Introducing Learning into Automatic Program Comprehension

Petri M. Gerdt and Jorma Sajaniemi

University of Joensuu, Joensuu, Finland
{petri.gerdt|jorma.sajaniemi}@cs.joensuu.fi

Abstract. Automatic program comprehension applications, which try to extract programming knowledge from program code, share many features of human program comprehension models. However, the human trait of learning seems to be missing among the shared features. We present an approach to integrate machine learning techniques into automatic program comprehension, and present an example implementation in the context of automatic analysis of roles of variables.

1 Introduction

Human program comprehension models [4, 9, 10, 16, 15] describe the scholars' view of what humans do when they try to understand what a program might be about. The models seek to explain why certain behavior happens when people ponder what code does. Human program comprehension models have also been utilized in systems that try to "understand" code automatically.

Quilici [11] defines *automated program comprehension* (APC) as the process of automatically extracting programming knowledge from source code. We will use the term automatic program comprehension as a synonym for the alternative notions used in literature: program recognition [18] and program understanding [7]. The basis of APC lays on the belief, that the programming knowledge used by the programmer of a certain program can be recognized and recovered by examining the program [7]. Even if publications about APC rarely cite research on human program comprehension it can be assumed that this assumption is based on the success of explaining programmers' cognition through different models of human program comprehension.

Cognitive science and artificial intelligence research have a long history of interaction, where cognitive theories have been implemented as computer models of reasoning (see [3] for a thorough introduction into the topic). These implementations in turn have given feedback in form of new ideas and feasibility studies to the cognitive science community. APC in its many forms continues the tradition of implementing systems based on notions of human reasoning: most implementations use data structures and algorithms that are almost directly comparable to various (human) program comprehension models. We believe that the emulation of one human trait could improve APC: machine learning [1].

The rest of the paper is organized as follows. Section 2 examines five theories of human program comprehension. The models are mirrored in automatized program comprehension and its derivates: the APC systems are after all programs, whose purpose is to understand programs like humans do. We will compare four APC approaches to human program comprehension models in Section 3. In Section 4 we present our suggestion of augumenting APCs with machine learning. Section 5 presents an example of an APC application that uses the machine learning scheme of Section 4.

2 Human Program Comprehension

Categorical syllogism is the basis of Aristotelian logic based on quantified deduction, where premises are expressed with the help of the quantifiers some, all, no, and some not [2]. Categorical syllogism provides sound theory for reasoning with quantifiers, which is used as the basis of the *mental model theory*, which seeks to explain how subjects' cognitive processes work. The mental model theory states that a subject builds a *mental model* of the world that satisfies the premises of syllogism and then inspects the model to see if some conclusion is satisfied [2]. Within this context understanding is the activity of building a mental model and verifying it.

Von Mayrhauser and Vans [17] have done a survey of mental model theories used in program comprehension research literature. They call mental model theories *program cognition models*. The purpose of a cognition model is to explain the cognitive processes of a programmer engaged in a task requiring code cognition.

We will next outline a general view of human program comprehension by summarizing five models of human program cognition: Shneiderman [15], Brooks [4], Soloway & Ehrlich [16], Letovsky [9], and Pennington [10]. The models differ in details and emphasize different parts of the comprehension process, but they do include some common elements: a representation of internal knowledge, a mental model, a comprehension strategy that directs the building of the mental model, and a definition of expert characteristics.

External and internal knowledge. Program comprehension is a process, where existing knowledge is combined with new knowledge. Knowledge can be divided into external and internal knowledge. *External knowledge* refers to any materials available to the comprehender that aid in the comprehension process, such as the program code and documentation. *Internal knowledge* is knowledge that resides within the long-term memory of the comprehender. The five models seems to share at least two kinds of internal knowledge that a program comprehender uses: program language knowledge and programming knowledge. Table 1 summarizes characteristics of internal knowledge in the different models. See [16] and [9] for detailed descriptions of how internal knowledge may be modeled.

