Automating and Validating Program 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

Program annotations help to catch errors, improve program understanding, and specify invariants. Adding annotations, however, is often a manual, laborious, tedious, and error prone process especially when programs are large. We offer a novel approach for automating a part of this process. Developers first specify an initial set of annotations for a few variables and types. 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 variables. We evaluate our prototype implementation on open-source software projects and our results suggest that Lean can generalize from a small set of annotated variables to annotate many other variables. 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, taking less than one hour to run on an application with over 20,000 lines of code.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques–Computer-aided software engineering (CASE); D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems; I.5.4 [**Pattern Recognition**]: Applications

General Terms

Experimentation, Documentation, Algorithms

Keywords

Annotations, semantic concepts, machine learning, program analysis, feature diagrams, runtime monitoring

1. INTRODUCTION

Program annotations assert facts about programs. They appear as comments in the source code as comments or within special language statements. An annotation may assert that values of program variables stay within certain ranges. Annotations may be used to describe program types, values, or identifiers. The importance of annotations for program checking and verification is emphasized in the proposal for a verifying compiler [17]. Program annotations help to catch errors, improve program understanding, recover software architecture, and specify invariants.

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]. Thus, the extra work required to annotate programs is likely to reduce development and maintenance time, as well as to improve software quality. However, annotating programs is often a manual, tedious, and error prone process especially for large programs. Although some programming languages (e.g., C#, 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.

In this work, we focus on deriving semantic concepts for unannotated variables from an initial set of annotated variables. Simply stated, semantic concept annotations 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[1]`, all for the `Address` concept [1]. The name of the variable `S[1]` 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.

Our solution, called *LEarning ANnotations (Lean)*, combines program analysis, run-time monitoring, and machine learning to automatically apply a small set of initial semantic annotations to additional unannotated types and variables. The input to Lean is program source code and a *concept diagram* describing relations between semantic concepts. Concept diagrams are graphs whose nodes represent semantic concepts and edges represent relationships between these concepts. The core idea of Lean is that after programmers provide a few initial annotations of some variables and types with these semantic concepts, the system will glean enough information from these annotations to annotate much of the rest of the program automatically. We define annotation rules that guide the assignment of semantic concepts to program variables and types, and resolve conflicts when they arise.

[1]These names are taken from open-source program Vehicle Maintenance Tracker whose code fragments are shown in Figure 1.

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 XXXXXXXXXX, XXXXXXXX
Copyright 2006 ACM 0-89791-88-6/97/05 ...$5.00.

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 a profile of the values of the instrumented variables. Lean uses this profile to train its learners to identify variables with similar profiles. Lean's learners then classify the rest of program variables by matching them with semantic concept annotations. Once a match is determined for a variable, Lean annotates it with the matching semantic concept.

Annotating 100% of variables automatically is not realistic. Many reasons exist: machine learning approaches do not guarantee absolute success in solving problems; concept diagrams representing program design specifications may not match programs; and some concepts may be difficult to relate to program variables due to lack of modularity. Consequently, Lean makes mistakes when learning annotations. In order to improve its precision, Lean uses program analysis to determine relations among annotated variables. Then Lean compares these relations with corresponding relations in concept diagrams. If a relation is present between two variables in the program code and there is no relation between concepts with which these variables are annotated, then Lean flags it as a possible annotation error.

We evaluate our approach on open-source software projects and obtain results that suggest it is effective. Our results 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% 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. A MOTIVATING EXAMPLE

The *Vehicle Maintenance Tracker (VMT)* is an open source Java application that records the maintenance of vehicles [4]. Fragments of the VMT code from three different files are shown in Figures 1(a)–1(c), and a concept diagram is shown in Figure 1(d).

A fragment of code from the file `vendors.java` shown in Figure 1(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 1(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 1(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 identifier, address, email, phone number, 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 1(a)– 1(c) with concepts from the diagram shown in Figure 1(d). This diagram is a graph whose nodes designate semantic concepts and edges specify relations between these

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

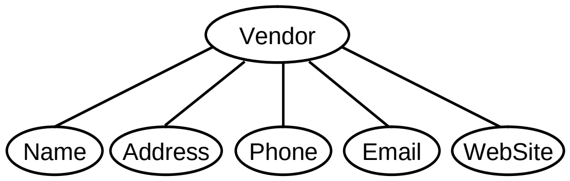
(a) File vendors.java.

```
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[3];
```

(c) File VMT.java.



