Automating and Validating Semantic Annotations

Mark Grechanik, Kathryn S. McKinley
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{gmark, mckinley}@cs.utexas.edu

Dewayne E. Perry
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712
perry@ece.utexas.edu

ABSTRACT

Use-case diagrams (UCDs) are widely used to describe requirements and desired functionality of software products. However, UCDs are loosely linked to the source code, and there are no approaches to maintain the correspondence between program variable and types (or program entities) and elements of UCDs.

We offer a novel approach for automating a part of the process of annotating program entities with names of elements from UCDs. Developers first annotate an initial set of a few program entities. Our *LEarning ANnnotations (Lean)* system combines these annotations with run-time monitoring, program analysis, and machine-learning approaches to discover and validate annotations on unannotated entities in Java programs. We evaluate our prototype implementation on open-source software projects and our results suggest that Lean can generalize from a small set of annotated entities to annotate many other entities.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques–Computer-aided software engineering (CASE); I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems

General Terms

Experimentation, Documentation, Algorithms

Keywords

Annotations, semantic concepts, machine learning, program analysis, use-case diagrams, runtime monitoring

1. INTRODUCTION

Use-case diagrams (UCDs) are a leading way to capture requirements for software by describing scenarios in which users and system components communicate to perform desired operations [17]. UCDs are especially valuable when maintaining and reengineering legacy systems since programmers use UCDs to understand systems at a high level [18].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Submission to ASE-2006, September 18-22, Tokyo, Japan
Copyright 2006 ACM 0-89791-88-6/97/05 ...$5.00.

Currently, major software design tools support UCDs as part of designing software systems [26][16]. However, UCDs are loosely linked to the source code, and there are no approaches that help to maintain the correspondence between program variable and types (or program entities) and elements of UCDs.

Program annotations assert facts about programs. They appear as comments in the source code or within special language statements. An annotation may assert that program entities belong to some semantic categories. Program annotations help to catch errors, improve program understanding, and recover software architecture.

One of the major uses of program annotations is to help programmers understand legacy systems. A Bell Labs study shows that up to 80% of programmer's time is spent discovering the meaning of legacy code when trying to evolve it [10], and Corbi reports that up to 50% of the maintenance effort is spent on trying to understand code [9]. In many cases, the meaning can be expressed by annotating program entities with names of elements (i.e., semantic concepts) from UCDs. Thus, the extra work required to annotate programs with semantic concepts from UCDs is likely to reduce future development and maintenance time, as well as to improve software quality.

Annotating programs with semantic concepts is often a manual, tedious, and error prone process especially for large programs. Although some programming languages (e.g., C# and Java) have support for annotations, many programmers do not annotate their code at all, or at least insufficiently. A fundamental question for creating more robust and extensible software is how to annotate program source code with a high degree of automation and precision. Linking semantic concepts from UCDs to program entities with annotations enables programmers to map requirements to the implementation, which is one of the most difficult problems in software engineering.

Our solution, called *LEarning ANnotations (Lean)*, combines program analysis, run-time monitoring, and machine learning to automatically propagate a small set of initial annotations to additional unannotated program entities. The input to Lean is program source code and UCDs. The core idea of Lean is that after programmers provide a few initial annotations of some program entities with semantic concepts from the UCDs, the system will glean enough information from these annotations to annotate much of the rest of the program automatically.

Lean works as follows. After programmers specify initial annotations, Lean instruments a program to perform run-time monitoring of program variables. Lean executes this program and collects the values of the instrumented variables. Lean uses these values along with names of program entities to train its learners to identify entities with similar values and names. Lean's learners then

classify the rest of program entities by matching them with semantic concept annotations. Once a match is determined for an entity, Lean annotates it with the matching semantic concept.

We evaluate our approach on open-source software projects written in Java and obtain results that suggest it is effective. Our results show that after users annotate approximately 6% of the program entities, Lean correctly annotates an additional 69% of program entities in the best case, 47% in the average, and 12% in the worst case, taking less than one hour to run on an application with over 20,000 lines of code.

2. BACKGROUND

Use cases (UCs) are widely used to describe requirements and desired functionality of software products. UCs are expressed with UCDs, an example of which is shown in Figure 1. It is a UCD for the *Vehicle Maintenance Tracker (VMT)* project, an open source Java application that records the maintenance of vehicles [4].

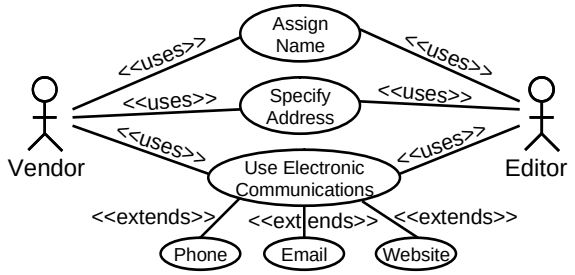


Figure 1: A use-case diagram for the VMT project.

UCs show actors, depicted as human figure icons, and actors who carry out actions that are depicted as ovals. Actors can be human users or components of software products. For example, the actor `Vendor` represents vendors who can be reached using `Electronic Communications`, and the actor `Editor` represents a component that is used to enter and display these `Electronic Communications`. Actors and actions are connected with lines symbolizing relationships between them. Lines are labelled with the description of these relationships, or labels `<<uses>>`, which are often omitted leaving a relationship unlabelled.

A special relation `<<extend>>` between two actions describes associative generalizations, where one action is a specialized case of the other action, which is more general. For example, vendors using `Phone`, `Email`, and `Website` are specialized actions of the more general action of using `Electronic Communications`. Actions connected with relations labelled `<<extend>>` are often implemented in program source code using inheritance.

Actors and actions represent semantic concepts of the domains for which programs are written. Simply stated, semantic concepts are nouns with well-accepted meanings in public or domain-specific knowledge. For example, the noun `Address` is a semantic concept meaning a place where a person or an institution is located. Programmers may introduce variables named `Address`, `Add`, or `S[2]`, all for the `Address` concept. These names are taken from the VMT programs whose code fragments are shown in Figure 2. The name of the variable `S[2]` does not match `Address`, and relating this variable to the `Address` concept is challenging because of the lack of information that helps programmers to identify this relation. While the variable named `Add` partially matches `Address`, it is ambiguous if the program also uses a `Summation` concept for adding numbers.

3. A MOTIVATING EXAMPLE

We use the VMT application as a motivating example throughout this paper. Fragments of the VMT code from three different files are shown in Figures 2(a)– 2(c), and a UCD for the VMT is shown in Figure 1.