The mental model. Comprehension involves the creation of a mental model, that represents the program under examination. Table 2 summarizes key characteristics of the mental models of five comprehension models. Two common

Table 1. Internal knowledge representation in selected models of human program comprehension.

Source	Internal knowledge representation
Shneiderman	Knowledge is divided into syntactic and semantic knowledge.
Brooks	Knowledge is arranged and accessed through index-like beacons.
Soloway & Ehrlich	Knowledge is divided into programming plans and rules of programming discourse. Three kinds of plans: strategic, tactical and implementational.
Letovsky	Like Soloway & Ehrlich, in addition: knowledge about recurring programming goals, efficacy knowledge, and domain knowledge.
Pennington	Knowledge is divided into text structure knowledge and programming knowledge.

features that all of the comprehension models share is that they have multiple levels of abstraction and at least one of the levels is language independent, i.e., the program comprehender creates a mental model of a program that is not based on any programming language syntax. In addition, the mental models are assumed to be hierarchical: the most abstract mental representation is at the top of the hierarchy and elements lower in the hierarchy are less abstract.

Table 2. Mental model characteristics.

Source	Mental model characteristics
Shneiderman	Language independent semantic representation of the program code with multiple levels of abstraction.
Brooks	A tree of hypotheses, where the most general hypothesis is the root.
Soloway & Ehrlich	A linked hierarchy of goals and plans.
Letovsky	Three layers, the most abstract first: specification layer, annotation layer, and implementational layer.
Pennington	Two distinct but interlinked parts: program model and domain model.

Comprehension strategy. During the comprehension process the comprehender creates a mental model that describes the program to be understood. The mental model of the program resides within the working memory of the comprehender. The comprehension process is directed by a *comprehension strategy*, which directs the acquisition of knowledge and the verification of the mental model. We use the term *acquisition process* as a synonym of comprehension strategy, which stresses the process nature of the comprehension process.

Different models of program comprehension describe the process of building the mental model differently, but most involve chunking and cross-referencing in some form. *Chunking* means the creation of higher-level-abstraction structures from chunks of lower-level structures. When groups of structures are recognized,

labels replace the lower-level chunks. Lower-level structures are thus chunked into larger structures at higher level of abstraction [17]. *Cross-referencing* means the relating of different levels of abstraction, building a mental model that spans all levels of abstraction in internal knowledge. Cross-referencing is an active process, which interacts with the mental model and knowledge stored in the long-term memory of the program comprehender.

There are two analogous comprehension strategies that appear in the program comprehension models. In the *top-down* approach the structures of the mental model are first formulated as general overviews of the target. The structures are then refined recursively until the mental model is detailed enough to be validated. The top-down approach starts with general knowledge, which is refined into more specific detailed knowledge. The *bottom-up* approach works vice versa, i.e., the construction of the mental model is started by formulating specific structures representing details in the target being examined. The specific structures are then linked together to form larger structures that can be validated. The bottom-up approach starts with specific knowledge, from which more general knowledge is constructed. The two approaches are not mutually exclusive, some models of program comprehension expect both top-down and bottom-up processing to happen.

Table 3 summarizes comprehension strategies of five models of program comprehension. All of the models have the common feature of cross-referencing external and internal knowledge with the mental model, and the creation of links between the different knowledge sources.

Table 3. Comprehension strategy characteristics.

Source	**Comprehension strategy characteristics**
Shneiderman	Building a mental model that covers multiple levels of abstraction.
Brooks	Hypothesis driven: reconstruction of mappings between the problem and programming domains through refinement of hypotheses.
Soloway & Ehrlich	Top-down processing of goals and plans driven by the rules of discourse: goals are chunked into sub-goals, which in turn are replaced by plans.
Letovsky	Conjecture driven, three kinds of conjectures: why, what, and how. May be top-down or bottom-up or both.
Pennington	Bottom-up processing of program code: program model is constructed by examining control structures and the domain model is constructed by examining data flow and other semantic information.

