Recovering Use-Case-Diagram-To-Source-Code Traceability Links

Mark Grechanik
Systems Integration Group
Accenture Technology Labs
Chicago, IL 60601
mark.grechanik@accenture.com

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

Dewayne E. Perry
Department of ECE
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 programs source code, and maintaining traces between the source code and elements of UCDs is a manual, tedious, and laborious process.

We offer a novel approach for automating a part of this process. Developers first specify few traceability links (TLs). Our system combines these links with run-time monitoring, program analysis, and machine-learning approaches to recover and validate additional TLs between types and variables in Java programs and elements of UCDs. We evaluate our prototype implementation on open-source software projects, and our results suggest that our approach can generalize from a small set of initial links to recover many other TLs with a high degree of automation and precision.

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 [13]. Currently, major software design tools support UCDs [18][11].

Requirements traceability (RT) is the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of ongoing refinement and iteration in any of these phases) [10]. RT is important not only for enabling successful completions of software projects, but also for improving the overall quality of software [10][12][21][16]. Even though tracing requirements expressed in UCDs to programs source code yields various benefits [1], in practice it is rarely done because it is a manual, tedious, and laborious process.

Our solution, called *LEarning ANd Analyzing Requirements Traceability (LeanArt)*, combines program analysis, run-time monitoring, and machine learning to automatically propagate a small set of initial *traceability links (TLs)*, also called *traces* or *links*, between *program variables and types (program entities)* and elements from UCDs to additional unlinked program entities thereby recovering new TLs. The input to LeanArt is program source code and UCDs. The core idea of LeanArt is that after programmers initially link few program entities to elements of the UCDs, the system will glean enough information from these links to recover TLs for much of the rest of the program automatically.

LeanArt is a lightweight approach for recovering TLs that differs fundamentally from other approaches since it uses runtime values of program entities in conjunction with static information, and it does not depend on exact matches between the names of elements of UCDs and the names of program entities. In addition, LeanArt uses program analysis and a compositional algorithm in a novel way to improve the precision of the recovered TLs.

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 link approximately 6% of the program entities to elements from UCDs, LeanArt correctly recovers 87% TLs in the best case, 64% in the average, and 34% in the worst case, taking less than thirty minutes to run on an application with over 20,000 lines of code.

2 Use Case Diagrams

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 manages maintenance records of vehicles (http://vmt.sourceforge.net).

UCs show actors, depicted as human figure icons, and

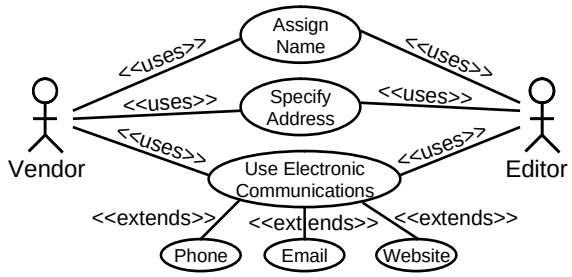

Figure 1. A UCD for the VMT project.

these actors carry out actions that are depicted as ovals. Actors can be human users or components of software products. For example, in Figure 1 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, which we call elements of UCDs, 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 some other action, which is more general. For example, vendors use `Phone`, `Email`, and `Website` which are specialized actions of the more general action of using `Electronic Communications`.

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.

When recovering TLs programmers map classes and variables shown in Figures 2(a)– 2(b) to the elements from the UCD shown in Figure 1 by observing that the names of some program entities are similar to the names of the corresponding elements of the UCD. For example, the names of fields in the class `VendorEdit` partially match the names of the corresponding fields in the class `vendors` and the names of the elements of the UCD (e.g., `Pho` – `PhoneText` – `Phone`).

However, this informal matching procedure does not work for the fragment of code shown in Figure 2(c). To what element of the UCD 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

```
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]});
  }
};
```

(c) File VMT.java.

Figure 2. Code fragments from selected programs of the Java-based VMT application.

number, email, and web site respectively. No VMT documentation mentions this information, and it is up to programmers to run this program in order to discover the meanings of the variable `S`.

