Recovering Software Specifications with Inductive Logic Programming

William W. Cohen
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
wcohen@research.att.com

Abstract

We consider using machine learning techniques to help understand a large software system. In particular, we describe how learning techniques can be used to reconstruct abstract Datalog specifications of a certain type of database software from examples of its operation. In a case study involving a large (more than one million lines of *C*) real-world software system, we demonstrate that off-the-shelf inductive logic programming methods can be successfully used for specification recovery; specifically, Grendel2 can extract specifications for about one-third of the modules in a test suite with high rates of precision and recall. We then describe two extensions to Grendel2 which improve performance on this task: one which allows it to output a set of candidate hypotheses, and another which allows it to output specifications containing determinations. In combination, these extensions enable specifications to be extracted for nearly two-thirds of the benchmark modules with perfect recall, and precision of better than 60%.

Introduction

Frawley *et al.* [1991] define *knowledge discovery* as the "extraction of implicit, previously unknown, and potentially useful information." Machine learning methods are often used to perform knowledge discovery from databases. Here we investigate the use of machine learning methods for a different knowledge discovery task—understanding a large software system. This is an important application area, as program understanding is a major task in maintaining large software systems [Corbi, 1989]; it has been estimated that for large software systems, more than half of the time spent on maintenance is spent on program understanding [Parikh and Zvegintzov, 1983, page ix].

We present a case study involving a large real-world software system which investigates the use of machine learning methods for *specification recovery*—*i.e.*, constructing specifications of software from the software itself. Such specifications are useful for program understanding, but are often not provided with existing software systems. More specifically, we investigate recovering Datalog definitions of database views from *C* routines which implement these views.

In addition to its potential practical importance, the problem of specification recovery raises (in this domain) a number of interesting research problems. One problem is technical: the natural representation for specifications is a first-order language that is in general hard to learn. Another problem is methodological: as the high-level goal is not prediction but discovery, it is not obvious how one should evaluate the learning system. Despite these difficulties, it will be demonstrated that off-the-shelf but state-of-the-art learning methods can be successfully used for specification recovery in this domain. However, these off-the-shelf methods can also be improved by adapting them more closely to the task at hand.

The Discovery Problem

The experiments of this paper were carried out with the "Recent Change" (henceforth RC) subsystem of the 5ESS[1] switching system. A large part of the 5ESS system is a *database* which encodes all site-specific data. As the database is used by the switch, which is a real-time, distributed system, it has a number of quirks; however, for the purposes of this paper, it can be regarded as a conventional relational database.

The database is optimized for retrieval, and hence information is often stored internally in a redundant and unintuitive format. Thus, the switch administrator accesses the database through the *RC subsystem*, which contains several hundred *views* into the database. RC views are essentially "virtual relations"; they can be accessed and updated in much the same manner as actual relations, but are designed to present a simpler interface to the underlying data. For reasons of efficiency each view is implemented as a *C* routine which supports a standard set of operations, such as reading and deleting tuples in the view. These routines can be quite complex. In particular, in updating a view, it is often

[1] 5ESS is a trademark of AT&T.

necessary to check that *integrity constraints* are maintained. Integrity constraints may affect many database relations and checking them is often non-trivial.

The purpose of our discovery system is to automatically construct high-level, declarative specifications of these views. The constructed specifications are in Datalog (*i.e.*, Prolog with no function symbols), and describe how views are derived from the underlying relations. As an example, a view v_1 which is a projection of columns 1,3,5 and 4 of relation r might be specified by the one-clause Datalog program

$$v_1(A,B,C,D) \leftarrow r(A,Z,B,D,C).$$

Similarly, a projection v_2 of the join of two relations s and t might be specified

$$v_2(A,B,C,D) \leftarrow s(A,B,D),\ t(B,A,Y,Z,C)$$

It may seem surprising that a language as restricted as Datalog can be useful in specifying real-world software systems. We emphasize that *these specifications are not complete descriptions* of the underlying code; they suppress many "details" such as integrity constraints, error handling, and how relations are stored in the database (which is actually distributed over several processors). However, they are a useful description of one important aspect of the code.

We conjecture that in many large systems, software understanding can be facilitated by such abstract, high-level descriptions. Ideally such high-level descriptions would be provided along with the source code: this would be the case, for example, if the source were generated automatically from a specification. For older software systems, however, this is unlikely to be the case. We note that software understanding is especially important for older systems, as they tend to be harder to maintain. These "legacy" systems are are also more likely to be replaced by a newer system with equivalent functionality—an enterprise which again requires understanding the existing system.