Expert characteristics. Most experiments within psychology of programming compare novice programmers and expert programmers, and most program cognition models describe what expertise or *expert characteristics* are in the context of the model. Typically experts have different knowledge acquisition strategies as well as different internal knowledge representations when compared

with novices. A definition of expert characteristics makes a model more useful for research: to verify a program comprehension model a researcher can compare how novice and expert programmers differ. Thus, an idea about expert characteristics is crucial for empirical evaluation of the models: expert characteristics serve as a hypothesis to be verified. Table 4 lists a short summary of expert characteristics of the comprehension models. Typically an expert's comprehension process works in the way that the comprehension model describes, novices differ in making errors in the process or being unable to create some mental constructs needed for program comprehension.

Table 4. Expert characteristics.

Source	**Expert characteristics**
Shneiderman	-
Brooks	Effective hypothesis verification ability.
Soloway & Ehrlich	Possesses programming plans and is able to use them.
Letovsky	A good ability to build and verify conjectures.
Pennington	Experts are able to cross-reference between domain and program knowledge.

Overview of human program comprehension. Figure 1 shows how the elements of program comprehension models discussed in this section are related by presenteing a very generalized overview of human program comprenesion as a process. The roundels represent data that is either stored outside the comprehender's memory or inside it. The gray rectangle represents action, the act of comprehension.

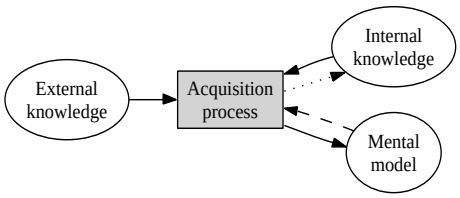

Fig. 1. Human program comprehension as a process.

The edges represent data flow: both external and internal knowledge are input to the acquisition process, whereas the mental model is the output of program comprehension. The acquisition process may add to the knowledge of the comprehender, this is represented by the dotted edge. Similarly, the building of the mental model affects, and may even direct, the acquisition process. The dashed edge represents this fact.

3 Automated Program Comprehension

APC approaches seek to automatize program comprehension to achieve some programming related goal. APC systems have many different application areas. Some maintenance related objectives of APC are the generation of documentation, refactoring (RECOGNIZE [8]) and using APC techniques to locate reusable code.The translation of a program from some language to another language can be done with an APC system, for example C programs have been translated to object-oriented C++ (Quilici's method [11]). Visualization of programming concepts may benefit from APC methods: the constructs that are to be visualized are automatically detected, thus reducing the effort of creating the visualization, or even automating the creation process. Researchers of human program comprehension can use APC systems for testing their hypotheses and theories (GRASPR [18]). APC approaches can be general purpose tools, that can potentially be used in any of the above tasks (UNPROG [7]). In addition, we believe that there are educational uses for APC systems in automatic assessment and other support systems that help students to comprehend programs.

Programming knowledge repository. Human program comprehenders have programming and domain knowledge stored in their long-term memory. This internal knowledge is used in the comprehension process. Similarly, an APC application needs a permanently stored *programming knowledge repository* of suitably encoded knowledge. Most APC systems do not deal with domain knowledge as this kind of knowledge is very specialized, and the purpose of most APCs is to analyze arbitrary programs. Programming knowledge, such as high-level language independent programming plans, is suitable for computerization. The encoding of programming knowledge differs in the level of abstraction and granularity, but all representations can be seen as encoded programming plans.

We will use the term plan as a general concept when referring to knowledge items stored in a programming knowledge repository. Other names for knowledge items include, for example, knowledge atoms [5], standard implementation plans (STIMPs) [7], abstract program concepts [8], common stereotypical computational structures or cliches [18], and templates [19]. Plans of the repository are encoded as various data structures, such as flow graphs, source code templates or logical constraints, which are typically organized as an interrelated or even recursive hierarchy. The hierarchies are typically organized as forests of tree structures that have goals as their root nodes and instructions as leafs. Knowledge repositories are typically constructed manually by humans by codifying plans found in example program samples. Table 5 presents a summary of the programming knowledge repositories of four APC systems.