A fragment of code from the file `vendors.java` shown in Figure 2(a) contains the declaration of the class `vendors` whose member variables of type `String` are `Name`, `Add`, `Pho`, `Email`, and `Web`. These variables stand for the vendor's name, address, phone number, email, and web site concepts respectively. A fragment of the code from the file `VendorEdit.java` shown in Figure 2(b) contains the declaration of the class `VendorEdit` whose member variables of types `Text` and `TextArea` represent the same concepts. Even though the names of these variables in the class `VendorEdit` are different from the names of the corresponding variables in the class `vendors`, these names partially match. For example, the variable name `Pho` in the class `vendors` matches the variable `PhoneText` in the class `VendorEdit` more than any other variable of this class when counting the number of consecutive matching letters.

This informal matching procedure does not work for the fragment of code shown in Figure 2(c). To what semantic concept does the variable `S`, which is the parameter to the method `addMaintenanceEditor`, correspond? It turns out that the variable `S` is an array of `Strings`, and its elements `S[1]`, `S[2]`, `S[3]`, `S[4]`, and `S[5]` hold values of vendor's name, address, phone number, email, and web site concepts respectively. No VMT documentation mentions this information, and programmers have to run the program and observe the values of these variables in order to discover their meanings.

Lean can automate the process of annotating classes and variables shown in Figures 2(a)– 2(c) with concepts from the UCD shown in Figure 1. We observe that the spellings of some variable names are similar to the names of corresponding concepts, i.e., `Pho`

```
public class vendors {
  private String Name,  Add, Pho, Email, Web;
          ……..……
}
```

(a) File vendors.java.

```
  public class VendorEdit extends InternalFrame {
          private Text NameText;
          private TextArea AddressText;
          private TextArea PhoneText;
          private Text EmailText;
          private Text WebText; }
```

(b) File VendorEdit.java.

```
public void addMaintenanceEditor(String[] S) {
addMaintenanceServices(new String[]{
  ((MaintenanceEdit)Desktop.getSelectedFrame()).
        getName(),  S[4], S[5]});
  }
};
String s = S[1];
if (s.equalsIgnoreCase(""))
    s = "New";
String residence = S[2];
```

(c) File VMT.java.

Figure 2: Code fragments from programs of the VMT project.

– Phone, Add – Address, Web – WebSite, Name – Name, and Email – Email. Lean uses these similarities to match names of variables and concepts, and subsequently to annotate variables and types with matching semantic concepts.

Variable names residence and Address are spelled differently, but they are synonyms. Extended with a vocabulary linking synonymic words, Lean hypothesizes about similarities between words that are spelled differently but have the same meaning. These vocabularies can link domain-specific concepts used by different programmers thereby establishing common meanings for different programs.

By observing patterns in values of program variables Lean can also determine whether they should be annotated with certain concepts. To observe patterns, Lean instruments source code to collect run-time values of the program variables. After running the instrumented program, Lean creates a table containing sample data for each variable. A sample table for the VMT application is shown in Table 1. Each column in this table contains variable name and values it held. Some values have distinct structures. The variable Pho contains only numbers and dashes in the format xxx-xxx-xxxx, where x stands for a digit and the dash is a separator. Values held by the variable Email have a distinct structure with the @ symbol and dots used as separators. Lean learns the structures of values for annotated variables using machine-learning algorithms, and it then assigns the appropriate semantic concepts to variables whose values match the learnt structures.

Email	Pho	Add
tc@abc.com	512-342-8434	Tamara Circle, Austin
mcn@jump.net	512-232-3432	McNeil Drive, Austin
sims@su.edu	512-232-6453	Sims Road, Dallas

Table 1: Values of some variables from the VMT program.

4. THE PROBLEM STATEMENT

Our goal is to annotate entities in Java programs with names of elements from the corresponding UCDs with a high degree of automation and precision. Elements from the UCD correspond to semantic concepts from the domains for which programs are written. Program entities can be annotated with these semantic concepts using the annotation type facility of Java. An example of a Java annotation type declaration and its use is shown in Figure 3. In Java, annotation type declarations are similar to interface declarations [2]. An @ sign precedes the interface keyword, which is followed by the Concept annotation type name that defines fields UCD and Label, for the name of the UCD and the name of one of its element respectively. Annotations consist of the @ sign followed by the annotation type Concept and a parenthesized list of element-value pairs.

Program comprehension is the process of acquiring knowledge about programs [27]. Better program comprehension reduces development time and faults [24]. It was stated that annotating programs with concepts from requirement specifications improves program understanding [6][27]. We assume that the main benefit of annotating entities in Java programs with names of elements from the corresponding UCDs is in better program comprehension. However, we do not conduct an empirical study showing the correlation between annotating program entities with concepts from UCDs and improved program understanding. This is a subject of our future work.

```
public @interface Concept {
    String UCD;
    String Label;
    }

@Concept(UCD="VMT", Label="Vendor")
public class vendors {...}
```

Figure 3: Example of annotating the class `vendors` from Figure 2(a) with the concept `Vendor` from the UCD shown in Figure 1 using the Java annotation type `Concept`.

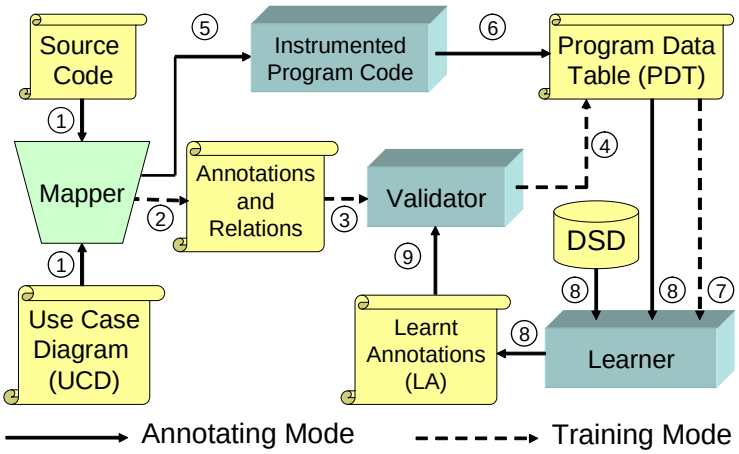
It is not possible to develop a sound and complete approach for automatic annotation of program entities with semantic concepts from UCDs. An annotation approach is sound when program entities are labelled with semantic concepts from UCDs correctly or not labelled at all. False annotations (i.e., labelling program entities with incorrect semantic concepts) are not produced by a sound annotation approach. An annotation approach is complete if it assigns semantic concepts to all program entities. While a sound and complete approach for annotating program entities with the names of the elements from UCDs automatically is desirable, it is, in general an undecidable problem[1].