To summarize, the *specification recovery* problem we will consider here is to automatically construct *view specifications* like the one above. This particular specification recovery problem is attractive for a number of reasons. The domain is broad enough to be interesting, as the implementation of a view can be arbitrarily complex. On the other hand, the domain is constrained enough so that specification recovery is often possible; in particular, there is reason to believe that concise abstract specifications do indeed exist for many of the views. Another advantage is that although the subsystem being studied is huge (more than 1,100,000 lines of C) it is highly modular; hence insight into the behavior of this large system can be gained by solving many moderate-sized specification recovery problems. The population of problems is also important enough that a successful discovery tool would be practically interesting, and is large enough to form a useful testbed for specification recovery techniques.

Learning View Specifications

The approach we propose to recovering view specifications is the following. First, execute the code to find all tuples in a view v, given a specific database DB. (This process is sometimes called *materializing* the view v.) Second, construct a dataset in which all tuples in the materialized view are positive examples, and all tuples not in the materialized view are (either implicitly or explicitly) negative examples. Third, learn a Datalog definition of v from this dataset, and return this learned definition as the specification of v. The learning step is the most complex one; however there are currently a number of "inductive logic programming" systems which learn logic programs from examples (*e.g.*, see [Quinlan, 1990; Muggleton and Feng, 1992]).

Learning specifications with FOIL

To make this idea concrete, let us return to our example view v_1.[2] Suppose the materialized view v_1 (derived from the relation r) is as follows:

Materialized view v_1:

ssnum	first	last	mi
421-43-4532	dave	jones	q
921-31-3273	jane	jones	q

Relation r:

id	dept	fname	lname	mi
921-31-3273	sales	jane	jones	q
421-43-4532	sales	dave	jones	q

From the view one can derive these examples of the target relation v_1:

$+v_1$(*421-43-4532,dave,jones,q*)
$+v_1$(*921-31-3273,jane,jones,q*)

Using these examples, and also using the relation r as background knowledge, the inductive logic programming system FOIL [Quinlan, 1990] will produce the correct specification $v_1(A,B,C,D) \leftarrow r(A,E,B,D,C)$. We say "correct" as this example is isomorphic to an actual RC view, and the specification correctly describes the implementation. The specification is also considerably more concise than the implementation—which in this case contains 80 non-comment lines of C—and hence is arguably also more understandable.

Note that in this example we reduced FOIL's search space dramatically by providing the single relevant background relation r, rather than all relations in DB. Restricting the search in this way is essential, as RC databases are large. Fortunately, automatically finding the relevant relations is easy, as computer-readable

[2]Except when otherwise indicated, the examples of this paper are isomorphic to real RC views. However, identifiers have been changed: in part to avoid divulging proprietary information, and in part to avoid reader confusion, as RC identifiers appear rather cryptic to an outsider. For pedagogical reasons, we have also used the simplest view specifications as examples.

documentation exists describing which RC relations are accessed by which RC views.

Although FOIL's performance on this small example is encouraging, an more detailed exploratory study conducted using real RC views revealed several limitations. First, for noise-free problems, FOIL is known to scale well with the number of training examples, but poorly with the arity of background relations [Pazzani and Kibler, 1992]. (This problem is shared by GOLEM [Muggleton and Feng, 1992], another well-known inductive logic programming system.) However, many of the relations in the RC database have large arities—*e.g.*, the 36 views used in our exploratory experiments used 18 relations with arity greater than 10 and 9 relations with arity greater than 25.

A second problem with FOIL stems from the fact that the materialized view provides no explicit negative examples. FOIL contains a mechanism for automatically generating negative examples, using the closed world assumption; however, while this technique is in principle appropriate, in practice it is impractical to create enough negative examples to prevent over-generalization on views of moderately high arity. For example, FOIL is unable to correctly learn the view $v_3(A,B,C,D,E,F,G) \leftarrow u(A,B,C,D),w(A,G,F,E,H)$ from a dataset of 136 examples; even with a sample of 360,000 negative tuples FOIL learns the overgeneral specification $v_3(A,B,C,D,E,F,G) \leftarrow u(A,B,C,D)$.

