An Approach to Automatic Detection of Variable Roles in Program Animation

Petri Gerdt, Jorma Sajaniemi
University of Joensuu, Finland

{Petri.Gerdt|Jorma.Sajaniemi}@cs.joensuu.fi

1 Introduction

Many students have difficulties in learning to program computers. One reason that makes computer programming a difficult skill to learn is that programs deal with abstract entities, that are unrelated to everyday things. Visualization and animation is a way to make both programming language constructs and program constructs more comprehensible (Hundhausen and Stasko, 2002; Mulholland, 1998). Petre and Blackwell (1999) note that visualizations should not work in the programming language level because within-paradigm visualizations, i.e., those dealing with programming language constructs, are uninformative. Hence students learning to program benefit more from visualization of higher-level program constructs than visualization of language-level constructs.

Sajaniemi (2002) has introduced the concept of the *roles of variables* which he obtained as a result of a search for a comprehensive, yet compact, set of characterizations of variables that can be used, e.g., for teaching programming and analyzing large-scale programs. A role characterizes the dynamic nature of a variable embodied by the sequence of its successive values as related to other variables and external events. A role is not a unique task in some specific program but a more general concept occurring in programs over and over again. Table 1 gives ten roles covering 99 % of variables in novice-level, procedural programs.

Roles can be utilized in program animation to provide automatic animation of concepts that are at a higher level than simple programming language concepts. In a classroom experiment (Sajaniemi and Kuittinen, in press), traditional teaching of elementary programming was compared with role-based teaching and animation. The results suggest that the introduction of roles provides students with a new conceptual framework that enables them to mentally process program information in a way similar to that of good code comprehenders. Moreover, the use of role-based animation seems to assist in the adoption of role knowledge and expert-like programming strategies.

In this paper, we will present the role concept and the role-based program animator PlanAni, and discuss possibilities to implement automatic role analysis that is needed in order to automatically animate programs provided by students.

2 The Role Concept

Variables are not used in programs in a random way but there are several standard use patterns that occur over and over again. In programming textbooks, two patterns are typically described: the counter and the temporary. The *role* of a variable (Sajaniemi, 2002) captures this kind of behavior by characterizing the dynamic nature of a variable. The way the value of a variable is used has no effect on the role, e.g., a variable whose value does not change is considered to be a *fixed value* whether it is used to limit the number of rounds in a loop or as a divisor in a single assignment. Roles are close to what Ehrlich and Soloway (1984) call variable plans; however, Ehrlich and Soloway give only three examples and have no intention to form an complete set.

Table 1 gives informal definitions for ten roles that cover 99 % of variables in novice-level procedural programs (Sajaniemi, 2002). Each variable has a single role at any specific time, but the role may change during the execution. If the final value of the variable in the first role is used as the initial value for the next role, the role change is called *proper*. On the other

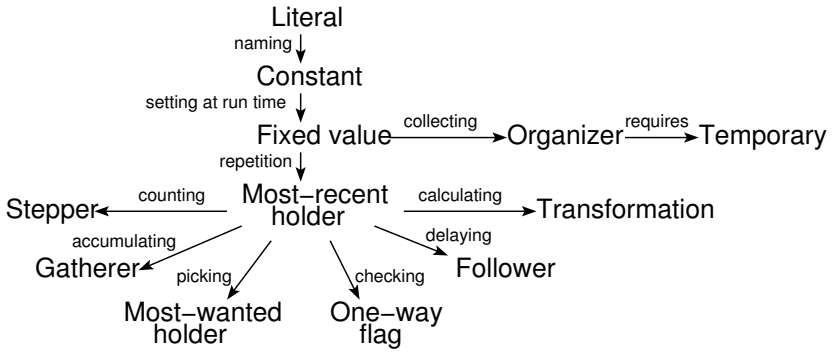

Figure 1: Role relationships. Literal and constant are programming language constructs; other nodes are the roles.

hand, if the variable is re-initialized with a totally new value at the beginning of the new role phase, the role change is said to be *sporadic*.

Roles can be used to describe the deep meaning of programs: what is the purpose of each variable and how do the variables interact with each other. For example, calculating the average of input values requires a *most-recent holder* for the input values, a *gatherer* for the running total, and a *stepper* for counting the numer of input items. Figure 1 describes the connections between roles that can be used as a basis for introducing the roles in elementary programming classes.