The APC systems seem to mirror human program comprehending models: the long-term knowledge stored in the human brain is represented by a programming knowledge repository, which is our term for the long-term storage of programming knowledge in APC systems. The rationale of the UNPROG system serves as a good example of the resemblance of APCs and human program cognition models. The use of STIMPs to represent programming knowledge is based on the notion of planfullness: "programming is stereotyped, making fre-

Table 5. Programming knowledge repositories of selected APC systems.

System	Overview of knowledge repository
UNPROG	Standard implementation plans (STIMPs), represented by the hierarchical program model (HMODEL), an abstraction of control and data flow that is organized as a tree.
RECOGNIZE	Abstract concepts stored in a concept model. Includes concept recognition rules: information about components of sub-concepts of the concept, constraints on and among the sub-concepts.
Quilici's method	Plan definitions encoded as frames: list plan attributes. Plan recognition rules: list of components and constraints.
GRASPR	Cliches, program properties encoded in graph grammar.

quent use of standard implementations" [7]. The notion of planfullness resembles the term plan-like, which Soloway and Ehrlich [16] use to denote program code that match expert programming plans.

Program representation. The mental model that humans build and process when they are examining a program is represented by a *program representation* that is constructed by the APC during the automatic analysis. The program representation is typically a complex data structure that can be queried and otherwise manipulated. Abstract syntax trees (ASTs) constructed during the scanning and parsing of the code of a program are a popular choice for the foundation of program representations [7, 8, 11]. Data flow graphs are another program representation type that is used. Some program representations combine both data structures along with other types of information. An important feature of the program representation is that it must be somehow comparable with the programming knowledge repository of an APC. In some cases the data structure used in the program representation is used to implement the program knowledge repository. For example, a plan might be represented by a partial abstract syntax tree that could be matched against the abstract syntax tree internal representation of a source program with common tree operations.

The reason is obvious: there must be a way to compare the knowledge in the repository to the information that is being gathered about the examined program. The reader is instructed to study the summary of the program representations in Table 6 by cross-examining it with Table 5 that contains information about knowledge repositories.

Table 6. Program representations of selected APC systems.

System	Overview of program representation
UNPROG	A tree-like hierarchical program model (HMODEL).
RECOGNIZE	An AST, on which programming concepts are added.
Quilici's method	An AST, whose nodes are frames.
GRASPR	An annotated flow graph.

Algorithms. In APC systems the comprehension strategy of a human comprehender is represented by the algorithm, which searches the program representation for any instances of the plans contained in the program repository. There are two recurring general approaches to implement the matching algorithms, which are similar to those of human comprehension strategies. In the *top-down* approach the system starts from determining the goals that a program might achieve and then seeks the plans that would implement these goals [11]. Then the instructions of the plans are matched to program instructions. The *bottom-up* approach starts from the opposite direction: it starts by examining instructions of the input program, and then tries to find plans that contain these instructions. Then the algorithm tries to infer goals that the found plans might define. Quilici [11] lists some problems with the two algorithmic approaches. The top-down method needs information about the goals that the program achieves. This information is not readily accessible in real world applications. The top-down approach does not suit partial plan recognition as it only knows how to deal with plans that are connected to goals. The bottom-up method suffers from the possibility of a combinatorial explosion: each instruction may be a part of several plans, which in turn may be parts of other plans. This feature limits the size of the analyzed programs and plan hierarchies that can be used. Quilici [11] suggests that methods that limit search space may help to circumvent this problem. A weakness in both top-down and bottom-up methods is the lack of domain or program specific plans, the general plans stored in the plan library may not apply to real world application as they are bound to contain very domain specific plans.

The algorithms used in the four example APC systems are summarized in Table 7. The algorithms can be categorized according to the way that they process the program representation: a top-down algorithm will process an AST from the root to the leaves and a bottom-up vice versa. The approach used in RECOGNIZE [18] is notably different: it is based on the parsing of graph grammars.

Table 7. Overviews of the algorithms of selected APC systems.

System	**Overview of algorithm**
UNPROG	HMATCH: compares STIMPs with the program representations top-down.
RECOGNIZE	Top-down traversal of the AST, the nodes may trigger an evaluation of a concept in the concept model.
Quilici's method	A hybrid top-down, bottom-up algorithm.
GRASPR	Graph grammar parsing based algorithm.