Since our approach cannot be sound and complete and fully automatic, some guidance from programmers is required. Initially, programmers annotate a small percentage of program entities with names of UCD elements, and our approach uses this information to annotate much of the rest of the program. We do not consider annotating fragments of code or selected statements or lines of code, only program entities. The former is a subject of our future work.

The complexity of the problem suggests the use of *machine learning (ML)* techniques to classify program entities as belonging to semantic concepts from UCDs. If we knew patterns for values or names of program entities for each semantic concept in advance, then we could write pattern-matching routines to assign the names of semantic concepts to these program entities. Even though writing these routines is manual, tedious, and laborious, this approach does not solve our problem. The information about patterns of values and names of program entities may be unavailable, and it is difficult to collect and analyze values and names of program entities and extract exact patterns manually. ML techniques can learn and extract the patterns information related to semantic concepts from UCDs automatically, and then classify program entities using these patterns thus annotating them with names of semantic concepts.

Annotating 100% of program entities correctly and automatically using ML techniques is not realistic. Many reasons exist: machine learning approaches do not guarantee absolute success in solving problems; UCDs representing program design specifications may not match programs; and some concepts may be difficult to relate to program entities due to the lack of modularity. ML approaches are only as good as the training data, and they do not guarantee 100% classification accuracy. Some data are more difficult to classify than other because they are hard-to-analyze (e.g.,

[1]Suppose that values described by some semantic concept are strings generated by some *context-free grammar (CFG)*. One CFG generates strings for some element of a UCD and some other CFG generates strings for some program variable. If strings generated for the semantic concept and the program variable are identical, then the program variable is described by, and consequently can be annotated with the name of this semantic concept. However, determining if two CFGs generate the same set of strings is an undecidable problem [28].



(a) Lean architecture.

1) Programmers annotate the Source Code with concepts from the UCD using the Mapper
2) The Mapper constructs and outputs α, δ, and γ relations in the XML format
3) The Validator validates initial mappings
4) The Validator outputs annotations for training
5) The Mapper instruments and compiles the source code
6) The program runs and its instrumented code outputs the Program Data Table (PDT)
7) Annotated variables and their values from the PDT are supplied to the Learner for training
8) Learner classifies program variables from the PDT and produces learnt annotations (LA) with the help of Domain-Specific Dictionary (DSD)
9) The Validator validates LAs and uses negative examples to retrain the Learner

(b) Lean process description

Figure 4: Lean Architecture and its process.

binary strings of nonprintable characters). It is difficult to achieve a high accuracy of classifiers on variables holding these data since patterns in binary data are inherently complex. For these and other reasons ML approaches may assign concept names from UCDs to program entities incorrectly.

Since our approach is not sound, mistakes are made when learning annotations. In order to improve its precision, program analysis helps to to determine relations among annotated entities. By comparing these relations with corresponding relations between elements in UCDs with which names these entities are annotated, it is possible to detect some false annotations automatically. For example, if a relation is present between two entities in the program code and there is no relation in a UCD between concepts with which these entities are annotated, then possible false annotation warnings should be issued.

5. LEAN ARCHITECTURE AND PROCESS

The architecture for Lean and its process description are shown in Figure 4. The main elements of the Lean architecture are the Mapper, the Learner, and the Validator shown in Figure 4(a). Solid arrows show the process of annotating program entities with names of elements (concepts) from UCDs, and dashed arrows specify the process of training the Learner.

The inputs to the system are program source code and a UCD `(1)`. The Mapper is a tool whose components are Java and XML parsers, program and UCD analysis routines, and an instrumenter. A Java parser produces a tree representing program entities. UCDs are represented in XML format, and the Mapper uses an XML parser to produce a tree representing elements and relations in the UCD `(2)`.

Programmers specify initial annotations as relations between elements from the UCD and program entities from the source code. Internally, annotations are represented as pairs $(t,c) \in \alpha$, where α is the annotation relation, t is a program entity, and c is a concept from a UCD. Programmers can also specify what entities should be excluded from the annotation process. For example, using Lean to annotate an integer variable counting the number of iterations in a loop consumes computing resources while there may not be an appropriate concept for annotating this variable, or annotating it does not warrant the amount of work required.

The Mapper constructs relations using the Mapper's program analysis routines. Relations between elements in UCDs are expressed as pairs $(c_p, c_q) \in \gamma$, where c_p and c_q are concepts from some UCD, and γ is the relation between elements in UCDs. The δ–relation describes relations between program entities, and it includes three relations: between types and types, between types and variables, and between variables and variables. The type-type δ–relations exist between classes connected via inheritance or between classes and interfaces [2]. The type-variable δ–relations exist between variables and types to which these variables are explicitly cast or declared. Finally, the variable-variable δ–relations specify that two variables are used in the same expression. The Mapper outputs all detected relations as the XML data, which is then passed to the Validator.

The Validator checks the correctness of annotations by exploring δ– and γ–relations `(3)`. It uses the heuristics stating that for a δ–relation between annotated entities in the source code there is a γ–relation between the elements in a UCD with whose names these program entities are annotated. This heuristic is based on the observation that relations between elements of UCDs are often preserved in the program code. For example, the `<<extend>>` relation between elements of a UCD is often implemented using the inheritance between classes implementing the UCD elements in the program code. The output of the Validator is a list of flagged annotations that can be false, and they should be verified by programmers manually. The Learner can improve its predictive capabilities by using the results of this verification `(4)`. That is, if the Learner assigns an annotation incorrectly, them it can be retrained on this negative example to improve the performance of the Learner.

The Mapper instruments the source code to record run-time values of program variables `(5)`. Runtime logging is added after the statements and expressions in which the monitored variables are assigned values. Lean's data flow analysis framework locates variable definitions and traces the uses of these variables until either the end of the scope for the definitions, or the definition of new variables with the same names that shadow previous definitions. Only distinct values of the monitored variables are collected. Values of annotated variables are used to train the Learner, which uses the

[2]We use the term `type` as a substitute for terms `class` and `interface`, and vice versa.

learnt information to classify unannotated variables.