(d) A concept diagram for the VMT application.

Figure 1: Code fragments from programs of the VMT project and its concept diagram.

concepts. We observe that the spellings of some variable names are similar to the names of corresponding concepts, i.e., `Pho` – `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.

We also observe that if concepts are related in a diagram, then types and variables that are annotated with these concepts are related in the code too. The relation between concepts in a diagram means that instances of data described by these concepts are linked in some way. For example, the concept Name is related in the concept Vendor, in the concept diagram shown in Figure 1(d). This relation can be expressed as "Vendor has a Name." The variable Name which is annotated with the concept Name is defined in the class vendors which is annotated with the concept Vendor. The containment relation in the source code corresponds to the "has-a" relation in the concept diagram. Lean explores program source code to analyze relations between program variables and types, and then compares them with relations among corresponding concepts in diagrams in order to infer and validate annotations. We explain this process in Section 6.

3. A PROBLEM STATEMENT

This section defines the problem of automating and validating program annotations formally. We define the structure of concept diagrams and types of relations between concepts in diagrams and variables and types in program code. These relations are used in our algorithm for inferring and validating annotations. Then, we present rules for annotating program variables and types, and give a formal definition of the problem.

3.1 Definitions

Feature modeling is a technique for modeling software with concept diagrams, where a feature is an end-user-visible characteristic of a system describing a unit of functionality relevant to some stakeholder [5, 9]. For example, for the code shown in Figure 1 features are Vendor, Name, Email, Phone, Website, and Address. Concept diagrams used in feature modeling are called *feature diagrams (FD)*. Feature diagrams are graphs whose nodes designate features and edges (also called *variation points*) specify how two features are attached to each other.

Three basic types of diagrams are shown in Figure 2. Features f_1 and f_2 are optional if one of them or both or none can be attached to the base feature P. Mandatory are features f_3 and f_4 since both of them must be attached to the base feature P. Finally, features f_5 and f_6 are alternative if either f_5 or f_6 are attached to the base feature P.

We view the process of annotating program variables and types with semantic concepts as *mapping* these variables and types to concepts. We limit our approach to mapping program variables and types to features; we do not consider mapping fragments of code or selected statements or lines of code to features. The latter is important in certain cases, and it is the subject of our future work.

We formalize the mapping between features and program variables and types in the definition of the *Annotation Function*.

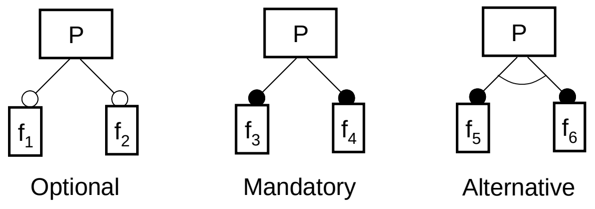


Figure 2: Basic types of feature diagrams.

DEFINITION 1 (ANNOTATION FUNCTION). *The annotation function $\alpha_o : o \rightarrow 2^F$ maps a program object (variable), o, to a subset of features, F, in a feature diagram, and the annotation function $\alpha_\tau : \tau \rightarrow 2^F$ maps a type (basic type, class, interface, or method), τ, to a subset of features, F.*

The dependency between a type τ and an object o is expressed by the function Type(o) = τ, which maps the object o to its type $\tau \in T$, where T is the set of types. Without the loss of generality we use the annotation function $\alpha : \pi \rightarrow 2^F$ that maps a program entity π (i.e., variable or type) to a subset of features F in some feature diagram, where program entity $\pi \in \{o, \tau\}$.

Relations between program entities are used to create and validate mappings between these entities and features. These relations are given in Definitions 2 – 10.

DEFINITION 2 (EXPRESSION). *The n-ary relation Expression $\subseteq v_1, \ldots v_n$ specifies a program expression, where $v_1, \ldots v_n$ are variables or methods used in this expression.*

DEFINITION 3 (NAVIGATE). *The navigation relation Navigate(p, q) $\subseteq$ Expression(p, q) is the expression p.q where q is a member (e.g., field or method) of object p.*

DEFINITION 4 (ASSIGN). *The assignment relation Assign $\subseteq$ p $\times$ Expression specifies the assignment expression p = Expression, where p is the variable on the left-hand side and the Expression relation stands for some expression on the right-hand side.*