Woods and Yang [19] prove that APC, where plans in a knowledge repository are compared with a program representation, is a NP-hard problem. They formally define the *simple program understanding problem* (SPU) [19], where the

knowledge repository is a graph of templates representing programming plans, and the program is represented by a graph. Program comprehension is done by comparing the subgraphs of the knowledge repository with the graph structures of the program representation.

Woods and Yang [19] then prove that the SPU problem is NP-hard by reduction from the subgraph isomorphism problem, which is known to be NP-hard. In addition, the authors prove that cliche, template or STIMP-based matching are NP-hard too, as they are reductions of the subgraph isomorphism problem. The APC approaches discussed in this section deal with the NP-hardness by using various heuristics, like constraints that reduce search space. Woods and Yang propose a solution that reformulates the SPU problem into a solvable constraint satisfaction problem.

Evaluation. The evaluation of an APC application should be straightforward in most cases: the APC should produce the output that a expert human program comprehender would with the same input. In other words the purpose of an APC is to mimic the behavior of expert human program comprehenders.

Table 8 summarizes the data about the evaluation of selected APC systems. None of the publications describing the APCs do include a thorough evaluation, not to speak of a formal evaluation that would be possible to replicate.

Table 8. The evaluations of selected APC systems.

System	Purpose
UNPROG	UNPROG correctly recognized 34 of 35 instances of a bounded linear search algorithm in 20 input programs. Input programs in Pascal, C, PL/I, Lisp, and pseudocode. The knowledge repository consisted of 9 STIMPs.
RECOGNIZE	Processes short programs of about 100-200 lines. The system needs an extensive collection of abstract concepts in order to be useful.
Quilici's method	Quilici tested the APC system with a relatively small library including approximatedly 100 plans and student programs. The publication does not report performance of the testing.
GRASPR	-

Overview of APC. Figure 2 shows how the elements of APC systems interact. The rounded nodes represent data and rectangles with gray background represent processes. The edges represents data flow within the process. The automatic comprehension process starts by the parsing of program code. The code is transformed into a program representation, which in turn is feeded as input to the algorithm together with the knowledge repository. The algorithm compares the constructs stored in the repository to the program representation and produces output accordingly.

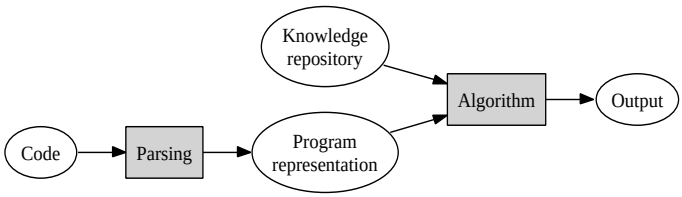

Fig. 2. An overview of APC.

4 Augumenting APC with Machine Learning

A human program comprehender learns how to read and understand programs. We believe that APC systems could be improved by emulating learning through the use machine learning algorithms [1]. Instead of building a knowledge representation explicitly, the programmers give the APC system labeled examples of the programming concept to be learned, and the APC creates an appropriate knowledge representation.

How, then, does a human look for plans in code? One characteristic of human program comprehension models is the use of beacons. According to Brooks [4] beacons are program features that indicate the presence of certain program constructs. Beacons help the comprehender to associate a hypothesis about the program to program code [17]. Soloway and Ehrich [16] think that beacons are indexes or indicators of programming plans. So, a human might look for code constructs that would serve as beacons into her programming knowledge. We believe that an APC application could use the beacon concept: the knowledge repository can represent program knowledge as sets of beacons, which we will call *features* (of programming plans, etc.). The presence of a certain feature or several features within a program that is being automatically comprehended would then indicate that the program contains a certain plan, etc.

An obvious requirement for these features is that they must be automatically detectable: it must be possible to technically specify a way to find each of the features. The features are detected with algorithms that examine the program representation of the APC.