After instrumenting the source code, the Mapper calls a Java compiler to produce an executable program (5). Then, the program runs, storing names and the values of program variables in the *Program Data Table (PDT)* (6). Both the Validator and the runtime logger code output PDT in the *Attribute Relation File Format (ARFF)* file format. ARFF serves as an input to the Learner (7), which is based on a machine-learning Java-based open source system WEKA [30]. Once the Learner is trained, it classifies unannotated program entities that are supplied to the Learner as the columns of the PDT. Lean classifies unannotated variables by obtaining their runtime values and their names and types by analyzing their names, and supplying them to the Learner which emits predictions for concepts with which these program entities should be annotated. In addition, *domain-specific dictionaries (DSDs)* increase the precision of the classification. The output of the Learner is a set of learnt annotations (LAs) (8). These LAs are sent to the Validator to check their correctness (9). The Validator sends corrected annotations to the Learner for training to improve its accuracy (4). This continuing process of annotating, validating annotations, and learning from the validated annotations makes Lean effective for long-term evolution and maintenance of software systems.

6. LEARNING ANNOTATIONS

This section shows how Lean learns and validates annotations. We explain the learning approach and describe the learners used in Lean. Then we show how to extend the Learner to adapt to different domains.

6.1 The Learning Approach

We treat the automation of the program annotation process as a classification problem: given n concepts from UCDs and a program entity, which concept matches this entity the best? Statistical measures of matching between entities and concept names from UCDs are probabilistic. The Learner classifies program entities with the probabilities that certain concept names can be assigned to them. A Lean classifier is trained to classify an unannotated entity based on the information learnt from the annotated entities.

Lean has its roots in the *Learning Source Descriptions (LSD)* system developed at the University of Washington for reconciling schemas of disparate XML data sources [12, 13]. The purpose of LSD is to learn one-to-one correspondences between elements in XML schemas. Since it is difficult to find a learning algorithm that can deliver consistently good results for different types of input data, Lean employs the LSD multistrategy learning approach [20, 13], which organizes multiple learners in layers. The learners located at the bottom layer are called *base learners*, and their predictions are combined by *metalearners* located at the upper layers.

In the multistrategy learning approach, each base learner issues predictions that a program entity matches a UCD element (concept) with some probability. A metalearner combines these predictions by multiplying these probabilities by weights assigned to each base learner and taking the average for the products for the corresponding predictions for the same program entity.

The Lean learning algorithm consists of two phases: the training phase and the annotating (classifying) phase. The training phase improves the ability of the learners to predict correct annotations for program entities. Trained learners classify program entities with concept names, and based on these classifications, Lean annotates programs. The accuracy of the classification process depends upon successful training of the Learner.

During the training phase, weights of the base learners are adjusted and probabilities are computed for each learner using the runtime data for annotated entity. Then, during the classification step the previously computed weights are used to predict concepts for unannotated entity. Weights of the learner are modified using regression [21].

6.2 The Learners

There are three types of base learners used in Lean: a name matcher, a content matcher, and a Bayes learner [12, 21]. Even though many different types of learners can be used with the multistrategy learning approach, we limit our study to these three types of learners since they proved to give good results when used in the LSD.

Name matchers match the names of program entities with names of UCD elements. The name matching is based on Whirl, a text classification algorithm based on the nearest-neighbor class [8]. This algorithm computes the similarity distance between the name of a program entity and a UCD element. This distance should be within some threshold whose value is determined experimentally.

Whirl-based name matchers work well for meaningful names especially if large parts of them coincide. Their performance worsens when names are meaningless or consist of combinations of numbers, digits, and some special characters (e.g., underscore or caret).

Content matchers work on the same principles and use the same algorithm (Whirl) as name matchers. The difference is that content matchers operate on the values of variables rather than their names. Content matchers work especially well on string variables that contain long textual elements.

Finally, Bayes learners, particularly the Naïve Bayes classifier, are among the most practical and competitive approaches to classification problems [21]. Naïve Bayes classifiers are studied extensively [14, 21], so we only state what they do in the context of the problem that we are solving here. For each program entity the Naïve Bayes classifier assigns some names of UCD elements c_k, such that the probability that this entity belongs to the concept c_k, is maximized. Bayes learners perform well when classifying numerical as well as string variables, and they compensate for the deficiencies of the Whirl algorithm when classifying variables holding numerical values and program entities whose names contain mixed characters.

6.3 Extending the Learner

Domains use special terminologies whose dictionary words mean specific things. Programmers use domain dictionaries to name variables and types in programs written for these domains. For example, when word "dice" is encountered in a value of some variable of a program written for a semiconductor domain, this variable may be annotated with the circuit concept. Many domains have dictionaries listing special words, their synonyms, and explaining their meanings.

In general, the annotation process is dependent on relations between specific benchmarks and DSDs. The quality of the content of a DSD may significantly improve the annotation process for some applications while other applications may not be affected positively especially if they belong to a different domain. Our goal is to show that DSDs improve the quality of the annotation process for a given domain, and expanding the scopes and the precision of DSDs leads to more precise annotations.

Lean incorporates the knowledge supplied by these dictionaries. Each concept in these dictionaries has a number of words that are characteristic of this concept. If a word from the dictionary is encountered in a value or the name of a program entity, then this entity may be classified and subsequently annotated by some concept to which this dictionary word belongs. We use a simple heuristic to change the probabilities that variables should be annotated

with certain UCD concept names. If no dictionary word is encountered among the name or the values of an entity, then its probabilities computed by Lean learners for this variable remain unchanged. Otherwise, if a word belongs to some concept, then the probability that the given entity, v, belongs to this concept is incremented by some small real number Δ_p, i.e., $p_{concept}(v) = p_{concept}(v) + \Delta_p$. We choose this number experimentally as `1/|W|`, where `|W|` is the number of words in the DSD. If the resulting probability is greater than `1.0` after adding Δ_p, then the probability remains `1.0`.

6.4 Learning Conditional Annotations

Often a program entity can be classified with more than one concept. Consider a fragment of code shown in Figure 5. The `while` loop iterates over the integer variable `counter` whose value modulo two serves as input to the method `GetAttribute`. This method iterates through some dataset and returns `String` type values which are assigned to the variable `var`. Suppose that the value returned by the method `GetAttribute` belongs to the concept `Address` when the value of the variable `counter` is even, and to the concept `Email` when the value is odd. It means that the variable `var` should be annotated with these two concepts. However, these annotations are conditional upon the value of the variable `counter`. For example, annotating the variable `var` with two concepts `A` and `B` should indicate that the concept `A` is valid when the value of the expression `counter%2` is equal to zero, and the concept `B` is valid when the value of this expression is equal to one.

```
int counter = 0; String var=null;
while((counter++) < SOMEINTEGER) {
    var = GetAttribute(counter%2);
}
```

Figure 5: Example of the code fragment whose variable `var` can be annotated with more than one concept.