DEFINITION 5 (CAST). *The cast relation Cast(p, q) $\subseteq$ Expression(p, q) is the cast expression (q)p where p $\in$ o and q $\in$ T i.e., casting an object p to some type q.*

DEFINITION 6 (SUBTYPE). *The relation SubType(p, q) specifies that a type p is a subtype of some type q. In Java this function is implemented via the implement interface clause. That is, a class p that implements some interface q is related to this interface via the SubType relation.*

DEFINITION 7 (INHERIT). *The Inherit(p, q) relation specifies that a class p inherits (or extends in Java terminology) some class q.*

DEFINITION 8 (CONTAINS). *The Contains(p, q) relation specifies that program entity q is contained, or defined within the scope of some type p.*

We call interfaces, classes, and methods containment types because they contain other types, fields and variables as part of their definitions. That is, interfaces contain method declarations, classes contain definitions of fields and methods, and methods contain uses of fields and declarations and uses of local variables.

DEFINITION 9 (δ–RELATION). *The relation $\delta(\pi_k, \pi_n)$ stands for "programming entity π_k is related to the programming entity π_n via the `Expression`, `Navigate`, `Assign`, or `Cast` relations."*

This relation is irreflexive, antisymmetric, and transitive. For example, from the expression `x = y + z` we can build four δ relations: δ`(x, y)`, δ`(x, z)`, δ`(y, z)`, and δ`(z, y)`. If variables are used in the same expression and their values are not changed after this expression is evaluated, then their order is not relevant in the δ–relation. However, if the value of a variable is changed as a result of the evaluation of the expression, then this variable is the first component of the corresponding δ–relation.

DEFINITION 10 (γ–RELATION). *The relation γ $(f_p,\ f_q)$ stands for "feature f_q is connected to feature f_p via a variation point."*

This relation is irreflexive, antisymmetric, and transitive. For example, from the mandatory feature diagram shown in Figure 2, we can build two γ relations: γ`(P,` f_3`)` and γ`(P,` f_4`)`. If a feature f_r is attached to a feature f_s, then feature f_s is the first component of the corresponding γ relation.

The annotation function α maps pairs from the relation δ to pairs from the relation γ. Suppose that `(a, b)` $\in \delta$ and `(c, d)` $\in \gamma$. Then the element `(a, b)` is annotated with the element `(c, d)` if and only if c $\in \alpha$(a) and d $\in \alpha$(b). As a shorthand we write `(c, d)` $\in \alpha$`((a, b))`.

3.2 Rules of Program Annotations

When programmers map types (i.e., interfaces, classes, and methods) and their variables (i.e., fields and objects) to features, these mappings may conflict with one another. For example, if a class is mapped to one feature and it implements an interface that is mapped to a different feature, then what default mapping would be valid for an instance of this class? This section offers heuristic rules to resolve ambiguities when annotating programs.

Direct mapping: $\gamma \in \alpha(\delta)$ expresses the fact that for a δ–relation between program entities that are annotated with feature labels there is a corresponding γ–relation between features that annotate these program entities.

Instance mapping: instances of a type that is mapped to some set of features are automatically annotated with these feature labels. We write this rule as `(Type(`o`)` $= \tau \wedge$ `f` $\in \alpha(\tau)$`)` $\Rightarrow$ `f` $\in \alpha(o)$.

Member annotation: If the container type is mapped to some feature, then all of its members are automatically mapped to the same feature, i.e., `(Contain(p, q)` $\wedge$ `f`$\in \alpha$`(p))` $\Rightarrow$ `f`$\in \alpha$`(q)`.

Expression annotation: if variables in an expression defined by the relation `Expression(` $\subseteq v_1, \dots v_n$`)` are annotated with some set of features, that is, without the loss of generality $f_1 \in \alpha(v_1), \dots, f_n \in \alpha(v_n)$, then Lean annotates this expression with a set of features as $f_1 \cup f_2 \cup \dots \cup f_n$.

Assignment annotation: Given the relation `Assign(p, expr)`, the expression `expr` is annotated with a set of features `f`, then the variable `p` is annotated with the same set of features. The converse holds true, i.e., `Assign(p, expr)` $\wedge$ `(f` $\in \alpha(p) \Leftrightarrow$ `f` $\in \alpha$`(expr))`. For example, the variable `s` in the fragment of code shown in Figure 1(c) is assigned the value of the variable `S[2]`, which is mapped to the concepts `Address`. According to the assignment annotation rule, the variable `s` maps to the concept `Address`.