While some programmers use meaningful names, others name program entities arbitrarily [2]. When names of program entities are meaningless, like in the case of the variable `S`, programmers run applications in order to obtain runtime values of program variables. Then programmers look for distinct structures in the values of these variables in order to determine their meaning. For example, the variable `S[3]` contains only numbers and dashes in the format `xxx-xxx-xxxx`, where `x` stands for a digit and the dash is a separator, which corresponds to the pattern for the US phone numbers. Consequently, programmers trace this variable to the element `Phone` from the UCD.

Recovering traceability links (TLs) is a manual and laborious process especially when names of program entities are meaningless. 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 [6], and Corbi reports that up to 50% of the maintenance effort is spent on trying to understand code [5]. We see the main benefit of recovering UCD-to-source-code TLs in reducing the amount of time that programmers spend discovering the meaning of legacy code.

4 The Problem Statement

Our goal is to recover TLs between variables and types (entities) in Java programs and elements from the corresponding UCDs with a high degree of automation and precision. We attempt neither to recover TLs between fragments of code (or lines of code or selected statements) and elements of UCDs, nor to infer traces between classes (or interfaces) and elements of UCDs based on the recovered links between the fields of these classes (or interfaces) and elements from UCDs. This is a subject of our future work.

In addition, we do not consider cases when a program entity can be traced to multiple elements of UCDs. For example, a variable that holds concatenated values for an address, a phone number, and an email may be traced to the corresponding elements of the UCD. In this paper we are interested in tracing such entities correctly to one element of some UCD. Extending our approach to recovering multiple TLs for a program entity is a subject of our future work.

It is not possible to develop a sound and complete approach for automatic recovery of TLs between program entities and elements of UCDs. An approach is sound when program entities are linked to elements from UCDs correctly or not linked at all. False TLs (i.e., tracing program entities to elements from UCDs incorrectly) are not produced by a sound approach. An approach for recovering TLs is complete if it recovers links to some elements from UCDs for all program entities. While a sound and complete approach for automatic recovery of TLs is desirable, it is in general an undecidable problem[1].

We want to design an automatic approach that mimics a human-driven manual and laborious procedure for recovering TLs between program entities and elements of UCDs with a high precision. That is, our approach should automate the process of searching for patterns in the names and values of program entities and use detected patterns to match these entities to elements of UCDs thereby recovering TLs. In addition, our approach should be able to discard incorrectly recovered TLs in order to increase the precision. Our approach should be lightweight and it should fit into a software development process without introducing additional operations for programmers.

[1]Suppose that values described by some element from a UCD 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 element from a UCD and the program variable are identical, then the program variable is described by, and consequently can be linked to this element. However, determining if two CFGs generate the same set of strings is an undecidable problem [19].

5 Our Approach

In this section we describe key ideas of and give intuition into why and how our approach works.

5.1 Key Ideas

Our main idea is to mimic the human-driven procedure of searching for common patterns and similarities between the names and values of program entities and the names of elements of UCDs. If programmers knew patterns for the values and the names of program entities in advance, then they could write pattern-matching routines to recover TLs between program entities and elements from UCDs whose names match these patterns. Even though writing these routines is tedious and laborious, this approach does not solve our problem. The information about patterns may be unavailable, and it is difficult to collect and analyze values and names of program entities and extract exact patterns manually.

in addition, with a widespread problem of using meaningless names for program entities [2], the only way to recover TLs is to peek at runtime values of program variables. Currently, this is done manually, with programmers running programs in the debugging mode and stopping at preset breakpoints to study the values of program variables.

We realize the idea of automating this human-driven procedure for recovering TLs by using *machine learning (ML)* techniques. With ML, program entities can be classified as belonging to elements of UCDs based on both the names of program entities, their runtime values, and the names of elements of UCDs thus recovering TLs. ML techniques can be used effectively for partial matches between names and values and when patterns in these names and values are not defined precisely.

Since our approach is not sound, mistakes are made when learning TLs. Our idea is to improve the precision of our approach by determining relations between program entities and by comparing these relations with corresponding relations between elements in UCDs to which these entities are traced. If a relation is present between two entities in the program code and there is no relation in UCDs between elements to which these entities are traced, then a false TL warning should be issued. This heuristics is based on the observation that relations between elements of UCDs are often preserved in the program code. We claim that it is possible to detect a large percentage of false TLs automatically using this heurustics, and we substantiate this claim with the results of our experiments in Section 9.