Currently, Lean supports multiple annotations for variables whose values are assigned in loops, and which are conditional upon the `counter`–like temporary variables. Temporary variables that are incremented by a predictable amount each time through the loop, are called *induction variables* [23]. Examples are variables whose definitions within the loop are of the form `counter = counter + c`, where `c` is a constant called the *loop invariant*. In our example the value of `c` is one. Lean combines induction variable values with the values of other variables to train the Learner, and subsequently classify the unannotated variables that depend on these induction variables.

We observed that programs where the same entities were assigned values of different semantic concepts required more analysis and Lean was less accurate in annotating these entities. Part of the problem was to determine under which condition values of different semantic concepts were assigned. It was equally difficult for programmers to understand the logic of such programs, leading us to believe that is it a poor design to assign values of different semantic meanings to the same program entities.

7. VALIDATING ANNOTATIONS

This section describes how to validate annotations assigned by the Learner. First, we describe the idea upon which the Validator is based. Then, we give the algorithm for validating annotations. Finally, we evaluate the time and space complexities of this algorithm.

7.1 The Idea

The idea of the validation algorithm is to guess annotations for unannotated program entities using existing annotations and δ– and γ–relations. Recall that γ is the relation between elements in UCDs, δ specifies relations between program entities, and α is the annotation relation. Annotations for unannotated entities are guessed by composing these relations. Relations δ and α can be composed if the second component of a δ–relation matches the first component of some α–relation. Relations α and γ can also be composed if the second component of a pair from the α–relation matches the first component of some pair from the γ–relation. We can write the composition rules as $\sigma = \delta \circ \alpha$, $\sigma = \alpha \circ \gamma$, and $\sigma = \delta \circ \alpha \circ \gamma$. Relation $(t, c) \in \sigma$ suggests that program entities, t may be annotated with concept names, c. These suggested annotations are not added to programs, they are used only to validate annotations determined by the Learner.

The validation algorithm uses the heuristics stating that for a δ–relation between annotated entities in the source code there is a γ–relation between the elements in a UCD with whose names these program entities are annotated. This heuristic is based on the observation that relations between elements of UCDs are often preserved in the program code. Suppose a programmer determines that some program entity t_n should be annotated with some concept c_p from some UCD. This annotation can be written as the α–relation $\alpha(t_n, c_p)$. Suppose that there are relations $\delta(t_m, t_n)$ and $\gamma(c_p, c_q)$ specifying that program entities t_m and t_n are related in the program, and concepts c_p and c_q are also related in some UCD. By composing these relations $\delta(t_m, t_n) \circ \alpha(t_n, c_p) \circ \gamma(c_p, c_q)$ we obtain the new relation $\sigma(t_m, c_q)$ suggesting that the program entity t_m can be annotated with the concept c_q.

After running the Learner it may suggest two α–relations: $\alpha(t_m, c_q)$ and $\alpha(t_m, c_w)$. Since there is a corresponding relation $\sigma(t_m, c_q)$, the learned relation $\alpha(t_m, c_q)$ is validated. However, the second learned relation $\alpha(t_m, c_w)$ is flagged as possibly false since there is no corresponding σ–relation. Programmers should review flagged annotations and reject them if they are proved to be false.

7.2 The Inference Algorithm

The algorithm `InferAnnotations` for inferring σ–relations is given in Algorithm 1. Its inputs are δ–, α–, and γ–relations. The α–relations contains annotations provided by programmers (i.e., initial annotations) and the outputs from the previous runs of the Learner, and the correctness of these annotations was confirmed by prior inspections. The algorithm iterates through all α– and δ–relations in the first two `for`–loops to find pairs of α–relations whose first component (program entity) is the same as the second component in some δ–relation pair. The composition of α– and δ–relations with matching components gives elements of the σ–relation.

Then, the inner `for`–loop iterates through all γ–relations to find pairs that can be composed with the pair from the α–relation from the outer `for`-loop. That is, the first component of the pair from the γ–relation should be the same as the second component of the pair from the α–relation. The result of this composition is a new pair of the σ–relation that suggests the annotation `q` for the program entity `s`.

7.3 The Validation Algorithm

The `Validate` algorithm shown in Algorithm 2 checks for the correctness of assigned annotations. The key criteria is to check if pairs from the learned α–relation set are present in the σ–relation set. The input to this algorithm is the set of α–relation pairs learned by the Learner and the set of σ–relation pairs obtained by the al-

Algorithm 1 The InferAnnotations procedure

```
InferAnnotations( δ, α, γ )
σ ↦ ∅
for all (a, b) ∈ α do
  for all (s, t) ∈ δ do
    if t = a then
      σ ↦ σ∪ (s, b)
      for all (p, q) ∈ γ do
        if p = b then
          σ ↦ σ∪ (s, q)
        end if
      end for
    end if
  end for
end for
```

gorithm `InferAnnotations` shown in Algorithm 1. Each α–relation has a color associated with it, which is initially set to `red`. The `red` color means that a given α–relation is not correctly annotated (i.e., false), and the `green` color means that the learned annotation is correct.

Algorithm 2 The validation procedure

```
Validate( σ, α )
for all (a, b) ∈ α do
  color(a, b) ↦ red
  for all (c, d) ∈ σ do
    if a = c ∧ b = d then
      color((a, b)) ↦ green
    end if
  end for
end for
for all (a, b) ∈ α do
  if color(a, b) = red then
    print: (a, b) is a possibly false annotation
  end if
end for
```

This algorithm does not specify what the correct annotation of a program entity is or what caused the error in program annotation. In fact, annotation errors may be caused by incorrect UCD, errors in program source code, or both. The last `for`-loop iterates through all α–relations, checks the colors, and prints α–relation pairs that are colored `red` as potentially incorrect.

7.4 The Computational Complexity

Suppose a program has n program entities and a UCD has m elements. Then it is possible to build $n(n-1)$ δ–relations, $m(m-1)$ γ-relations, and nm α-relations. Thus, the space complexity is $O(n^2 + m^2 + nm)$. The time complexity is determined by three `for`–loops in the `InferAnnotations` algorithm, which iterate over all relations. Considering all other operations in the algorithms taking $O(1)$, the time complexity is $O(n^3m^3)$.