Cast annotation: casting an object `p` to some type `q` automatically remaps this object `p` to the feature to which this casting type is mapped. If the type `q` is not mapped to any feature, then the original mapping of the object `p` is not changed. That is, `(Cast(p, q)` $\wedge$ `f` $\in \alpha$`(q))` $\Rightarrow$ `f` $\in \alpha$`(p)` and `(Cast(p, q)` $\wedge\ \alpha$`(q)` $= \oslash \wedge$ `f` $\in \alpha$`(p))` $\Rightarrow$ `f` $\in \alpha$`(p)`.

Sometimes a default mapping should be overwritten. For example, a class may be mapped to one feature, but its instances should be mapped to some other features. The following rule handles this condition.

Instance overriding: default feature labels assigned to an instance by the instance mapping rule can be overwritten: `(Type(`o`)` $= \tau \wedge$ `f` $\in \alpha(\tau) \wedge$ `Overwrite(`o`, g))` $\Rightarrow$ `(f` $\in \alpha(\tau) \wedge$ `g` $\in \alpha(o)$`)`.

Containment: if an object `q` is a member of type `p` that is annotated with some feature label `f`, then the object `q` is also annotated with the feature label `f`: `(Contain(p, q)` $\wedge$ `f` $\in \alpha$`(p))` $\Rightarrow$ `f` $\in \alpha$`(q)`.

Member overriding: the mappings for members of the containment types can be overwritten: `(Contain(p, q)` $\wedge$ `f` $\in \alpha$`(p)` $\wedge$ `Overwrite(q, g))` $\Rightarrow$ `(f` $\in \alpha$`(p)` $\wedge$ `g` $\in \alpha$`(q))`.

Precedence: if the containment type is mapped to one feature and the type of its member variable is mapped to a different feature, then this variable is mapped to the same feature to which its type is mapped: `(Contain(p,` o`)` $\wedge$ `f` $\in \alpha$`(p)` $\wedge$ `g` $\in \alpha$`(q)` $\wedge$ `Type(`o`) = q)` $\Rightarrow$ `g` $\in \alpha(o)$.

Interface: when programmers map interfaces to features, these mappings are preserved in classes that implement mapped interfaces: `(Subtype(p, q)` $\wedge$ `f` $\in \alpha$`(q)` $\wedge$ `Contains(q, z))` $\Rightarrow$ `(f` $\in \alpha$`(z)` $\wedge$ `Contains(p, z))`. That is, if an interface is mapped to some features and is implemented by some class that is mapped to different features, then the interface fields and methods implemented by this class remain mapped to the interface features.

Inheritance: if a class extends some other class that is mapped to some feature, then the extended class is automatically mapped to the same feature: `(Inherit(p, q)` $\wedge$ `f` $\in \alpha$`(q)` $\wedge\ \alpha$`(p)` $= \oslash$`)` $\Rightarrow$ `f` $\in \alpha$`(p)`. This rule is dictated by the Java idiom of inheritance stating that a class may implement many interfaces, but it can extend only one class. The extended class can be explicitly remapped to a different feature without affecting the mapping defined for the parent class.