A final disadvantage of FOIL is that due to coding conventions and the like, there are many regularities across view recovery problems which could potentially be exploited in learning; however, FOIL has no mechanism for using such constraints. As a concrete example, it happens that RC view specifications always are composed of *generative* clauses—*i.e.*, when $v(X_1,\ldots,X_n)$ is the head of a clause in a view specification, each X_i always appears somewhere in the body of the clause. However, FOIL can and often does hypothesize non-generative specifications.

Learning specifications with Grendel2

To summarize our exploratory study, we concluded that to recover view specifications, better mechanisms were needed for preventing over-generalization and for incorporating domain-specific constraints. One inductive logic programming system which has such mechanisms is Grendel2.

Grendel2 is a successor system to Grendel [Cohen, 1992]. In addition to the usual set of positive and negative examples, Grendel takes as input an explicit description of the hypothesis space to be searched—*i.e.*, the intended *bias* of the learning system—written in a formalism called an *antecedent description grammar* (ADG). ADGs are essentially context-free grammars. Grendel's hypotheses are sets of clauses whose antecedents are sentences of an ADG provided by the user.

ADGs provide a way of describing biases used in "theory-guided" learning systems like FOCL [Pazzani and Kibler, 1992]; however they cannot easily express certain other biases. To concisely describe "language biases" like ij-determinacy [Muggleton and Feng, 1992] a generalization of ADGs called "augmented ADGs" has been proposed [Cohen, 1993]. Augmented ADGs are analogous to *definite clause grammars* [Sterling and Shapiro, 1986].

Grendel2 is an extension of Grendel that uses augmented ADGs. Like Grendel, Grendel2 uses a FOIL-like technique to search its hypothesis space.[3] The output of Grendel2 is in general a set of clauses, each of which is a sentence of the augmented ADG.

By applying Grendel2 to the view recovery problem, we were able to exploit a number of domain-specific constraints. In Grendel2's datasets, each example is not simply a tuple in the materialized view, but a tuple to which extra information has been attached: specifically, each view tuple is tagged with the list of database tuples read in constructing this view tuple. (These tags were obtained automatically by adding additional "instrumentation" code to the database interface routines.) For example, the dataset for v_1 above is

+trace(v_1(421-43-4532,dave,jones,q),
[r(421-43-4532,sales,dave,jones,q)]).
+trace(v_1(921-31-3273,jane,davis,q),
[r(921-31-3273,sales,jane,davis,q)])

An augmented ADG was then written which generates certain common types of view specifications—namely, projections, certain types of joins, and combinations thereof—and which also encodes a number of additional constraints. One constraint is that clauses generated by the ADG must access the database in an order consistent with the traces of each view tuple. Also, clauses must be generative, in the sense defined above, and use only relevant database relations. These constraints can be expressed quite compactly using ADGs: the largest ADG used in the experiments in this paper (the one used for Grendel/MD, below) contains just 11 rewrite rules and about 30 lines of code.

No negative examples are given to (or constructed by) Grendel2. Instead of using negative examples to prevent over-general hypotheses from being produced, generality is implicitly constrained by the requirement that hypotheses be sentences of the ADG.

The augmented ADG also generates clauses in a special format, as shown on the left-hand side of Figure 1. This format is convenient, as in the RC subsystem a mnemonic *field name* is associate with each column in a view or relation; these field names can be easily added to the specification, as is shown by the variant specification on the right-hand side of Figure 1. (For

[3] Note that this is different from the learning method described in [Cohen, 1993], in which a restricted class of augmented ADGs were converted into simple ADGs and passed to Grendel.

trace(v_1(A,B,C,D),[r(E,F,G,H,I)]) $\leftarrow$ E=A, r(E,F,G,H,I), B=G, C=H, D=I.	v_1(Vssnum,Vfirst,Vlast,Vmi) $\leftarrow$ Rid=Vssnum, r(Rid,Rdept,Rfname,Rlname,Rmi), Vfirst=Rfname, Vlast=Rlname, Vmi=Rmi.

Figure 1: Specifications generated by Grendel2. On the left, a specification as produced by the learner. On the right, a more readable version of the specification, formed by removing "trace" information and adding mnemonic field names.

readability, information associated with the traces has been removed from this specification.) Since RC programmers prefer to think in terms of field names rather than column positions, we believe this format to be more readable to persons familiar with the domain. Addition of these mnemonic names is automatic, using existing computer-readable documentation on the RC database.[4]

Evaluation of Discovery Systems

While machine learning tools are typically evaluated by the accuracy of the hypotheses that they produce, this metric is appropriate only when the final purpose of the system is prediction. In this section we will address the methodological question: how should one evaluate a discovery system of the type described above?