8. THE PROTOTYPE IMPLEMENTATION

Our prototype implementation included the Mapper, the Validator, and domain-specific dictionaries. The Mapper is a GUI tool written in C++ that includes the EDG Java front end [1] and an MS XML parser. The Mapper contains less than 2,000 lines of code. Its program analysis routines recover δ–relations between program entities and γ–relations between elements of the UCD. The Mapper contains the instrumenter routine that adds the logging code to the original program outputting runtime values of variables into the PDT in ARFF format.

The Validator is written in C++ and contains less than 1,000 lines of code. Its routines implement the `InferAnnotations` and `Validate` algorithms as described in Section 7. The Validator takes its input in XML format and outputs a PDT in ARFF format. The input XML file contains annotations specified by users along with program entities excluded from the annotation process, and δ– and γ– relations. The output ARFF file contains names of program entities and concepts assigned to them.

9. EXPERIMENTAL EVALUATION

In this section we describe the methodology and provide the results of experimental evaluation of Lean on open-source Java programs.

9.1 Subject Programs

We experiment with a total of seven Java programs that belong to different domains. `MegaMek` is a networked Java clone of `BattleTech`, a turn-based sci-fi boardgame for two or more players. `PMD` is a Java source code analyzer which, among other things, finds unused variables and empty catch blocks. `FreeCol` is an open version of the `Civilization` game in which players conquer new worlds. `Jetty` is an open source HTTP Server and Servlet container. The `Vehicle Maintenance Tracker (VMT)` tracks the maintenance of vehicles. The `Animal Shelter Manager (AMS)` is an application for animal sanctuaries and shelters that includes document generation, full reporting, charts, internet publishing, pet search engine, and web interface. Finally, `Integrated Hospital Information System (IHIS)` is a program for maintaining health information records.

9.2 Selecting Input Data

A bias in choosing input data to the subject programs affects the results of our experiments. Selecting homogeneous input data is likely to increase the accuracy with which the Learner classifies data, and selecting heterogeneous data may reveal the limitations of the Learner in extracting patterns for different data. However, completely random heterogeneous input data may not be good representatives of the program inputs. Our goal is to select data inputs for subject programs as close as possible to the real-world data.

Input data for the AMS application were extracted from the worldwide animal shelter directory. Input data for the VMT application were taken from the database of the Cobalt Group company that builds solutions for the automotive retail marketplace. PMD source code analyzer was run on Java programs taken from samples supplied with the Java Development Kit. Jetty served web pages copied and saved from news information web sites. IHIS used data from the American Hospital Directory and other hospital databases available from the Internet. Input data for the MegaMek and FreeCol games were supplied with the applications as well as generated when playing these games.

9.3 Methodology

To evaluate Lean, we carried out two experiments to explore how effectively Lean annotates programs and how its training affects the accuracy of predicting annotations.

In one experiment, a group of graduate students created a DSD and a UCD for each subject program. These students were not familiar with the subject programs, and acquired information about them by reading their documentation and comments in the source code. Then, these users annotated a small subset of variables for each program manually using the Mapper, and then run Lean to an-

Program	Size of DSD, words	Lines of code	Number of concepts	Num of entities	Running Time, min	User annots, %	Lean annots w/o DSD	Lean annots with DSD	Accu–racy, %
Megamek	60	23,782	25	328	56	10%	58%	64%	64%
PMD	20	3,419	12	176	28	7.4%	23%	34%	35%
FreeCol	30	6,855	17	527	39	4.7%	56%	73%	79%
Jetty	30	4,613	6	96	32	12.5%	42%	81%	52%
VMT	80	2,926	8	143	25	5.6%	65%	72%	83%
ASM	60	12,294	23	218	43	5.5%	57%	79%	87%
IHIS	80	1,883	14	225	18	8%	53%	66%	68%

Table 2: Results of the experimental evaluation of Lean on open source programs.

notate the rest of the program. The goal of this experiment is to determine how effective Lean is in annotating variables for programs of different sizes that belong to different domains. Each annotation experiment is run with and without a DSD in order to study the effect of the presence of DSDs on the quality of Lean annotations. A different group of graduate students validated the correctness of annotations assigned by Lean.

We measure the number of variables annotated by Lean as well as the number of annotations rejected by the validating algorithm. The number of variables that Lean can possibly annotate, `vars`, is `vars = total - (excluded + initial)`, where `total` is the total number of variables in a program, `excluded` is the number of variables excluded from the annotation process by the user, and `initial` is the number of variables annotated by the user. Lean's accuracy ratio is computed as `accuracy = (vars - rejected)/vars`, where `rejected` is the number of annotations produced by Lean and rejected by the validating algorithm.

The goal of the other experiment is to evaluate the effect of training on the Lean's classification accuracy. Specifically, it is important to see the amount of training involved to increase the accuracy of annotating programs. Training the Learner is accomplished by running instrumented programs on input data and collecting values for program variables. These values along with names of program entities are used to train the Learner. Each training run is done with a distinct input data set. Depending on the number of training runs Lean can achieve certain accuracy in classifying data on which it was not trained. If the Learner should be trained continuously to maintain even low accuracy, then performance-demanding applications may be exempt from our approach. On the contrary, if a program should run a reasonable number of times with distinct data sets for training to achieve good classification accuracy, then our approach is practical and can be used in the industrial setting.

9.4 Results

Table 2 contains results of the experimental evaluation of Lean on the subject programs. Its columns contain the name of a program, the size of the DSD, the number of lines of code in the subject, the number of concepts in the UCD, the number of program entities that Lean could potentially annotate, the Lean running time in minutes, the percentage of initial annotations computed as ratio `initial/total`, where `total` is the total number of variables in a program, and `initial` is the number of variables annotated by users. The next two columns compare the percentage of total annotations without and with the DSD. The last column of this table shows Lean's accuracy when used with DSDs.

The highest accuracy is achieved with programs that access and manipulate domain-specific data rather than general information. The lowest level of accuracy was with the program PMD which analyzes Java programs whose code does not use terminologies from any specific domain. The highest level of accuracy was achieved with the programs ASM and VMT which are written for specific domains with well-defined terminologies, and whose entity names are easy to interpret and classify. Our experiments show that the `Validate` algorithm performs well in practice for the majority of cases by discarding up to 83% of incorrectly assigned annotations.