5.2 Relations

TLs are pairs $(t,c) \in \alpha$, where α is the traceability relation, t is a program entity, and c is an element of some UCD. Relations between elements in UCDs are expressed as pairs $(c_p,c_q) \in \gamma$, where c_p and c_q are elements of some UCDs, and γ is the relation between these elements. 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, variable-variable δ–relations specify that two variables are used in the same expression.

We distinguish between the *extend* and the *use* γ–relations between elements of UCDs. Two elements of UCDs are connected with the *use* γ–relation if they are linked with an edge labeled anything but `<<extend>>`. Actions and actors connected with relations labelled `<<extend>>` in UCDs are often implemented in program source code using inheritance. It means if a type-type δ–relation exist between classes in the source code, then it is likely that the elements of UCDs which are traced to these classes are connected using the extend γ–relation.

5.3 Validation Algorithm

The idea of the validation algorithm is to guess TLs for untraced program entities using existing traces 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 traceability relation. Traces are guessed by composing these relations. Relations δ and α can be composed if the second component of some pair in the δ–relation matches the first component of some pair in the 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 the program entity t may be traced to the element c of a UCD. These suggested TLs are used only to validate traces determined by the ML techniques. The set $\alpha \backslash \sigma$ is the set of flagged TLs that should be reviewed by programmers.

We propose a validation algorithm that uses the heuristics stating that for a δ–relation between program entities in the source code there is a γ–relation between the elements in UCDs to which these entities are traced. Suppose a programmer determines that some program entity t_n should be traced to some element c_p of some UCD. This trace 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 a program, and elements c_p and c_q are also related in some UCDs. 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 may be traced to the element c_q.

After applying the ML part of the LeanArt, it may recover two TLs expressed as 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 recovered relation $\alpha(t_m,c_q)$ is validated. However, the second recovered relation $\alpha(t_m,c_w)$ is flagged as possibly false since there is no corresponding σ–relation. Programmers should review flagged traces and reject them if they are proved to be false.

This validation algorithm can be used to recover TLs, however, its accuracy is too low, and it performs worse than the ML component. Our experiments showed that without ML component the validation algorithm recovers many additional incorrect TLs and it misses correct TLs. The results of this experiment are described in Section 9.6.

6 LeanArt Architecture and Process

The architecture for LeanArt is shown in Figure 3. The main elements of the Lean architecture are the Mapper, the Learner, and the Validator. TLs are stored in the Links database along with the information about UCDs and program entities.

LeanArt works as follows. Initially, programmers create traces by linking a small percentage of program entities to elements of UCDs. Then LeanArt instruments a program to perform run-time monitoring of program variables. LeanArt uses a Java compiler to compile this instrumented program. When this program is executed, LeanArt collects the values of the program variables, and it uses these values along with the initial traces and the names of program entities and elements of UCDs to train its Learner to identify entities with similar values and names. LeanArt's Learner then classifies the rest of program entities by matching them with the names of the elements of UCDs. Once a match is determined for an entity and it is approved by the Validator, LeanArt links this entity with the matching element from some UCD and stores it in the Links database.

The inputs to the system are a UCD `(1)`, program source code `(2)`, and a set of initial traces produced by the programmer `(3)`. The system's component that accepts these inputs is the Mapper, which is a tool whose components are a Java parser, program and UCD analysis routines, and an instrumenter. The Mapper constructs δ and γ relations using its program analysis routines `(4)`, and it enters initial TLs into the Links database `(5)`.

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

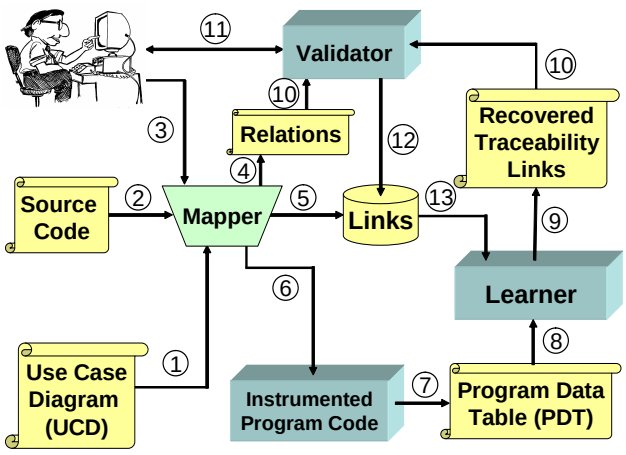

Figure 3. LeanART architecture.