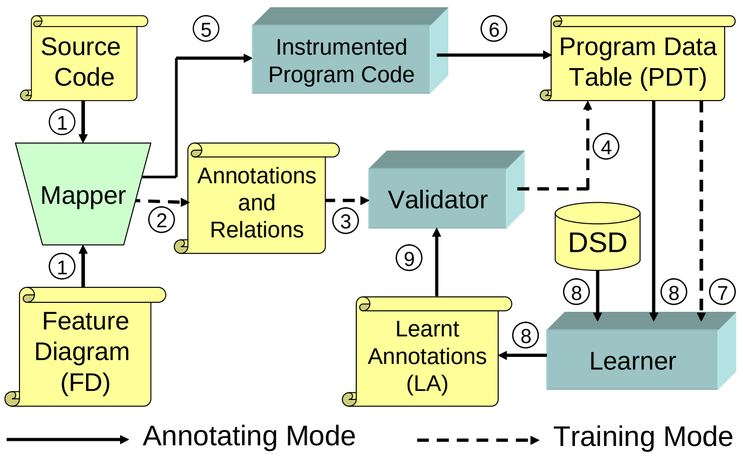
3.3 The Formal Problem Statement

When programmers specify an initial mapping between program entities and features, they define a partial function α_0 over the domain of program entities $\pi_0 \in \pi$ and the range of features $f_0 \subseteq f$. The goal of Lean is to compute the partial function α_1 over the subset of the domain of program entities $\pi_1 \subseteq \pi$ and the range of features $f_1 \subseteq f$ abiding by the rules specified in Section 3.2, such that $\pi_0 \subseteq \pi_1$ and $f_0 \subseteq f_1$ and $\forall$ `(a, b)` $\in \delta$, `a,b` $\in \pi$; `c,d` $\in$ `F`, such that for `c` $\in \alpha$`(a)` $\wedge$ `d` $\in \alpha$`(b)` $\mid \exists\, \gamma \in$ `(c, d)`. That is, if program entities `a` and `b` are annotated with feature labels `c` and `d` respectively, and these entities are related to each other via some δ–relation, then the features labeled `c` and `d` in some FD should be related via some γ–relation.

4. LEAN ARCHITECTURE AND PROCESS

The architecture for Lean and its process description are shown in Figure 3. The main elements of the Lean architecture are the Mapper, the Learner, and the Validator shown in Figure 3(a). Solid arrows show the process of annotating program entities with feature labels, and dashed arrows specify the process of training the Learner.

The inputs to the system are program source code and an FD. The Mapper is a *Graphic User Interface (GUI)* tool whose components are Java and XML parsers, program and FD analysis routines, and an instrumenter. A Java parser produces a tree representing pro-



(a) Lean architecture.

1) Programmers annotate the Source Code with concepts from the FD using the Mapper
2) The Mapper expands and outputs initial mappings and δ and γ relations
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 3: Lean Architecture and its process.

gram entities. FDs are represented in XML format, and the Mapper uses an XML parser to produce a tree representing features and variation points in the FD. For detailed explanation of architecture components and formats, see our technical report [16].

Programmers use the Mapper GUI to specify initial mappings between features from the FD and program entities from the source code. The Mapper GUI presents both the FD and the programming entities using tree widgets. The Mapper generates initial annotations from specified mappings in the XML format. Each variable is described by its *access path*, and features are specified by their names. For example, if a variable named `var` is declared in the method `M` of the class `C` which is defined in the package `P`, then the access path to this variable is `P.C.M.var`.

The user 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 feature for annotating this variable, or annotating it does not warrant the amount of work required. Variables whose values contain binary (nonprintable) characters may also be excluded since some machine learning algorithms may not classify these variables correctly.

The Mapper builds relations defined in Section 3.1 using the Mapper's program analysis routines. Then the Mapper's rule engine analyzes these relations and initial annotation links, and expands these initial annotations to other program entities using the rules from Section 3.2. The Mapper outputs user annotations and relations in the XML file, which is then passed to the Validator.

The Validator checks the correctness of annotations by exploring relations between variables in the source code and features in the FD. Recall the direct mapping rule stating that for a δ–relation between annotated entities in the source code there is a γ–relation between the features with whose labels these program entities are annotated. The output of the Validator is a list of rejected annotations which the Learner may use to improve its predictive capabilities.

The Mapper instruments the source code to record run-time values of program variables. 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 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 learnt information to classify unannotated variables.

After instrumenting the source code, the Mapper calls a Java compiler to produce an executable program. Then the program runs storing names and the values of program variables in the Program Data Table (PDT). 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, which is based on a machine-learning Java-based open source system WEKA [25]. Once the Learner is trained, it classifies unannotated program variables 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 supplying them to the Learner which emits predictions for feature labels with which these variables 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). These LAs are sent to the Validator to check their correctness. The Validator sends corrected annotations to the Learner for training to improve its accuracy. 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.

Lean produces Java annotations using the annotation type facility of Java 1.5. An example of a Java annotation type declaration and its use is shown in Figure 4. 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 `FD` and `Label`, for the name of feature diagram and a feature label respectively. Annotations consist of the `@` sign followed by the annotation type `Concept` and a parenthesized list of element-value pairs.

5. 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 dif-

```
public @interface Concept {
    String FD;
    String Label;
    }
@Concept(FD="VMT", Label="Vendor")
public class vendors {...}
```

Figure 4: Example of the declaration of a Java annotation type and its use by Lean.

ferent domains.