This question is made more difficult by the fact that data is often stored redundantly in the RC database. To take a simple artificial example, suppose that we have the following view and relation:

Materialized view v_4:

first	last
jane	jones
dave	smith
⋮	⋮

Relation n:

fname	lname	login
jane	jones	jones
dave	smith	smith
⋮	⋮	⋮

If there is an integrity constraint that requires the *id* field of relation n to be identical to the *lname* field, then the following are equally valid specifications of view v_4:

```
v4(First,Last) ←
    n(Fname,Lname,Login),
    First=Fname,
    Last=Lname.

v4(First,Last) ←
    n(Fname,Lname,Login),
    First=Fname,
    Last=Login.
```

It is not at all obvious what the discovery system *should* do in this case. We will assume here that both specifications are of interest to the user, and that an ideal discovery system would recover both of them.

To be precise, let us fix a specification language $\mathcal{L}$. For every view v, there is a set of specifications $CorrectSpecs_{\mathcal{L}}(v)$ in $\mathcal{L}$ that can be considered *correct* in the sense that for every legal database, they will produce the same set of tuples as the actual C implementation of v. Assuming that the goal of the user of the system is to find all correct specifications of each view, the specification recovery problem can thus be restated quite naturally as an information retrieval task: given the query "what are the correct specifications of view v?" the discovery system will return a set $ProposedSpecs(v)$, which would ideally be identical to $CorrectSpecs_{\mathcal{L}}(v)$.

If one knew the actual value of $CorrectSpecs_{\mathcal{L}}(v)$, one could measure the performance of a discovery system via the standard information retrieval measures of *recall* and *precision*. *Recall* is the percentage of things in $CorrectSpecs_{\mathcal{L}}(v)$ that appear in $ProposedSpecs(v)$; it measures the percentage of correct specifications that are proposed. *Precision* is the percentage of things in $ProposedSpecs(v)$ that appear in $CorrectSpecs_{\mathcal{L}}(v)$; it measures the number of correct *vs.* incorrect specifications that are proposed. (Notice that for the learning algorithms discussed above, $ProposedSpecs(v)$ is always a singleton set, and hence high recall can be obtained only if $CorrectSpecs_{\mathcal{L}}(v)$ is usually a singleton set.)

Unfortunately, in the RC subsystem it is not easy to determine $CorrectSpecs_{\mathcal{L}}(v)$ for a view v: although one can find one correct specification s by manually reading the code, to determine if a second specification s' is equivalent to s for all legal databases requires knowledge of the integrity constraints enforced by the system, which are not completely documented. Thus, to evaluate the system, we used a cross-validation like approach to evaluate specifications. A routine was written that takes a database DB and a materialized view v and enumerates *all* consistent one-clause specifications allowed by the grammar. Let us call this set $ConsistentSpecs_{\mathcal{L}}(v, DB)$. Recall that a correct specification is by definition one that materializes the correct

[4] In passing, we note that the ordering of the conjuncts in the body of this clause is not arbitrary, but is based on knowledge about which view and relation fields are indexed. Literals in the body are ordered so that following the usual Prolog evaluation order, the operation of a specification mimics the operation of the actual C code in performing an indexed read.

view for all databases. Hence

$$CorrectSpecs_{\mathcal{L}}(v) \;\equiv\; \bigcap_{j} ConsistentSpecs_{\mathcal{L}}(v, DB_j)$$

where the index j runs over all legal databases DB_j. Thus one can approximate $CorrectSpecs_{\mathcal{L}}(v)$ by taking a series of databases DB_1, DB_2, ..., DB_k and using the rule

$$CorrectSpecs_{\mathcal{L}}(v) \;\approx\; \bigcap_{i=1}^{k} ConsistentSpecs_{\mathcal{L}}(v, DB_i) \tag{1}$$

For our experiments, we collected several RC databases. To estimate the recall and precision of a discovery system on a view v, we used the system to find *ProposedSpecs(v)* from one database $DB*$, and then computed recall and precision, using Equation 1 to approximate $CorrectSpecs_{\mathcal{L}}(v)$.[5] This process was repeated using different training databases $DB*$, and the results were averaged.[6] This evaluation metric is much like cross-validation; however, rather than measuring the predictive accuracy of a hypothesis on holdout data, a hypothesis is evaluated by seeing if it is 100% correct on a set of holdout databases. This rewards the hypotheses most valuable in a discovery context: namely, the hypotheses that are with high probability 100% correct.