The Mapper instruments the source code to record runtime values of program variables. After instrumenting the source code, the Mapper calls a Java compiler to produce an executable program (6). Then, the program runs, storing names and the values of program variables in the *Program Data Table (PDT)* (7). PDT serves as an input to the Learner (8), which is based on WEKA, a machine-learning Java-based open source system [22].

Learner is trained on TLs from the Links database (13) and the runtime data from the PDT (8). This step highlights our assumption that initial TLs used to train the Learner should be correct. Once the Learner is trained, it classifies untraced program entities that are supplied to the Learner in the PDT by analyzing their runtime values and names. We discuss the Leaner in Section 7.

The output of the Learner is a set of Recovered Traceability Links (RTLs) (9). These RTLs are sent to the Validator which uses the compositional algorithm described in Section 5.3 to check if these RTLs may be false (10). The Validator sends its recommendations to the programmer (11) who reviews them and approves or rejects suspected TLs. Approved links are stored in the Links database (12). The Learner can improve its predictive capabilities by using the results of this validation (13). That is, if the Learner recovers TLs incorrectly, then it can be retrained on these negative examples to improve its performance. This continuing process of recovering, validating, and learning from invalidated TLs makes LeanArt effective for long-term evolution and maintenance of software systems.

7 Learning Traceability Links

We treat the automation of recovering TLs as a classification problem: given elements of UCDs and a program entity, which element matches this entity the best? The Learner classifies program entities with the probabilities that these entities can be traced to certain elements of UCDs based on the information learnt from previously traced entities.

Since it is difficult to find a learning algorithm that can deliver consistently good results for different types of input data, LeanArt employs the *Multistrategy Learning Approach (MLA)* which organizes multiple learners in layers [15]. 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 MLA, each base learner issues predictions that a program entity matches a UCD element 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. Our choice of the MLA is based on the opportunity to plug into LeanArt a variety of learning algorithms that have different and complementary properties, thereby improving the precision of recovering TLs.

Base learners match the names and the values of program entities with the names of elements of UCDs. In LeanArt, we experiment with well-known and proven algorithms such as Whirl [4] and Naïve Bayes classifier, however, many other classifiers are available and can be used in LeanArt. While Whirl-based matchers work well for meaningful names, Bayes learners perform well when classifying numerical as well as string entities, and they compensate for the deficiencies of the Whirl algorithm.

8 The Prototype Implementation

Our prototype implementation includes the Mapper and the Validator. The Mapper is written in C++, and it uses the EDG Java front end (http://www.edg.com). 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 an instrumenter routine that adds the logging code to the original program, and the logging code outputs the values of the program variables into the PDT in the *Attribute Relation File Format (ARFF)* which is the standard format for Weka [22].

The Validator is written in C++ and contains less than 1,000 lines of code. Its routines implement the compositional validation algorithm which is described in Section 5.3. The Validator takes its input in the XML file that contains TLs specified by programmers and δ– and γ– relations.

9 Experimental Evaluation

In this section we describe the results of experimental evaluation of LeanArt 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 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. 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.

Table 1 contains characteristics of the subject programs, their UCDs, and relations. Its first column shows the names of the subject programs, followed by the number of non-commented lines of code, LOC. Other columns show the number of UCDs, number of elements of UCDs, and the numbers of γ– and δ–relations.

Program Name	LOC	# of UCDs	# of elem	# of γ–rels	# of δ–rels
Megamek	23,782	4	25	76	16,263
PMD	3,419	3	12	51	913
FreeCol	6,855	2	17	83	13,672
Jetty	4,613	2	6	12	540
VMT	2,926	3	8	24	1,739
ASM	12,294	3	23	117	18,033
IHIS	1,883	4	14	35	1,208

Table 1. Characteristics of the subject programs, their UCDs, and relations.

9.2 Selecting Input 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 Creating UCDs

Since the subject programs do not come with UCDs, two groups of graduate students created UCDs for subject programs. They did this work as part of taking graduate software engineering courses. These students were not familiar with the subject programs, and acquired information about them by reading their source code, and running these programs under debuggers to study values of program variables. Then, these students recovered TLs for program entities for each program manually based on their analysis of debugging information and their understanding of the source code. This process took approximately four and a half months for twenty three graduate students.