Figure 3 shows an overview of the proposed machine learning enhanced APC process that uses automatically detectable features. The roundels represent data and rectangles with gray backgrounds phases of the process. The process is dual, it contains a *learning mode* and a *classification mode*. The edges represent data flow; the edges of the learning mode are dashed, whereas the edges belonging to the classification mode are solid.

The purpose of the learning mode is to construct the knowledge repository by giving the APC application labeled programs. The labels indicate what constructs can be found in the program; a label could for example indicate that the lines 6–14 of the program represent a certain programming plan. In the beginning of the learning mode the labeled code is parsed to a program representation. The

program representation is then analyzed in feature detection, where the features that are present in the input program are detected.

The outcome of feature detection in the learning mode is a set of labeled feature vectors, i.e., labeled lists of features. The feature vectors are input to model construction, where a computable representation of the vectors is constructed. The exact form of the representation depends of the machine learning method that is being used: the decision tree generating algorithms ID3 and C4.5 would, for example, create either a tree data structure or if-else-rulesets [1]. The model construction process may be reduced to a data preparation step in some cases: nearest neighbour type algoritms do not use a pre-computed model, so the model construction phase could perhaps only sort the feature vectors for better performance. The output of the model consruction phase is stored in the knowledge repository for future use. The model stores information about what

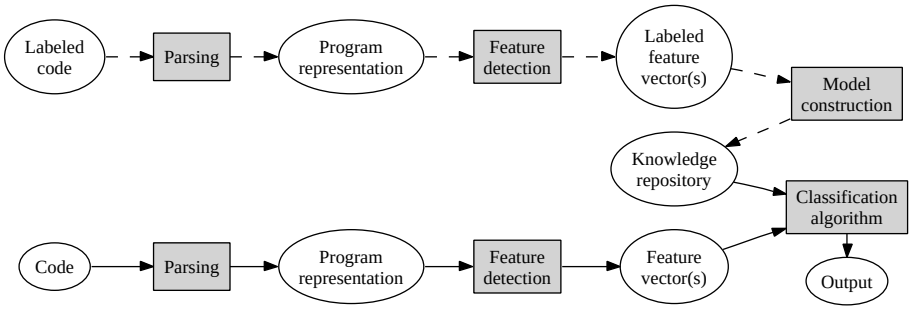

Fig. 3. An overview of machine learning enhanced automatic program comprehension.

collections of features account for what recognizable phenomena. As a naive example: the presence of a nested for-loop, and repeated array access in the loop could be labeled as "possible sorting algorithm". The learning mode may increment the knowledge repository instead of wiping out previously learned issues. The knowledge repository may include several different models, for example, representing different views of the phenomena that the application is supposed to comprehend.

When the learning mode has created a knowledge repository it is possible to execute the other mode of the dual process: the classification mode. The classification mode gets unannotated code as input, i.e., the program code does not include any hints of what could be "understood" from it. The first two phases, parsing and feature detection, are identical in the two modes. When the classification mode has produced feature vectors, the vectors are compared to the model stored in the knowledge repository with a classification algorithm. The algorithm is different for different machine learning methods: the decision tree based approaches would simply do a tree walk in this phase, whereas the nearest neighbour type algorithms would compare each of the feature vectors to those stored in the konwledge repository in order to find the nearest one(s). The

output of the whole classification mode are the labels that are assigned to the feature vectors by the classification algorithm.

The benefits of the machine learning enhanced APC architecture is that it is very modular. The features can be specified through algorithms, that are language independent. Then the concrete implementations of the algorithms can access program information through an interface to the program representation. This way the program representation implementation is independent of the algorithms and vice versa. The feature vectors serve as an interface between the feature detection phase and the phases of model construction and classification. Thus it is possible to "plug in" different machine learning methods, perhaps to find the most effective one. There are free open-source machine learning libraries and applications available that can be used.

An interesting possibility is the sharing of feature detection algorithms. If a researcher manages to create a feature detection algorithm that automates the detection of a beacon, then we believe that the chances are that this algorithm could be used in other, seemingly unrelated, program comprehension tasks too. Other APC developers could implement the algorithm and add it to the feature detection phase of their APC application.