The next experiment evaluates the accuracy of the Learner. For each subject application we collected up to 600 distinct input data sets. We trained the Learner for each subject applications on the subset of the input data, and used Lean to annotate program entities using the rest of the input data. Figure 6 shows the dependency of classification accuracy from the number of distinct training samples used to train the Learner. When annotating the AMS application, the Learner achieved the highest accuracy, close to 90%. This accuracy was achieved when the number of distinct training samples reached 500. The results of this experiment show that applications need to be run only few hundred times with distinct input data in order to train the Learner to achieve good accuracy. Since most applications are run at least several thousand times during their testing, using Lean as a part of application testing to annotate program source code is practical. Potentially, if a low-cost mechanism [5] is applied to collect training samples over the life time of applications, then Lean can maintain and evolve program annotations with evolving programs.

Next, we used the Learner trained for the VMT application to annotate entities in other applications. This methodology is called *true-advice* versus *self-advice* which uses the same program for training and evaluation. Figure 7 shows the percentage of variables that the Lean Learner annotates with self-advise (left bar) versus the true-advice annotations (right bar) when the Learner is trained on the VMT application. This experiment shows that Lean can be trained on one application and used to annotate other programs if they operate on the same domain-specific concepts. ASM and IHIS

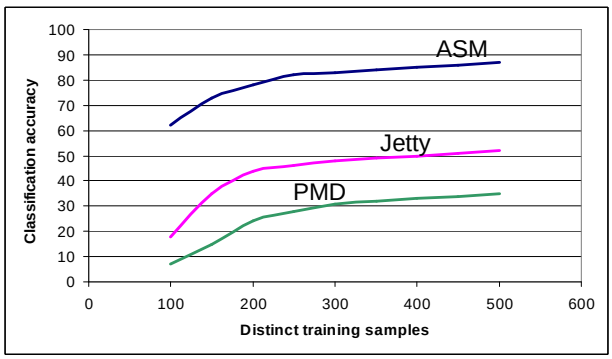


Figure 6: The graph of the accuracy of the Lean Learner.

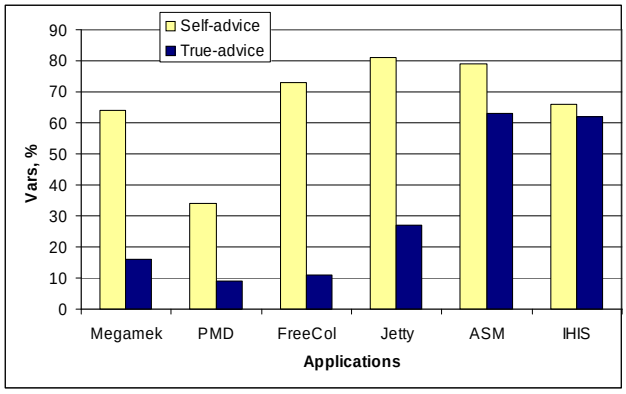


Figure 7: Percentage of program entities that the Lean Learner annotates with self-advise (left bar) versus the true-advise annotations (right bar) when the Learner is trained on the VMT application.

share common concepts with the VMT application, and it allows learners to be trained and used interchangeably thus achieving the high degree of automation in annotating program variables.

Finally, we determine how choosing entities for initial annotations randomly and increasing their number affects the accuracy and the coverage of the Learner. In our experience programmers tend to choose program entities for initial annotations with which they are already familiar. These entities tend to have descriptive names reflecting the concept, and it eases the selection process for initial annotations. However, choosing entities this way for initial annotations may affect the outcome of the experiment. Patterns in names and values in some program entities may be detected faster by some classifiers and used more effectively to annotate some program entities than the others that match different patterns. Also, increasing the number of entities chosen for initial annotations leads to a better coverage of the Lean Learner. At the same time choosing more entities for initial annotations makes the Lean process more expensive. So the question is what percentage of the total program entities should be chosen for initial annotations to give an optimal coverage?

The results shown in Table 2 come from experiments in which programmers selected entities for initial annotations based on how easy it was to map these entities to semantic concepts from UCDs.

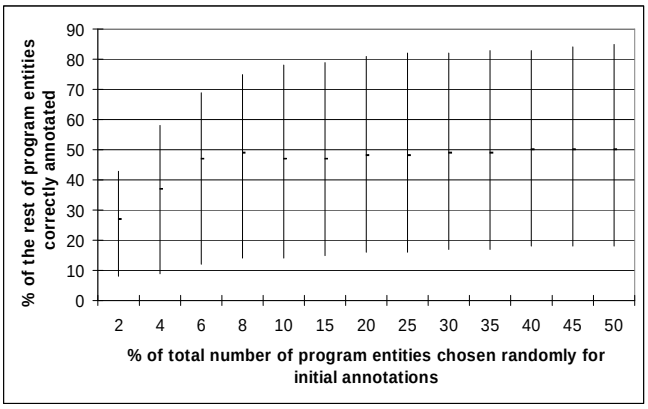


Figure 8: Dependency of the classification coverage and accuracy from the percentage of randomly selected program entities for initial annotations.

As Lean annotates other entities, their annotations are collected, validated, and stored. We perform the same experiments with choosing initial annotations randomly using the annotations from the previous runs of Lean. This way we reduce the dependency of the annotation process from particular entities chosen for initial annotations.

The results of this experiment are shown in Figure 8 in the stock-price-style graph. The horizontal axis shows the percentage of the total number of program entities chosen randomly for initial annotations, and the vertical axis shows the percentage of the rest of program entities correctly annotated (the correctness of Lean annotations is validated by the `Validator` and approved by programmers who inspect them). For each number of initially annotated program entities we run a series of experiments in each of which these entities were chosen randomly. The vertical lines on this graph show the maximum and minimum numbers of correctly annotated entities and the average number is shown by the horizontal mark on the vertical lines. While the gap between the minimum and the maximum numbers of the correctly learned annotations is large, the average shows that in order to get a good annotating coverage it is sufficient to annotate less than ten percent of program entities.

10. RELATED WORK

Related work on program annotations falls into two major categories: systems that derive annotations as invariants or assertions from program source code, and systems that automate the annotation process for software-unrelated artifacts.

