For example, the distance from remove to select (i.e., 10) is the distance from write to read (4) plus the distance from decrease to same (6).

```
remove = instance(access=write & newsize=decrease);
select = instance(access=read & newsize=same);
```

Based on this result, the distance from pop to top is 10, because they differ only in the attribute funct, which is defined as remove and select respectively.

Distances between instances are used by the query command to retrieve object descriptions from the database. For example, the following query finds components that are similar to an operation that retrieves an arbitrary element from a data structure (e.g., set) in at most logarithmic time.

```
query(funct=instance(in selectors & target=any) & timing=elog);
```

Consider another example. We want to find a data structure with three operations: one to initialize, one to insert an element, and one to traverse the structure without modifying it; we do not need concurrent control but we do need unbounded size.

```
query(maxsize=unbounded & control=sequential
     & subunit={instance(funct=create),
              instance(funct=insert),
              instance(funct=passive-iter)));
```

In the above query, operations are not fully described; because timing is not specified, any value for timing is equally acceptable.

4 Software Defects

One important quality of software systems is its ability to function without defects. Traditional software construction processes have specific subprocesses to detect defects (e.g., "unit test", "acceptance test"). However, detecting defects 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 and thus reducing the cost associated with its correction. The ability 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 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 defects, we need to characterize the different kinds of defects associated with a particular software environment and project. We also need to understand the relationships between defects associated with a product and its attributes, and to model defects, software systems, and their relations. 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 defects in a particular software environment.

4.1 Characterizing Defects Using EDF

The following classification scheme for defects is based on work by Basili and Rombach [1]. The reader should be aware that this classification scheme is specific to the particular environment we intend to use for our example: different organization environments and software projects would have to tailor the classification to their particular needs.

Kinds of Defects. In general, 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. If a defect is document related, it is called a fault. If it is process related, it is called either a failure or an error: failures are associated with processes that are performed automatically and errors are associated with human processes.

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

```plaintext
attribute entity : (data, process);
attribute creator : (computer, human);
```

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. For example, a failure during the execution of a program is caused by a fault in the program. Faults in a document are 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.

```plaintext
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.

```plaintext
attribute severity : (fatal, critical, noncritical, minor);
```

Defects and the Lifecycle. We are interested in determining when and where a defect enters the system or when it is detected. Because phases in the lifecycle are related to documents (e.g., the requirements phase is related to the requirements document), we use phases to measure time in which errors and failures occur as well as determine the (kind of) document in which a fault occurs. The occurrence attribute relates a defect to the phase on which it enters the system. The detection attribute relates a defect to the phase on which it is detected. We explicitly declare the type phase that is used in these two attributes.

```plaintext
type phase = {requirements, specification, design, coding, unit-test, integration, integration-test, acceptance-test, maintenance, operation};
```

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 in failures associated with the execution of a particular kind software products: implementation of data structures.

```plaintext
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.

```plaintext
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 to know whether something is missing (omission) or something is wrong (commission). We use the fault-mode attribute to distinguish between this two cases.

```plaintext
attribute fault-mode : (omission, commission);
assertion has fault-mode => in faults;
```

Kinds of Errors. Defects on human processes (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 paper. We simply characterize errors by the particular domain that is misunderstood or misused, using the error-kind attribute.

---

1 System failures are also caused by environmental accidents; here we only consider software-related failures.
4.2 Examples

The following examples of defects and their characterizations 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.

\[
\text{fault-1} = \text{instance}(\text{in faults & occurrence=coding & detection=unit-test & severity=critical & cause=error-I & fault-kind=interface})
\]

\[
\text{error-1} = \text{instance}(\text{in errors & severity=minor & error-kind=clerical})
\]

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 reformattting the table. The specific problem is that a code optimizer swapped two statements. These defects are classified as follows.

\[
\text{failure-2} = \text{instance}(\text{in failures & severity=noncritical & occurrence=operation & cause=swapped-stmt & failure-kind=overflow})
\]

\[
\text{swapped-stmt} = \text{instance}(\text{in faults & severity=critical & occurrence=coding & detection=operation & cause=\text{failure-op} & fault-kind=control-flow & fault-mode=commission})
\]

\[
\text{failure-op} = \text{instance}(\text{in failures & occurrence=coding & detection=operation & failure-kind=tool-failure})
\]

4.3 Explaining and Predicting Defects

Having a database with software components, software defects, and their inter-relations 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.

\[
\text{overflow-fail} = \text{instance}(\text{in failures & severity=fatal & occurrence=integration-test & failure-kind=overflow})
\]

We do not know the kind of the fault that caused overflow-fail, so we query the database for faults that have caused similar failures. These faults are retrieved by this query.

\[
\text{query(in faults & occurrence=coding & cause=error-I & failure-kind=interface})
\]

Ultimately, we would like to have some statistical analyses of the retrieved instances; currently EDF lacks this capability, but we plan to extend the language to allow user-defined queries written in a conventional programming language.

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 document attribute for faults is the package in which the fault occurs; the fault-set attribute for packages describes the set of known faults.

\[
\text{attribute document : packages;}
\]

\[
\text{assertion has document => in faults;}
\]

\[
\text{attribute fault-set : set of faults;}
\]

\[
\text{assertion has fault-set => in packages & fault-set=set(document=self})
\]

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.

\[
\text{query(maxsize=bounded & control=sequential & subunits=(hash-create, hash-insert, hash-lookup, hash-delete}})
\]

Where the subunits are assumed to be already defined. 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.
5 Conclusion

The Extensible Description Formalism approach is based on our extension to faceted classification, a method that has been successful in practice for software component classification [3, 9]. The EDF language is a notation to describe domains (i.e., attributes, terms and their distances) and databases in an integrated system. EDF overcomes several limitations of faceted systems: it has a formal semantic definition [10] so it is analyzable; objects can be described partially; classifications are easily extensible and they can represent different kinds of objects; and queries are more general.

EDF has been used to represent classifications in very different domains (i.e., software components, software defects). We developed a prototype implementation of EDF in Common Lisp for a Macintosh II computer. This prototype has been used to build a database with several hundred software components. We are currently developing another implementation in C under Unix that will include a direct-manipulation graphical interface, user-defined measures of distance, multiple distance criteria, and extended logical expressions.

Acknowledgements

We are grateful to our advisors, Marvin Zelkowitz and James Hendler. This research has been supported by NASA Goddard Space Flight Center grant NSG-5123, by Air Force grant AFOSR 90-0031, and the University of Maryland Systems Research Center. P. Straub is on a scholarship from the Catholic University of Chile. E. Ostertag is on a scholarship from ORDEN S.A., Chile.

References