The objective assessment of the performance of the APC systems presented in Section 3 is hard, the reported evaluations are not comparable. Machine learning addresses this problem through well established evaluation methods. For example, cross-validation is a technique which is used within machine learning to test how well a model corresponds to the training data. In simple cross-validation a training set is divided into two parts: the training set and the validation set. The application is trained with the training set, and then it is made to classify the validation set. There are several variations of cross-validation for different data sets. It must be noted that the cross-validation results are no direct implications of the classification accuracy of the application when it encounters new data outside the learning set. Nevertheless, it is a method to get comparable and constistent results with a defined training set. Cross-validation makes it possible to compare to APCs by giving them the same training set: the cross-validation accuracy is a comparable measure. Cross-validation is a good tool for the developer: it is possible to experimentally seek the features that best describe a phenomena by changing the feature set that is being detected in the feature detection phase and then performing cross-validation.

5 Automatic Detection of Variable Roles

The *roles of variables* is recent concept that describes the stereotypic behaviors of variables [12]. The roles of variables is a simple and compact way to represent higher level programming knowledge: according to an analysis [14] of nine programming textbooks 11 roles cover 99 % of variables in procedural, functional, and object-oriented programs.

The roles are quite simple; short informal definitions are enough to communicate the concept. For example, a variable that does not get a new proper value

after its initialization has the role of *fixed value*. A variable stepping through a systematic, predictable succession of values is a *stepper*; for example, a variable controlling a for-loop is a typical stepper. A *most-recent holder* is a variable that holds the latest value encountered when going through a succession of unpredictable values, or simply the latest value obtained as input. See [13] for a more comprehensive treat of the roles of variables concept.

The purpose of the automatic detection of variable roles is to find the roles of variables in arbitrary programs that include no role information. Roles are cognitive constructs and represent human programming knowledge. Different people may think of roles differently, and the exact definition of what behavior constitutes a certain role is a matter of opinion and point of view.

The ambigious nature of the role concept is one of the strengths of it as programmers can discuss variables through a vocabulary that does not restrict the discussion too much. The vague nature of cognitive constructs is reflected in the fact that the roles do not have technical definitions. The lack of definitions is the cause of a major challenge in the automatic detection of variable roles: the analysis tries to find traces of cognitive structures of the programmer in program code. This property of the problem is typical to automatic program comprehension in general.

We have designed, implemented, and validated a system that performs the automatic detection of variable roles [6] based on the framework described in Section 4. We will henceforth call it the *ADVR system*, where ADVR is an accronym of Automatic Detection of Variable Roles.

In Section 4 we presented the beacon-like concept of features that can be used in a machine learning APC system. We have developed a set of 13 features to facilitate the automatic detection of variable roles, which we use in the ADVR system. We call the features *flow characteristics* (FCs), as we are interested of the data flow affecting variables. Table 9 presents examples of the FCs with short descriptions. Each FC is detected with a separate detection algorithm. The algorithms use various static program analysis results, and some are more complicated than others in terms of computational complexity. The FCs are quite abstract in order to separate them from language dependent issues; they describe properties that commonly exist in procedural and object-oriented programming. See [6] for a comprehensive description of the FCs and their detection algorithms.

To assess the performance of the ADVR system and the set of FCs, role assignments done by the ADVR system were compared with human role assigners. The material for the validation consisted of example programs in three Pascal textbooks. The variables in the example programs were assigned roles by researchers. The validation method was leave-one-out cross-validation. The overall accuracy of the ADVR system was 93%. This can be compared with the accuracy of computer science educators who assigned roles to variables after an introduction to the role concept [14]; their accuracy with short Pascal programs was 85%. The same programs that the educators analyzed were given also to the ADVR system (using this time both the learning material and the matching material of the above validation as a large learning material). The ADVR system

Table 9. Examples of flow characteristics.