A technique for annotating source code with XML tags describing grammar productions is based on modified compilers for C, Objective C, C++ and Java programs [25]. Like our research, this parse tree approach uses grammars as external semantic relations to guide the automatic annotation of program code. However, this approach is tightly linked to grammars that do not express domain-specific concepts and relations among them. By contrast, our solution operates on UCDs describing domains for which programs are written rather than grammars of languages in which programs are written.

Various tools and a language for creating framework annotations allow programmers to generate annotations using frameworks' and example applications' source code, automate the annotation process with dedicated wizards, and introduce coding conventions for framework annotations languages [29]. Like our research, concepts from framework description diagrams are used to annotate program source code. By contrast, the Lean's goal is to automate the annotation process rather then introduce a language that allows programmers to enter annotations manually using some tools.

`Calpa` is a system that generates annotations automatically for the `DyC` dynamic compiler by combining execution frequency and value profile information with a model of dynamic compilation cost to choose run-time constants and other dynamic compilation strategies [22]. Calpa is shown to generate annotations of the same or better quality as those found by a human, but in a fraction of the time.

`Daikon` is a system for automatic inferences of program invariants that is based on recording values of program variables at runtime with their following analysis [15]. Typically, print statements are inserted in C source code to record the values of parameters to functions and other variables before and after functions are called. Then, these values are analyzed to find variables whose values are not changed throughout the execution of certain functions. These variables constitute invariants that annotate respective functions.

Like our research, `Calpa` and `Daikon` systems automate the

generation of annotations and the user is relieved from a task that can be quite difficult and highly critical. Rather than identifying run-time constants and low-level code properties that are extracted from the source code, Lean enables programmers to automate the process of annotating programs with arbitrary semantic concepts.

A number of systems automate the annotation process for software-unrelated artifacts. Techniques used in these systems are similar to ones that Lean uses. `OpenText.org` project presents an interesting approach in automating text annotations [3]. It is a web-based initiative for annotating Greek texts with various linguistic concepts. Similar to Lean, the result of the annotation is kept in an XML format which is later converted in the ARFF format required by WEKA. Like in Lean, machine learning algorithms are used to classify text and assign annotations based on the results of the classification. The major difference between our approach and the `OpenText.org` is that the latter is used to annotate texts while the former annotates program code.

An automated annotation system for bioinformatics analysis is applied to existing genom sequences to generate annotations that are compared with existing annotations to illustrate not only potential errors but also to detect if they are not up-to-date [7]. Unlike Lean, this system cannot be applied to programs, however, Lean can use its ideas to further improve the validation of existing annotations as programs evolve.

A semi-automatic method uses information extraction techniques to generate semantic concept annotations for scientific articles in the biochip domain [19]. This method is applied to annotate textual corpus from the biochip domain, and it was shown that adding semantic annotations can improve the quality of information retrieval.

11. CONCLUSION

The contributions of this paper are the following:

- a system called Lean that automates program annotation process and validates assigned annotations;
- Lean implementation in C++ that uses open source machine learning tools and Java and XML parsers;
- a novel algorithm for validating annotations;
- our experiments show that after users annotate approximately 6% of the program variables and types, Lean correctly annotates an additional 69% of variables in the best case, 47% on the average, and 12% in the worst case.

We believe that our approach has significant potential. Even though Lean is currently implemented to work with Java programs, there are no inherent limitations against using Lean for other languages. Since requirements can be expressed in other forms besides UCDs, Lean can be extended to work with other representations of requirements diagrams. Our experience suggests that Lean is practical for many applications, and its algorithms are efficient and effective.

12. REFERENCES

[1] Edison design group. *http://www.edg.com*.

[2] Jsr 175: A metadata facility for the java programming language. *http://jcp.org/en/jsr/detail?id=175*.

[3] The opentext.org project. *http://www.opentext.org*.

[4] Vehicle maintenance tracker. *http://vmt.sourceforge.net/*.

[5] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, pages 168–179, 2001.

[6] V. R. Basili and H. D. Mills. Understanding and documenting programs. *IEEE Trans. Software Eng.*, 8(3):270–283, 1982.

[7] K. Carter and A. Oka. Bioinformatics issues for automating the annotation of genomic sequences. *Genome Informatics*, 12:204–211, 2001.

[8] W. W. Cohen and H. Hirsh. Joins that generalize: Text classification using WHIRL. In *KDD*, pages 169–173, 1998.

[9] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[10] J. W. Davison, D. Mancl, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, 5(2):44–54, 2000.

[11] N. Dershowitz and Z. Manna. Inference rules for program annotation. *IEEE Trans. Software Eng.*, 7(2):207–222, 1981.

[12] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD*, 2001.

[13] A. Doan, P. Domingos, and A. Y. Halevy. Learning to match the schemas of data sources: A multistrategy approach. *Machine Learning*, 50(3):279–301, 2003.

[14] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification.* Wiley-Interscience, October 2000.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE*, pages 213–224, 1999.

[16] J. E. Hansen and C. Thomsen. *Enterprise Development with Visual Studio .NET, UML, and MSF.* Apress, May 2004.

[17] I. Jacobson. Object oriented development in an industrial environment. In *OOPSLA*, pages 183–191, 1987.

[18] I. Jacobson and F. Lindström. Re-engineering of old systems to an object-oriented database. In *OOPSLA*, pages 340–350, 1991.

[19] K. Khelif and R. Dieng-Kuntz. Ontology-based semantic annotations for biochip domain. In *EKAW*, 2004.

[20] R. Michalski and G. Tecuci. *Machine Learning: A Multistrategy Approach.* Morgan Kaufmann, February 1994.

[21] T. M. Mitchell. *Machine Learning.* McGraw-Hill, March 1997.

[22] M. Mock, C. Chambers, and S. J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *MICRO*, pages 291–302, 2000.

[23] R. Morgan. *Building an optimizing compiler.* Digital Press, January 1998.

[24] D. E. Perry and C. S. Stieg. Software faults in evolving a large, real-time system: a case study. In *ESEC*, pages 48–67, 1993.

[25] J. F. Power and B. A. Malloy. Program annotation in XML: A parse-tree based approach. In *WCRE*, pages 190–200, 2002.

[26] T. Quatrani. *Visual Modeling With Rational Rose and UML.* Addison-Wesley, October 2002.

[27] S. Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, 1995.

[28] M. Sipser. *Introduction to the Theory of Computation.* Course Technology, February 2005.

[29] A. Viljamaa and J. Viljamaa. Creating framework specialization instructions for tool environments. In *The Tenth Nordic Workshop on Programming and Software Development Tools and Techniques*, 2002.

[30] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, 2005.