9.4 Threats to Validity

A major threat to the validity of this study is that we were not able to find nontrivial programs whose UCDs had been created before these programs were written. In our experiments students created UCDs by reverse engineering subject programs, and these UCDs may not be identical to ones created as part of the forward engineering process, when UCDs are created first and programmers are guided by these UCDs when they write programs.

UCDs created by reverse engineering programs are different from the UCDs created as part of the forward engineering process in several respects. Reverse-engineered UCDs may match the source code better since they are closely based on the source code and the names of their elements as well as relations between them may match program entities with a higher precision than the elements of the UCDs created before subject programs are written. However, if programmers use UCDs created at the early stages of requirements gathering to guide them when writing code, then the resulting source code should match UCDs just as close as the reverse-engineered UCDs would match the source code.

The other threat to validity is that students might make mistakes when recovering TLs manually, and we did not have a control group to verify their TLs due to the difficulty to find students for this laborious and tedious process. Even if this control group existed, it would be difficult to make sure that they did a better job than the original group. This uncertainty reflects a real-world environment when

Program Name	Run min	PE	ITL	RTL	CTL	WTL	BTL	DTL	ATL	GTL	BTLR	VPR	ACC
Megamek	26	328	20	308	92	21	113	16	178	194	0.07	0.81	0.58
PMD	9	176	11	165	47	22	69	39	56	95	0.13	0.26	0.34
FreeCol	20	527	31	496	113	15	128	79	288	367	0.03	0.43	0.58
Jetty	6	96	6	90	12	4	16	9	64	73	0.04	0.55	0.71
VMT	14	143	9	144	16	2	18	29	86	125	0.01	0.92	0.87
ASM	28	218	13	205	22	8	30	37	137	174	0.04	0.85	0.67
IHIS	11	225	14	211	25	4	29	23	158	181	0.02	0.86	0.75

Table 2. Results of the experimental evaluation of LeanArt with the initial TLs (ITL)≈6%.

programmers, who write source code using UCDs, could also make mistakes when providing TLs.

Our subject programs are of small to moderate size because it is difficult to find a large number of graduate students or programmers who would spend significant amount of time recovering TLs manually for large-scale software projects. Large applications whose creation is guided by UCDs may have different characteristics compared to our small to medium size subject programs. Increasing the size of applications to millions of lines of code may lead to a nonlinear increase in the analysis time and space demand for LeanArt. If this happens, then more time should be spent on making LeanArt scalable.

9.5 Control Variables

We observe and measure a number of control variables. `PE` is the number of program entities, and the `ITL` is the number of initial TLs, `ITL < PE`. The number of TLs that should be recovered, `RTL = PE - ITL`, `RTL > 0`. The Learner issues RTL predictions some of which may be incorrect. Thus `RTL` is the sum of *Good Traceability Links (GTL)* and *Bad Traceability Links (BTL)*, `RTL = GTL + BTL`. `BTL` is the sum of `CTL`, which is the number of correct TLs that are mistakenly discarded by the Validator, and `WTL`, which is the number of wrong TLs that the Validator passes, `BTL = CTL + WTL`. `GTL` is the sum of the `DTL`, which is the number of correctly discarded TLs and `ATL`, which is the number of correctly accepted TLs, `GTL = DTL + ATL`.

The quality of LeanArt is measured using three ratios: `ACC`, `VPR`, and `BTLR`. Leaner's accuracy ratio is computed as $\texttt{ACC} = \frac{\texttt{ATL}}{\texttt{RTL}}$, and the Validator's precision ratio is computed as $\texttt{VPR} = \frac{\texttt{GTL - BTL}}{2\times\texttt{RTL}} + \frac{1}{2}$. The `ACC` variable is the ratio of correctly recovered TLs, and the `VPR` ratio shows how mistaken the Validator is when analyzing recovered TLs. Constants are used in the formula for the VPR in order to normalize its values, $\texttt{VPR} \in [0,1]$, where `VPR=0` means that all recovered TLs are incorrect, and `VPR=1` means that they are correct.