5.1 The Learning Approach

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

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. Lean employs the LSD multistrategy learning approach [19, 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 variable matches each feature 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 labels of the predictions for the same program variable.

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 variables. Trained learners classify program variables with feature labels, 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 variables. Then, during the classification step the previously computed weights are used to predict feature labels for unannotated variables.

Each base leaner is assigned a weight (a real number between 0 and 1) that characterizes its accuracy in predicting annotations of program variables. Initially, all weights are the same. For each classified program entity the weights of the learner are modified to minimize the squared error between the predefined value (i.e., 1 if the prediction is correct, or 0 otherwise) and the computed probability multiplied by the weight assigned to the learner. This approach is called regression [20].

5.2 The Learners

There are three types of base learners used in Lean: a name matcher, a content matcher, and a Bayes learner [12, 20]. 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 LSD.

Name matchers match the names of program entities with feature labels. 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 feature label. 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. They do not perform well 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, and they perform poorly on binary (nonprintable) and numeric types of variables.

Finally, Bayes learners, particularly the Naïve Bayes classifier, are among the most practical and competitive approaches to classification problems [20]. Naïve Bayes classifiers are studied extensively [14, 20], so we only state what they do in the context of the problem that we are solving here. For each program variable the Naïve Bayes classifier assigns some feature label f_k, such that the probability that that this variable belongs to the feature f_k, is maximized.

5.3 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 "wafer" is encountered in a value of some variable of a program written for a semiconductor domain, this variable may be annotated with the `wafer` concept. Many domains have dictionaries listing special words, their synonyms, and explaining their meanings.

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 variable, then this variable may be classified and subsequently annotated by some concept. We use a simple heuristic to change the probabilities that variables should be annotated with certain feature labels. If no dictionary word is encountered among the name or the values of a variable, 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 variable, 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. VALIDATING ANNOTATIONS

This section describes how we infer and validate program annotations using program analysis. First, we state the rationale for inferring and validating annotations. Then, we give the algorithm for inferring annotations for program variables that are components of δ–relations. Finally, we show how to validate program annotations.

6.1 The Rationale

Lean cannot annotate all variables due to a number of factors. Machine learning approaches are only as good as the training data, and they do not guarantee 100% classification accuracy. Some variables cannot be classified because they assume hard-to-analyze values. Examples are variables whose values are binary (nonprintable) strings. It is difficult to train classifiers on such variables since patterns in binary data are inherently complex. For these and other reasons Lean may assign feature labels to variables incorrectly.

We want to explore the direct mapping rule defined in Section 3.2 to infer annotations for partially annotated δ–relations (that contain annotated and unannotated components) in order to increase the annotation coverage. Further, we use this rule to validate the correctness of annotations for fully annotated δ–relations. This is accomplished by the `InferAnnotation` and `Validate` algorithms described below.

6.2 The Inference Algorithm

The algorithm InferAnnotations for inferring annotations is given in Algorithm 1. Its input is the set of δ–relations and mappings α. The algorithm iterates through all δ–relations in the main `for`-loop to find partially annotated ones. Then, for each found δ–relation the annotation function is applied to the annotated component to obtain the set of features with which this component is annotated. Then, for each feature in this set all γ–relations are found whose components match this feature. The other components of the obtained γ–relations make it into the set of possible annotations for the unannotated component of the δ–relation. The last `if` condition in the algorithm deals with the same variable used in two or more expressions in the same scope. Since Lean may annotate this variable differently, an intersection is taken of the feature sets with which the uses of this variable are annotated to reject incorrect annotations.

Algorithm 1 The InferAnnotations procedure

```
**InferAnnotations**( $\delta$, $\alpha$ )
**for all** (a, b) $\in \delta$ **do**
    $\alpha$(a) $\mapsto f_k$
    $\alpha$(b) $\mapsto f_m$
    **if** $f_k = \oslash$ **then**
        **AddFeatureLabel**( $f_k$, $f_m$ )
    **else if** $f_m = \oslash$ **then**
        **AddFeatureLabel**( $f_m$, $f_k$ )
    **end if**
    **if** $\exists$ (u, v) $\in \delta$ s.t. u = a $\vee$ v = a **then**
        **if** u = a **then**
            $\alpha$(u) $\mapsto f_s$
            $\alpha$(u) $\mapsto f_s \cap f_k$
        **else if** v = a **then**
            $\alpha$(v) $\mapsto f_s$
            $\alpha$(v) $\mapsto f_s \cap f_k$
        **end if**
        $\alpha$(a) $\mapsto f_s \cap f_k$
    **end if**
**end for**

**AddFeatureLabel**( $f_k$, $f_m$ )
**for all** (c, d) $\in \gamma$ **do**
    **if** d $\in f_m$ **then**
        $f_k \mapsto f_k \cup$ c
    **else if** c $\in f_m$ **then**
        $f_k \mapsto f_k \cup$ d
    **end if**
**end for**
```