Table 1: Roles of variables in novice-level procedural programming.

Role	Informal description
Fixed value	A variable initialized without any calculation and not changed thereafter.
Stepper	A variable stepping through a systematic, predictable succession of values.
Follower	A variable that gets its new value always from the old value of some other variable.
Most-recent holder	A variable holding the latest value encountered in going through a succession of values, or simply the latest value obtained as input.
Most-wanted holder	A variable holding the best or otherwise most appropriate value encountered so far.
Gatherer	A variable accumulating the effect of individual values.
Transformation	A variable that always gets its new value with the same calculation from values of other variables.
One-way flag	A two-valued variable that cannot get its initial value once its value has been changed.
Temporary	A variable holding some value for a very short time only.
Organizer	An array used for rearranging its elements.

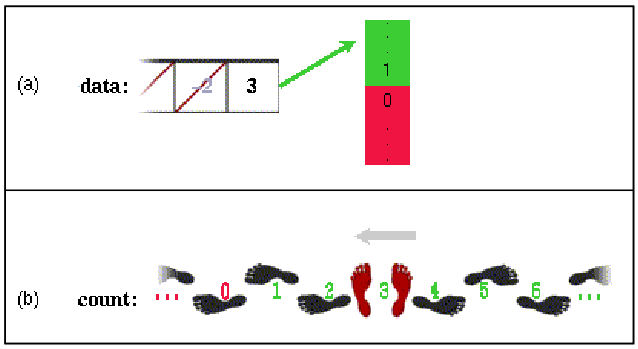

Figure 2: Visualizations of the same operation for different roles: comparing whether a *most-recent holder* (a) or a *stepper* (b) is positive.

3 The PlanAni Program Animator

We have implemented a program animator, PlanAni, that is based on the role concept (Sajaniemi and Kuittinen, 2003). In PlanAni, each role has a visualization—role image—that is used for all variables of the role. Role images give clues as to how the successive values of the variable relate to each other and to other variables. For example, a *fixed value* is depicted by a stone giving the impression of a value that is hard to change, and a *most-wanted holder* by flowers of different colors: a bright one for the current value, i.e., the best one found so far, and a gray one for the previous, i.e., the next best, value. A *most-recent holder* shows its current and previous values, also, but this time they are known to be unrelated and this fact is depicted by using two squares of a neutral color.

In addition to role images, PlanAni utilizes role information for role-based animation of operations. As the deep meaning of operations is different for different roles, PlanAni uses different animations. For example, Figure 2 gives visualizations for two syntactically similar comparisons "`some_variable > 0`". In case (a), the variable is a *most-recent holder* and the comparison just checks whether the value is in the allowed range. In the visualization, the set of possible values emerges, allowed values with a green background and disallowed values with red. The arrow that points to that part of the values where the current value of the variable lays, appears as green or red depending on the values it points to. The arrow flashes to indicate the result of the comparison.

In Figure 2(b) the variable is a *stepper* and, again, the allowed and disallowed values are colored. However, these values are now part of the variable visualization and no new values do appear. The values flash and the user can see the result by the color of the current value. In both visualizations, if the border value used in the comparison is an expression (as opposed to a literal value), the expression is shown next to the value.

Figure 3 is an actual screenshot of the PlanAni user interface. The left pane shows the animated program with a color enhancement showing the current action. The upper part of the right pane is reserved for variables, and below it there is the input/output area consisting of a paper for output and a plate for input. The currently active action in the program pane on the left is connected with an arrow to the corresponding variables on the right. frequent pop-ups explain what is going on in the program, e.g., "*creating a gatherer called sum*". Users can change animation speed and the font used in the panes. Animation can proceed continuously (with pauses because the frequent pop-ups require clicking "Ok" button, or pressing "Enter") or stepwise. Animation can be restarted at any time but backward animation is not possible.