The difference between the `ACC` and `VPR` measures is that `ACC` is used to evaluate the performance of the Learner while `VPR` shows the combined performance of the Learner and the Validator. The variable `ACC` is analogous to the recall parameter in information retrieval, which is the ratio of the number of relevant documents retrieved to the total number of documents.

The intuition behind the variable `VPR` is that the benefits of program comprehension obtained through recovering TLs may be negated by wrong TLs mixed up with correct ones since programmers will use them to make decisions [23]. The ratio `VPR` can be used in conjunction with the variable `BTLR`, which stands for *Bad Traceability Links Ratio* and it is computed as $\texttt{BTLR} = \frac{\texttt{WTL}}{\texttt{RTL}}$. The closer the VPR ratio to `1` and the closer the `BTLR` to `0`, the more correct LeanArt is and the fewer wrong TLs are recovered.

9.6 Experiments

The goal of the first experiment is to determine how effective LeanArt is in recovering TLs for subject programs. Since all program entities are traced to elements of UCD manually, we compare TLs recovered and validated by LeanArt with the links manually determined by graduate students. Ideally, if all recovered TLs coincide with manually recorded traces, then the LeanArt accuracy `ACC` is `1.0`, the VPR is `1.0`, and the `BTLR = 0`.

Table 2 contains results of the experimental evaluation of LeanArt on the subject programs with the number of initial TLs (`ITL`) selected at approximately 6%. These `ITLs` were selected based on the closest similarities between the names of program entities and the names of elements of UCDs. The columns of this table contain the names of subject programs, the LeanArt running time in minutes for each subject program, the number of program entities, `PE`, the number of initial TLs, `ITL`, and other control variables described in the previous section. The last three columns show the BTLR ration, LeanArt's precision, `VPR`, and the

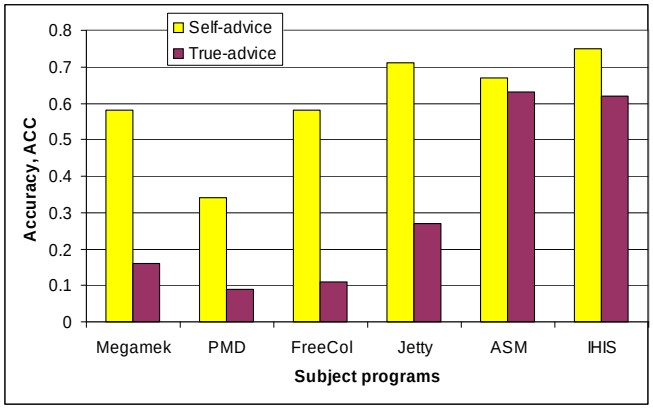

Figure 4. Accuracy, ACC, of the Learner recovering TLs with self-advice (left bar) versus the true-advice (right bar) when the Learner is trained on the VMT application.

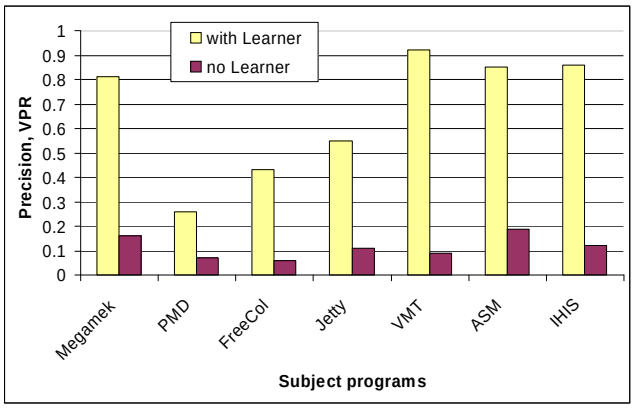

Figure 5. LeanArt's precision, VPR, when using the Learner (left bar) versus no Learner used (right bar).

accuracy, ACC.

The highest accuracy is achieved with programs ASM and VMT which are written for specific domains with well-defined terminologies, and whose entity names are easy to interpret and classify. The lowest level of accuracy was with the program PMD which analyzes Java programs whose code does not use terminologies from any specific domain. Our experiment shows that the Validate algorithm performs well in practice for the majority of cases achieving the VPR = 0.92 for the VMT application with the BTLR = 0.01, which means that this algorithm accepts correctly recovered TLs while discarding most wrong TLs.