As an example, if we report a precision of 75% for a view v, this means on average 75% of the specifications obtained by running the learner on a single database were 100% correct on *all* the sample databases. If we report a recall of 50% for v this means that on average half of the specifications that are consistent with all of the sample databases can be obtained by running the learner on a single database.

Experimental Results

The work reported in this paper is still in progress; in particular we are still engaged in the process of collecting additional datasets, and improving the augmented ADG that encodes domain-specific knowledge. In this section, we will describe a controlled study in which we compared three different Grendel2-based learning algorithms. Importantly, these algorithms (and the ADGs used with them) were based only on data from the exploratory study; thus they were developed *without any knowledge of the views used as benchmarks below*. In other words, this is a purely prospective test of the learning systems.

[5] This approximation is relatively expensive to compute because all enumerating consistent specifications must be enumerated; thus we generated only consistent *one-clause* specifications. Fortunately many RC views can be specified by a single clause.

[6] Ideally the process is repeated using every possible database for training. However, in the actual experiments, it was sometimes the case that a view was empty in one of more databases. Such databases were not used for training.

In the experiments, we used four RC databases, ranging in size from 5.6 to 38 megabytes, and 19 benchmark views. The views contain up to 209 fields, involve relations containing up to 90 fields, and contain between one and 531 tuples. While it is difficult to measure the size of a complete implementation of any single view, the views have an average of 747 non-comment lines in top-level module of their C implementations, the longest top-level C implementation is 2904 lines long, and the shortest is 210 lines. In addition to measuring the recall and precision of recovered specifications, we also measured the percentage of benchmark views for which any specifications could be recovered; this is shown in the table below in a column labeled *scope*. All of the learning algorithms described in this section abort with an error message when a problem falls outside their scope.

The result of applying Grendel2 to this set of benchmarks is shown in the first line of Table 1. For this set of problems, and using an augmented ADG written on the basis of our exploratory study, Grendel2 is able to learn specifications for just under a third of the views. When it does succeed, however, it achieves over 80% recall and precision.

We also evaluated the performance of two extensions of Grendel2. To motivate the first extension, note that when many consistent hypotheses exist, Grendel2 makes a more or less arbitrary choice among them. A useful alternative would be for the learning system to return instead the entire set $ConsistentSpecs_{\mathcal{L}}(v, DB)$.

Unfortunately, when there is insufficient training data, there may be an enormous number of consistent specifications. In these cases directly outputting $ConsistentSpecs_{\mathcal{L}}(v, DB)$ is not desirable. To address this problem we developed a *factored form* for view specifications which enables certain large sets of specifications to be compactly presented. As an example, the factored specification

v_5*(VBuffid, VAddr, VLength, VBytes)* ←
RKey=VBuffid,
p(RKey,RLoc,RLen,RBytes),
VAddr=RLoc,
(VLength=RLen ; VLength=RBytes),
(VBytes=RLen ; VBytes=RBytes).

is shorthand for four specifications, one for every possible pairing of the variables *VLength*, *VBytes*, *RLen* and *RBytes*.

The row of Table 1 labeled Grendel2/M shows results for an extension of Grendel2 that finds a factored representation of all consistent single-clause specifications. Even though Grendel2/M is restricted to single-clause view specifications, it has the same scope as Grendel2; however, the recall is now perfect. Somewhat suprisingly, Grendel2/M also has slightly better precision, although the difference is not statistically significant.