PlanAni is implemented using Tcl/Tk and it has been tested both on Linux/Unix and Windows NT. The architecture consists of four levels as depicted in Figure 4. The lowest level takes care of primitive graphics and animation, and implements the user interface. The next level knows how to animate the smallest actions that are meaningful to viewers of the animation. This level is language independent in the sense that it can be used to animate

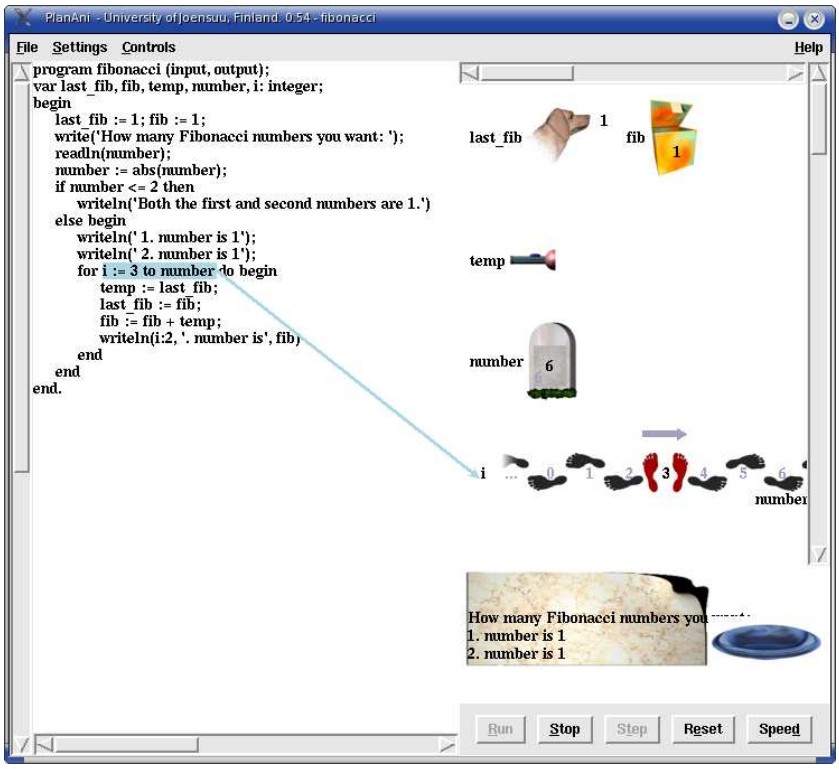

Figure 3: A screenshot of the PlanAni user interface.

Automatic role analysis
Program–level animation
Statement–level animation
Primitives for graphics

Figure 4: Architectural levels of PlanAni.

programs written in various languages, e.g., Pascal, C, and Java.

The third level takes as input a program to be animated, annotated with the roles of variables and possible role changes, and animates the program automatically. Finally, the uppermost level does not need role information because it finds roles automatically. Currently, the two uppermost levels are not implemented. As a consequence, animation commands must be authored by hand for each program to be animated. Typically 5 animation lines are required for each line in an animated program.

The uppermost level of PlanAni will accept programs without role annotations. It should make a dataflow analysis and assign roles and role changes based on this analysis. Its output—the input to the third level—looks like the program in Figure 5 with two variables that are initially *most-recent holders*. When the second loop begins, the role of the variable `count` changes to *stepper*. Since the last value of the old role is used as the first value in the new role, the role change is marked as proper.

The implementation of the fourth level is challenging for two reasons. First, roles are cognitive concepts which means that different people may assign different roles to the same variable. There is also the possibility of repeating phases of the same behavior making automatic detection of, e.g., *one-way flag* behavior hard. Second, the recognition of roles requires extensive dataflow analysis which is a research area of its own.

```
program doubles (input, output);
var count{MRH}, value{MRH}: integer;
begin
    repeat
        write('Enter count: '); readln(count)
    until count > 0;
    while count{proper:STEP} > 0 do begin
        write('Enter value: '); readln(value);
        writeln('Two times ', value, ' is ', 2*value);
        count := count - 1
    end
end.
```

Figure 5: A Pascal program with role annotations.

4 Automatic Detection of Roles

Roles are cognitive constructs and represent human programming knowledge. An automatic analysis of roles takes as input computer programs and tries to assign roles to the variables. The automatic analysis must find a connection between the program code and the cognitive structures of the programmer. The fact that human cognition is not exact whereas program code is exact makes this task a challenging one.

A related example of extracting programming related information automatically is the PARE (Plan Analysis by Reverse Engineering) system (Rist, 1994). PARE extracts the plan structure of an arbitrary Pascal program that it gets as input. A plan is a series of actions that achieve a goal, which PARE defines as a sequence of linked lines of code. Lines of code that either use or control another lines are linked, and define a sequence of lines, which constitute a plan. A program may have many plans, which together build up the plan structure of the program. PARE deals with larger constructs when compared to automatic role analysis: the plans detected by PARE represents a view to the solution of a problem, whereas automatic role analysis essentially searches program code for clues of stereotypical usage of variables.