6.3 The Validation Algorithm

The Validate algorithm shown in Algorithm 2 checks for the correctness of assigned annotations. The key criteria is expressed by the direct mapping rule. The input to this algorithm is the set of δ–relations, γ–relations, and mappings α. Each δ–relation has a color associated with it, which is initially set to `red`. The `red` color means that a given δ–relation is not correctly annotated, and the `green` color means that all components of a given δ–relation are either annotated correctly, or not annotated at all. For this procedure to work, initial labeling should be valid.

Algorithm 2 The validation procedure

```
**Validate**( $\delta$, $\gamma$, $\alpha$ )
**for all** (a, b) $\in \delta$ **do**
    color((a, b)) $\mapsto$ red
    $\alpha$(a) $\mapsto f_k$
    $\alpha$(b) $\mapsto f_m$
    **if** $f_k \neq \oslash \wedge f_m \neq \oslash$ **then**
        **for all** (c, d) $\in \gamma$ **do**
            **if** (c $\in f_m \wedge$ d $\in f_k$) $\vee$ (c $\in f_k \wedge$ d $\in f_m$) **then**
                color((a, b)) $\mapsto$ green
                break
            **end if**
        **end for**
    **else if** c $\in f_m$ **then**
        color((a, b)) $\mapsto$ green
    **end if**
**end for**
**for all** (a, b) $\in \delta$ **do**
    **if** color((a, b)) $\neq$ green **then**
        **print** error
    **end if**
**end for**
```

The outer `for`-loop iterates through all δ–relation, and annotations f_k and f_m for components of each relation are obtained. If one or both components of a given δ–relation are not annotated, then this relation is colored `green`. Otherwise, the inner `for`-loop searches through γ–relations to find one whose components are members of f_k or f_m annotation sets. If such a γ–relation exists, then the corresponding δ–relation is colored `green` and the inner loop is exited. Otherwise, if the inner loop exits without finding a γ–relation whose components are members of f_k or f_m annotation sets, then the δ–relation is `red` and may not be valid.

This algorithm does not specify what the correct annotation of a program variable is or what caused the error in program annotation. In fact, annotation errors many be caused by incorrect feature diagram, errors in program source code, or both. The last `for`-loop iterates through all δ–relations, checks the colors, and prints `red` relations as potentially incorrect.

6.4 The Computational Complexity

Suppose a program has n variables and a feature diagram has m features. Then it is possible to build $\frac{n(n-1)}{2}$ δ–relations and $\frac{m(m-1)}{2}$ γ-relations. Thus, the space complexity is $O(n^2 + m^2)$. The time complexity is determined by two `for`–loops in the `Validate` and `InferAnnotations` algorithms. The external `for`–loops iterate over all δ–relations and the internal `for`–loops iterate over all γ–relations. Considering all other operations in the algorithms taking $O(1)$, the time complexity is $O(n^2m^2)$.

7. 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 relations between program entities as specified in Section 3.1, and expand initial annotations to unannotated program entities using the rules specified in Section 3.2. 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. At this point, ARFF files are input into WEKA to train Learners and classify unannotated variables.

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 6. The Validator takes its input in XML format and outputs a PDT in ARFF format. The input XML file contains annotations specified by users and expanded by the Mapper, along with excluded program entities and δ and γ relations. The output ARFF file contains variable names and concepts assigned to them.

8. EXPERIMENTAL EVALUATION

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

8.1 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 a `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.

8.2 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 the first experiment, we create a DSD and an FD for each subject program. Then we annotate a small subset of variables for each program manually using the Mapper, and then run Lean to annotate 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.

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 second 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. 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.

8.3 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 features in the FD, the number of variables 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 without a strong influence of any domain terminology. 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 variable names are relatively easy to interpret and classify.

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 variables using the rest of the input data. Figure 5 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 [6] is applied to collect training samples over the life time of applications, then Lean can maintain and evolve program annotations with evolving programs.