The goal of the second extension was to increase the scope of Grendel2. Our exploratory study showed that often data is not simply copied from a relation

Learner	Scope	Recall	Precision	
Grendel2	31.6%	83.3%	83.3%	
Grendel2/M	31.6%	100.0%	87.5%	
Grendel2/MD	63.1%	100.0%	61.0%	—all views
			82.2%	—views solved by Grendel2/M
			52.2%	—remaining views

Table 1: Results for Grendel2 and extensions

to a view; instead some simple conversion step is performed. For example, in the specification below, view v_6 is a copy of q with every value of "0" in the third column replaced by the empty string.

$$
\begin{aligned}
&v_6(\mathit{VProblemCode}, \mathit{VSeverity}, \mathit{VHelpBufId}) \leftarrow \\
&\quad \mathit{RPCode}=\mathit{VProblemCode}, \\
&\quad q(\mathit{RPCode}, \mathit{RLevel}, \mathit{RTextBufId}), \\
&\quad \mathit{VSeverity}=\mathit{RLevel}, \\
&\quad \mathit{zero2nullstr}(\mathit{RTextBufId}, \mathit{VHelpBufId}). \\
\\
&\mathit{zero2nullstr}(0,''). \\
&\mathit{zero2nullstr}(\mathit{Id},\mathit{Id}) \leftarrow \mathit{Id}\neq 0.
\end{aligned}
$$

Grendel2 typically fails on such views; furthermore, learning views with conversion functions like *zero2nullstr* is very difficult since a wide range of conversions are performed. We addressed this problem by extending Grendel2/M to generate view specifications that contain *determinations*. A determination between two variables X and Y is denoted $Y \prec X$, and indicates that the value of Y can be be functionally derived from the value of X. Using determinations, v_5 could be specified

$$
\begin{aligned}
&v_6(\mathit{VProblemCode}, \mathit{VSeverity}, \mathit{VHelpBufId}) \leftarrow \\
&\quad \mathit{RPCode}=\mathit{VProblemCode}, \\
&\quad q(\mathit{RPCode}, \mathit{RLevel}, \mathit{RTextBufId}), \\
&\quad \mathit{VSeverity}=\mathit{RLevel}, \\
&\quad \mathit{VHelpBufId} \prec \mathit{RTextBufId}.
\end{aligned}
$$

Specifications with determinations are even more abstract than the Datalog specifications generated by Grendel; in particular, they cannot be used to materialize a view. However, they are plausibly useful for program understanding.[7]

The row labeled Grendel2/MD of Table 1 shows results for this extension of Grendel2/M.[8] Determinations increase the scope of the learner to more than 60% but also decreases precision to around 60%—much less than the 87.5% precision obtained by Grendel2/M. However, closer examination of the data shows that the difference is less than it appears: Grendel2/MD obtains 82% recall on the problems also solvable by Grendel2/M, but only 52% recall on the problems that it alone can solve. Thus, only some of the decrease in precision appears to be due to the larger hypothesis space used by Grendel/MD; the greater part seems to be due to the fact that (in this set of benchmarks) the views requiring determinations are harder to learn from the data provided.

Interestingly, there seems to be little correlation between the complexity of the C implementation of a view and the performance of the learning systems.[9] This suggests that specification recovery methods based on learning may be a useful complement to methods based primarily on source code analysis.

Related work

Specification and design recovery has been frequently proposed in the software engineering community as an aid in maintaining or replacing hard-to-maintain code, and space does not permit a detailed discussion of all previous related work. Instead we will discuss in general terms the most important differences between our methods and known techniques.

Known techniques for extracting specifications from software rely mostly on deductive static analysis of code. (For example, see Biggerstaff [1989], Kozaczynski and Ning [1990], Breuer and Lano [1991], or Rich and Wills [1990].) The techniques of this paper share some common ground with this previous work; notably, the methods are also knowledge-based, being provided with a good deal of information about the likely form of the specifications being extracted. The primary difference from previous work is that specifications are extracted primarily using *inductive* reasoning about the behavior of the code, rather than deductive reasoning about the code itself.

One previous method that makes some use of program traces is described by Sneed and Ritsch [1993]. However, their main goal is to augment static analysis with information about dynamic properties of a program such as timing information and test set coverage, rather than to reduce the complexity of static analysis.

Our experimental results were in recovering view specifications from C code and materialized views. A good deal of research has also been devoted to the related problem of recovering logical data mod-

[7] Note also that actually materializing a view is not part of the evaluation procedure; hence these specifications can also be evaluated using the metrics of scope, recall, and precision.

[8] The grammar allows only determinations between X and a single variable Y, or between X and a constant value.

[9] The C code for the views successfully learned by Grendel2/MD averages 717 non-comment lines long, and the longest view (2904 lines) was one of those learned. Code for the unlearnable views averages 813 lines (not statistically significantly from the average length of learnable views) and the shortest unlearnable view is only 354 lines.

els from databases. However, most methods that have been proposed for this task are only partially automated [Aiken *et al.*, 1993; Hainaut *et al.*, 1993; Premerlani and Blaha, 1993], while our method is fully automatic.