Next, we used the Learner trained for the VMT application to recover TLs for other applications. This methodology is called *true-advice* versus *self-advice* which uses the same program for training and evaluation. Figure 4 shows the accuracy ratio ACC with which the LeanArt recovers TLs correctly with self-advice (left bar) versus the true-advice (right bar) when the Learner is trained on the VMT application. This experiment shows that LeanArt can be trained on one application and used to recover TLs for other programs if they operate on similar data. ASM and IHIS share some names and data with the VMT application, and it allows learners to be trained and used interchangeably thus achieving the high degree of automation.

The goal of the next experiment is to evaluate whether the validation algorithm can be used to recover TLs without using the Learner. Recall that the validation algorithm suggests possible TLs based on composing relations. We hypothesize that many TLs suggested by the validation algorithm are incorrect. When used with the Learner, these incorrect traces do not affect the results since they are discarded when there are no matching links recovered by the Learner. To carry out this experiment we turned off the Learner and used the Validator instead of it. The results are shown in Figure 5 where the measurements for the LeanArt's precision, VPR, are depicted when running LeanArt with the Learner (left bar) versus no Learner used (right bar). The precision is five to ten times worse when Learner is not used, that is the validation algorithm recovers many additional incorrect TLs and it misses correct ones.

Finally, we determine how choosing different program entities for initial TLs randomly and increasing their number affects LeanArt's accuracy and precision. In general, programmers tend to choose familiar program entities for specifying initial TLs. These entities tend to have names matching the names of elements from UCDs, and it eases the selection process for initial TLs. However, we are interested to check to see how LeanArt performs if initial TLs were chosen at random. While increasing the number of entities chosen for initial TLs may lead to better accuracy of the Learner, choosing more entities for initial TLs makes the LeanArt process more expensive. The goal of this experiment is to provide a guideline to what percentage of the total program entities should be chosen for initial TLs to give an acceptable accuracy of recovering additional TLs.

The results of this experiment are shown in Figure 6 in the stock-price-style graph. The horizontal axis shows the percentage of the total number of program entities chosen randomly for initial TLs, and the vertical axis shows the accuracy of the Learner, ACC. For each percentage of initial traces we run a series of experiments in each of which program entities were chosen randomly for these traces. The vertical lines on this graph show the maximum and minimum ACC for running the experiment on the same number of different initial TLs, and the average ACC is shown by the horizontal mark on the vertical lines. While the gap be-

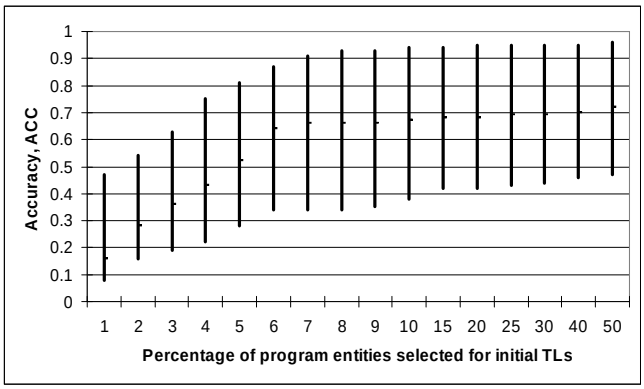

Figure 6. Dependency of the classification coverage and accuracy from the percentage of randomly selected program entities for initial annotations.

tween the minimum and the maximum `ACCs` is large, the average shows that in order to get a good accuracy it is sufficient to create initial TLs for less than seven percent of program entities.

10 Related Work

While there is a lot of research on requirements traceability in general, we review related work that falls into two major categories: systems that use *Information Retrieval (IR)* techniques to recover TLs, and systems that use static and dynamic source code analysis in conjunction with other formal methods to obtain additional information about TLs once they are manually entered into a system. We think that many techniques and systems described below are complementary to LeanArt since they can be used together to achieve high precision and automation for recovering and analyzing TLs between different kinds of documents.

ARTS is one of the earliest systems for automating requirements traceability that belongs to the latter category [7]. ARTS allows users to enter programs and requirements manually and then perform some automatic analyses on the data. On the contrary, LeanArt automates the process of recovering traceability links between programs source code and requirements expressed as UCDs, and its output can be used as an input to ARTS.