Finally, we used the Learner trained for the VMT application to annotate variables in other applications. This methodology is called *true-advice* versus *self-advice* which uses the same program for training and evaluation. Figure 6 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 share common concepts with the VMT application, and it

Program	Size of DSD, words	Lines of code	Number of features	Num of vars	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.

allows learners to be trained and used interchangeably thus achieving the high degree of automation in annotating program variables.

Algorithms for inferring and validating annotations predict annotations for partially annotated δ–relations (i.e., when one component of a δ–relation is annotated, and the other is not) and detect when incorrect annotations are assigned in certain situations. We found that these algorithms are not sound since they can miss incorrect annotations and they cannot pinpoint the source of the fault that led to incorrect annotations. However, as our experiments show, these algorithms perform well in practice for the majority of cases by discarding up to 83% of incorrectly assigned annotations.

9. 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. One of the earliest papers that belongs to the first category describes a technique for annotating Algol-like programs automatically given their specifications [11]. The annotation techniques are based on the Hoare-like inference rules which derive invariants from the assignment statements, or from the control structures of programs. Like the Lean approach, the annotation process is guided by a set of rules. The program is incrementally annotated with invariant relationships that hold between program variables at intermediate points. Unlike our approach program annotation process is viewed strictly as discovery of invariants by applying inference rules.

A comprehensive study of program annotations presents a classification of the assertions that were most effective at detecting faults and describes experience using an assertion processing tool that addresses ease-of-use and effectiveness of program annotations [23]. Unlike our approach, this tool does not automate the annotation process.

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 [22]. 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 semantic relations and diagrams that are not linked to program source code or language grammars.

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 [24]. 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 [21]. 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

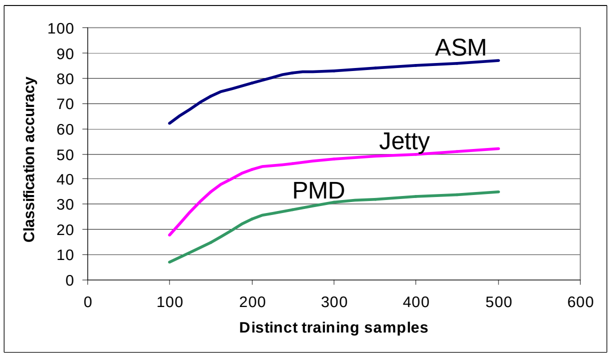


Figure 5: The graph of the accuracy of the Lean learner.

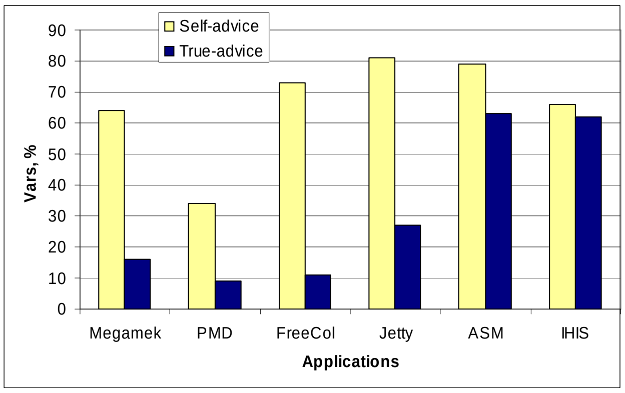


Figure 6: Percentage of variables that the Lean Learner annotates with self-advise (left bar) versus the true-advise annotations (right bar) when the Learner is trained on the VMT application.

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 [18]. 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.

10. 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 formalization of Lean rules;
- novel algorithms for inferring and 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.

In our experiments it took less than one hour to annotate each subject application. Our experience suggests that Lean is practical for many applications, and its algorithms are efficient and effective.

Acknowledgments

We warmly thank Don Batory for his comments and suggestions.

11. 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] Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute at Carnegie Mellon University, 1990.

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

[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] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison Wesley, June 2000.

[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] M. Grechanik, K. S. McKinley, and D. E. Perry. Automating and validating program annotations. Technical Report TR-05-39, Department of Computer Sciences, The University of Texas at Austin, 2005.

[17] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. In *CC*, pages 262–272, 2003.

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

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

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

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

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

[23] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Software Eng.*, 21(1):19–31, 1995.

[24] 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.

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