The primary objective of automatically assigning roles to variables can be divided into two subgoals: to automatically find characteristics of a variable from a source program ("dataflow analysis" of the identification phase in Figure 6); and to map these characteristics to a certain human understood role ("Matching" in Figure 6).

The first subgoal requires customization of compiler and dataflow analysis methods and techniques (Aho et al., 1988; Nielson et al., 1998) in order to extract information about how data flows through a variable; this information is compressed into a flow characteristics description (FC). The second subgoal presupposes the creation of a database that contains mapping information between human defined variable roles and flow characteristics. The creation of this database is called the learning phase in Figure 6. The database is obtained by taking a set of existing programs with roles annotated in the style of Figure 5, and applying the same dataflow analysis technique as in the identification phase. With this method, a set of flow characteristics can be attached to each role, and some machine learning mechanism ("Learning" in Figure 6) can then be used to generalize the results into a role–FC database.

Role definitions may differ from person to person and it is probable that programmers with different backgrounds and experience levels produce different mappings between the role concept and actual program code. By asking programmers with different backgrounds to assign roles to the programs used in the learning phase, it becomes possible to analyze the differences in the resulting databases, i.e., differences of the programmers' mental models.

The complexity of the extraction of FCs differ, some can be determined with a relative

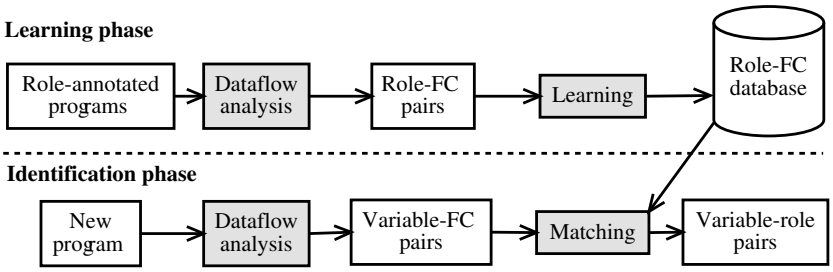

Figure 6: An overview of the learning and matching phases in automatized variable role detection.

simple syntax analysis, others need more complicated analyses of the data flow. The whole process of program analysis begins with the scanning of the program, during which the program is partitioned into tokens, such as reserved words and strings. These tokens are then processed through syntax analysis, in which the tokens are matched to the syntax of the programming language. The variables of the program are located during the scanning process and basic information about their type and assignments are found during syntax analysis.

Some examples of FCs are *related variables* and *value scaling*. The related variables charasteristic indicates how many variables affect the value of the variable under examination. The value scaling charasteristic tells that the value of a variable is scaled by some constant or variable in some context during its lifetime. The related variables FC can be determined during syntax analysis by simply recording all variables that appear on the right hand side of assignments. Syntax analysis also produces data that can be used to find out whether the value of a variable is scaled. The simplest case of value scaling is the modification of the assigned value by some constant, such as in "`x := x + 1`". In a more complex case value scaling may include two variables on the right hand side of an assignment, where the other represents the scaling factor. Both related variables and value scaling may happen in the same assignment; in general FCs are not exclusive, but rather conditions which may apply in association with each other.

The extraction of some FCs require the examining the execution order of a program. For example the FC called *change frequency* records how many times the value of a variable changes during its lifetime. This information requires that program execution is simulated in order to find out how many assignment are made to a certain variable. The lifetime of a variable is an important source of information for determining FCs. An example of a dataflow analysis technique, which is needed when examining the lifetime of a variable is the live variables analysis (Nielson et al., 1998). A variable is said to be *live* at a certain point of program execution if the value assigned to it will be used later as the execution proceeds. The live variables analysis may show that a value assigned to a variable in the beginning of a program is not used at all, in fact the next use of the variable is independent of the previous value. In this case the variable may have two different roles during its lifetime. If a variable is not live at a certain point of a program, then it may not have an active role at that time. A variable may in fact have many different roles during its lifetime, thus the lifetime of a variable and the lifetime of a role are two different things.