TOOR is a tool for tracing requirements between different artifacts of software development [17]. It uses a template-based approach where users are required to fill out the information about artifacts and relations into their corresponding templates, and TOOR uses this information to enhance TLs. Like LeanArt, TOOR exploits relations between software artifacts. However, LeanArt is mostly concerned with automating the part of process of recovering TLs between requirements and program entities, and this activity is performed manually in TOOR.

TraceAnalyzer is a tool that detects TLs between test and usage scenarios, models (e.g., use cases or class diagrams), and classes in the source code by collecting and analyzing runtime information about class methods [8][9]. Like in LeanArt, programmers specify a small number of TLs, and TraceAnalyzer may recover additional links between requirements and program classes. However, TraceAnalyzer requires test and usage scenarios be linked to classes, and it does not support TLs at a finer granularity (e.g., to program variables and other types besides classes).

IR approaches convert the problem of recovering TLs into the problem of forming queries from names of elements of one software artifact and obtaining links to mostly relevant elements from other artifacts as a result of executing these queries. One common difference between IR-based approaches and LeanArt is that IR-based techniques depend on meaningful names assigned to program entities, while LeanArt performs well on programs with meaningless names of program entities.

A *goal centric traceability (GCT)* approach uses IR techniques in order to establish TLs between nonfunctional requirements and software artifacts expressed using UML diagrams [3]. A main difference between LeanArt and GCT approaches is that LeanArt is designed to recover TLs between UCDs and programs source code while GCT is designed to work solely with class diagrams.

A *requirement-to-object-model (ROM)* recovers TLs between textual parts of requirement documents and UML class diagrams [20]. Unlike ROM, LeanArt is designed to recover TLs between programs source code and UCDs.

Latent Semantic Indexing (LSI) is an IR-based approach for recovering documentation-to-source code traceability links [14]. LSI utilizes comments and identifier names within the source code to match them with sections of corresponding documents. In contrast, LeanArt does not depend on similarities between names of identifiers in program source code and words in requirements documents.

11 Conclusion

We offer a novel approach for automating a part of the process of recovering traceability links between the source code of Java programs and elements of use case diagrams. Our system, LeanArt, combines initial traceability links with run-time monitoring, program analysis, and machine-learning approaches to recover and validate additional TLs between types and variables in Java programs and elements of UCDs. 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 link approx-

imately 6% of the program entities to elements from UCDs, LeanArt correctly recovers 87% TLs in the best case, 64% in the average, and 34% in the worst case, taking less than thirty minutes to run on an application with over 20,000 lines of code.

We believe that our approach has significant potential. Even though LeanArt is currently implemented to work with Java programs, there are no inherent limitations against using LeanArt for other languages. Since requirements can be expressed in other forms besides UCDs, LeanArt can be extended to work with other representations of requirements diagrams.

References

[1] B. C. D. Anda, K. Hansen, I. Gullesen, and H. K. Thorsen. Experiences from using a UML-based development method in a large safety-critical project. *Empirical Software Engineering*, 2005.

[2] N. Anquetil and T. C. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON*, page 4, 1998.

[3] J. Cleland-Huang, R. Settimi, O. B. Khadra, E. Berezhanskaya, and S. Christina. Goal-centric traceability for managing non-functional requirements. In *ICSE*, pages 362–371, 2005.

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

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

[6] 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.

[7] M. Dorfman and R. F. Flynn. ARTS - an automated requirements traceability system. *Journal of Systems and Software*, 4(1):63–74, 1984.

[8] A. Egyed. A scenario-driven approach to traceability. In *ICSE*, pages 123–132, 2001.

[9] A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *ASE*, pages 163–171, 2002.

[10] O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In *RE*, pages 94–101, 1994.

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

[12] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering.* Springer, November 2004.

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

[14] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–137, 2003.

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

[16] J. D. Palmer. *Traceability.*, pages 412–422. IEEE Computer Society Press, February 2000.

[17] F. A. C. Pinheiro and J. A. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, 1996.

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

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

[20] G. Spanoudakis. Plausible and adaptive requirement traceability structures. In *SEKE*, pages 135–142, 2002.

[21] R. Watkins and M. Neal. Why and how of requirements tracing. *IEEE Software*, 11(4):104–106, 1994.

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

[23] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *ICSE*, pages 215–223, 1981.