Flow characteristic	Description
Loop assignment	The variable has an assignment that is done within a loop structure.
Arbitrary sequence	The variable goes through values that are resolved dynamically at run time.
Singlepass	The variable is defined and referred to during a single pass of a loop, and the definition is not referred to in subsequent passes of the loop.
Following	The variable's value sequence follows another variable's or variables' value sequence(s).
Initial value	The definition of a variable is done before a loop, where the variable is redefined.

achieved 95% correctness. Thus, the reliability of the ADVR system seems to be comparable to that of computer science educators.

We are currently developing a new Java-based version of the ADVR system, that analyzes both Pascal and Java programs, and includes an improved flow characteristics set.

6 Conclusions

The goal of automatic program comprehension is to extract programming knowledge from program code. Automatic program comprehension applications share many features of human program comprehension models. However, the human trait of learning seems to be missing among the shared features.

In this paper we have presented a possible way to combine machine learning and APC systems. Our solution is based on the use of beacon-like features, that are detected from source code. Then machine learning techniques are used to search for patterns among the features. Our hypothesis is that different program comprehension tasks can be represented as patterns of features. Thus, the computer would mimic a human program comprehender who uses beacons as clues when trying to find out what a program does. The use of machine learning in APC transforms the problem of defining complex representations of programming knowledge into the gathering and labeling of example code.

We have implemented an APC system, which succesfully uses machine learning to detect roles of variables. Our experiences are encouraging and we believe that machine learning techniques are a promising resource to be explored in the context of automatic program comprehension.

References

1. E. Alpaydin. *Introduction to Machine Learning.* MIT Press, Cambridge, Massachusetts, USA, 2004.

2. J. R. Anderson. *Cognitive Psychology and Its Implications*. Worth Publishers, 5th edition, 2000.
3. W. Bechtel and G. Graham, editors. *A Companion to Cognitive Science*. Blackwell Publishers Ltd, 1998.
4. R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.
5. R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand a program. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE)*, pages 69–78, Honolulu, HI, 1998. IEEE Computer Society Press.
6. P. Gerdt. A system for the automatic detection of variable roles. Licenciate thesis, Department of Computer Science, Univeristy of Joensuu, Finland, 2007.
7. J. Hartman. Understanding natural programs using proper decomposition. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 62–73, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
8. W. Kozaczynski, J. Ning, and T. Sarver. Program concept recognition. In *Proc. of KBSE'92 Seventh Knowledge-Based Software Engineering Conference*, pages 216–225, Los Alamitos, CA, September 20-23 1992. IEEE Computer Society Press.
9. S. Letovsky. Cognitive processes in program comprehension. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 58–79. Ablex Publishing Company, 1986.
10. N. Pennington. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113. Norwood, NJ: Ablex Publishing Company, 1987.
11. A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, 1994.
12. J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 37–39. IEEE Computer Society, 2002.
13. J. Sajaniemi. Roles of variables home page. http://www.cs.joensuu.fi/~saja/var_roles/, 2007. (Accessed Jun. 11th, 2007).
14. J. Sajaniemi, M. Ben-Ari, P. Byckling, P. Gerdt, and Y. Kulikova. Roles of variables in three programming paradigms. *Computer Science Education*, 16(4):261–279, 2006.
15. B. Shneiderman. Empirical studies of programmers - the territory, paths, and destinations. In E. Soloway and S. Iyengar, editors, *Empirical Studies of Programmers*, pages 1–12. Ablex Publishing Company, 1986.
16. E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, SE-10:595–609, 1984.
17. A. von Mayrhauser and A. M. Vans. Program understanding - a survey. Technical Report CS-94-120, Department of Computer Science, Colorado State University, August 1994.
18. L. M. Wills. Using attributed flow graph parsing to recognize clichés in programs. In *Proceedings of the 5th International Workshop on Graph Grammars and their Application to Computer Science*, pages 170–184. Springer-Verlag, 1996.
19. S. Woods and Q. Yang. The program understanding problem: Analysis and a heuristic approach. In *Proceedings of the 18th International Conference on Software Engineering*, pages 6–15. IEEE Computer Society Press, 1996.