Another reason for examining the execution order is the fact that the sequence in which the FCs of a variable appear are important. Consider for example the variable `count` in the program of Figure 5. The change frequency of `count` is determined to be frequent, as it is assigned values repeatedly inside two loops. On the other hand the value scaling FC apply to the variable `count` too, as its value is modified with the statement "`count := count - 1`" in the while loop. If the appearance order of the FCs are not considered, then the role of the variable `count` looks like a *stepper*. A *stepper* can be identified by the combination of a frequent change frequency and the value scaling FCs.

If the sequence in which the FCs of the variable `count` appear is considered, then we can

identify two groups of FCs. The first group includes only the frequent change frequency FC (in the repeat loop) and the second group includes a pair of FCs: frequent change frequency and value scaling (in the while loop). The fact that the FCs of `count` can be grouped into two distinct groups suggests that `count` might have two different roles during its lifetime. The latter group identifies the role of *stepper* as discussed above. The former group including only the change frequency FC needs an additional FC to indentify a role: *user input*, which indicates that the value of the variable is dependant on user input. Thus we get a grouping of the FCs frequent change frequency and user input, which suggests that the role of the variable `count` is *most-recent holder* during the first loop of the program in Figure 5.

Certain roles appear in pairs, which adds a new dimension to the automatic detection of roles. For example a variable with the role of an *organizer* appear often with an another variable with the role of a *temporary*. The *temporary* variable is used to facilitate the re-organizing of the *organizer* variable. An another example of a role pair is a *gatherer*, whose gathering activities are controlled by a *stepper*.

Our initial prototype will analyze Pascal programs and deal with only three roles: *fixed value*, *stepper*, and *most-recent holder*. These three roles covered the majority of variables (81.7–90.3 %) in a study of three Pascal programming textbooks (Sajaniemi, 2002). The implementation is done using Tcl/Tk and it is based on Yeti (Pilhofer, 2002), a Yacc-like compiler-compiler (Aho et al., 1988).

5 Conclusion

In this paper we have presented the concept of variable roles, which are a set of characterizations of variables. The concept explicitly embodies programming knowledge in a compact way that is easy to use in teaching programming to novices. The PlanAni program animator visualizes the roles in a program by attaching a role image to each variable and animating operations on the variables according to their roles. This way the program visualizations that the PlanAni animator produces goes further than the mere surface structure of the program.

Automatic role detection is needed in order to make automatic role-based animation of arbitrary programs possible. The combination of a non-exact cognitive concept and the exact nature of programming languages makes this task a non-trivial one. We suggest that automatic role analysis is possible by performing dataflow analysis combined with a machine learning strategy. Automatized role analysis also makes it possible to deal with the roles of variables in large-scale programs, which may provide interesting opportunities for large-scale program comprehension.

Acknowledgments

This work was supported by the Academy of Finland under grant number 206574.

References

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1988.

K. Ehrlich and E. Soloway. An Empirical Investigation of the Tacit Plan Knowledge in Programming. In J. C. Thomas and M. L. Schneider, editors, *Human Factors in Computer Systems*, pages 113–133. Ablex Publishing Co, 1984.

S. A. Hundhausen, C. D. Douglas and J. T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing*, 13:259–290, 2002.

P. Mulholland. A Principled Approach to the Evaluation of SV: A Case Study in Prolog. In J. Stasko, J. Dominique, M. H. Brown, and B. A. Price, editors, *Software Visualization – Programming as a Multimedia Experience*, pages 439–451. The MIT Press, 1998.

F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Heidelberg, 1998.

M. Petre and A. F. Blackwell. Mental Imagery in Program Design and Visual Programming. *International Journal of Human-Computer Studies*, 51(1):7–30, 1999.

F. Pilhofer. YETI - Yet another Tcl Interpreter. Internet WWW-page, URL: http://www.fpx.de/fp/Software/Yeti/, 2002. (March, 2004).

R. S. Rist. Search Through Multiple Representations. In D. J. Gilmore, R. L. Winder, and F. Detienne, editors, *User-Centred Requirements For Software Engineering Environments*. Springer-Verlag, New York, 1994.

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.

J. Sajaniemi and M. Kuittinen. Program Animation Based on the Roles of Variables. In *Proceedings of the ACM 2003 Symposium on Software Visualization (SoftVis 2003)*, pages 7–16. Association for Computing Machinery, 2003.

J. Sajaniemi and M. Kuittinen. An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education*, in press.