Conclusions

To summarize, we have proposed using learning techniques to extract concise, high-level specifications of software as an aid in understanding a large software system. Based on the assumption that the "useful" specifications are those that with high probability exactly agree with the implementation, we outlined a method for estimating from test data the *recall* and *precision* as well as the *scope* of a learning system. Using this methodology, we demonstrated that Grendel2 can extract specifications for about one-third of the modules from a test suite with high recall and precision.

Two extensions of Grendel2 were also described which improve its performance as a discovery system: one which allows it to output a set of hypotheses, and another which allows it to learn specifications including determinations. Both extensions appear to be unique among inductive logic programming systems. In combination, these techniques allow specifications to be extracted for nearly two-thirds of the benchmark views with perfect recall, and precision of better than 60%. These results are especially encouraging because they were obtained using test cases drawn from a large (more than one million lines of C) real-world software system, and because they are a prospective test of a learning system still under development.

These results have implications both for the software engineering community and the machine learning community. Specification recovery is a potentially important application area for machine learning. The results of this paper suggest that broad classes of specifications can be extracted by currently available methods. However, the research problems encountered in specification recovery problems are quite different from those encountered in "mainstream" machine learning tasks. It seems probable that further inquiries into this area are likely to raise many topics for further research.

Acknowledgements

This research would have impossible without the data-collection efforts of Hari Vallanki, Sandra Carrico, Bryan Ewbank, David Ladd, and Ken Rehor. I am also grateful to Jason Catlett and Cullen Schaffer for advice on methodology, and to Prem Devanbu and Bob Hall for comments on a draft of this paper.

References

Aiken, P.; Muntz, A.; and Richards, R. 1993. A framework for reverse engineering of dod legacy systems. In *Working Conference on Reverse Engineering*. IEEE Computer Society Press.

Biggerstaff, Ted J. 1989. Design recovery for maintenance and reuse. *IEEE Computer* 36–49.

Breuer, P. T. and Lano, K. 1991. Creating specifications from code: Reverse engineering techniques. *Journal of Software Maintenance: Research and Practice* 3:145–162.

Cohen, William W. 1992. Compiling knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland. Morgan Kaufmann.

Cohen, William W. 1993. Rapid prototyping of ILP systems using explicit bias. In *Proceedings of the 1993 IJCAI Workshop on Inductive Logic Programming*, Chambery, France.

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

Frawley, William; Piatesky-Schapiro, Gregory; and Matheus, Christopher 1991. Knowledge discovery in databases: An overview. In Piatesky-Schapiro, Gregory and Frawley, William, editors 1991, *Knowledge Discovery in Databases*. The AAAI Press.

Hainaut, J.-L.; Chandelon, M.; Tonneau, C.; and Joris, M. 1993. Contribution to a theory of database reverse engineering. In *Working Conference on Reverse Engineering*. IEEE Computer Society Press.

Kozaczynski, W. and Ning, J. 1990. SRE: A knolwedge based environment for large scale software re-engineering activities. In *Proceedings of the 11th International Conference on Software Engineering*.

Muggleton, Stephen and Feng, Cao 1992. Efficient induction of logic programs. In *Inductive Logic Programming*. Academic Press.

Parikh, Girsh and Zvegintzov, Nicholas, editors 1983. *Tutorial on Software Maintanance*. IEEE Computer Society Press.

Pazzani, Michael and Kibler, Dennis 1992. The utility of knowledge in inductive learning. *Machine Learning* 9(1).

Premerlani, W. J. and Blaha, M. R. 1993. An approach for reverse engineering of relational databases. In *Working Conference on Reverse Engineering*. IEEE Computer Society Press.

Quinlan, J. Ross 1990. Learning logical definitions from relations. *Machine Learning* 5(3).

Rich, Charles and Wills, Linda 1990. Recognizing a program's design: A graph-parsing approach. *IEEE Software* 82–89.

Sneed, H. M. and Ritsch, H. 1993. Reverse engineering via dynamic analysis. In *Working Conference on Reverse Engineering*. IEEE Computer Society Press.

Sterling, Leon and Shapiro, Ehud 1986. *The Art of Prolog: Advanced Programming Techniques*. MIT Press.