On the Concept of Variable Roles and its Use in Software Analysis

Yulia Demyanova, Helmut Veith, Florian Zuleger
Vienna University of Technology

***Abstract*—Human written source code in imperative programming languages exhibits typical patterns for variable use, such as flags, loop iterators, counters, indices, bitvectors, etc. Although it is widely understood by practitioners that these patterns are important for automated software analysis tools, they are not systematically studied by the formal methods community, and not well documented in the research literature. In this paper, we introduce the notion of variable roles on the example of basic types (int, float, char) in C. We propose a classification of the variables in a program by variable roles which formalises the typical usage patterns of variables. We show that classical data flow analysis lends itself naturally both as a specification formalism and an analysis paradigm for this classification problem. We demonstrate the practical applicability of our method by predicting membership of source files to the different categories of the software verification competition SVCOMP 2013.**

I. Introduction

Programs written in imperative programming languages, such as C, Java, Perl, Python, share typical patterns of variable use, like flags, loop iterators, counters, indices, bitvectors, temporary variables, and so on. Experienced programmers have informal knowledge of these patterns, to which we refer as *variable roles*. For example, from the piece of C code `while(i<n) a[i++]=0;`, it can be deduced that `i` is a loop iterator and an array index. Similarly, from the statement `x&=y`, we can infer that `x` is a bitvector.

In common programming languages, there is no direct mapping from data types to roles - multiple roles can be associated with the same type. For example, in C, the type `int` can be used to store such different values as boolean, file descriptor, bitvector, and character literal. Moreover, it is not clear how to extend standard type systems for languages like C to express roles like array index, counter, and loop iterator. Additionally, one variable can have several roles simultaneously, like the variable `i` in the loop example above. In type systems, in contrast, one variable must be assigned one and only one type. Therefore, roles can not be considered simply as refined types.

Information about variable roles is implicitly contained in the structure of the source code, thus the roles can often be inferred by syntactic analysis. This can be done by analysing the expressions or statements of a given kind, for example, matching array indices in array subscripts. Alternatively, roles can be inferred by searching for code patterns, for example, `t=x; x=y; y=t;` is a typical pattern for a temporary variable `t`.

The notion of a variable role has two dimensions. In general, variable roles represent heuristics, which means that they can be systematically studied and analysed, but they need to be treated as auxiliary heuristic information. Thus, variable roles can *guide* a verification tool, but the *soundness* of a formal analysis must not depend on variable roles. Certain variable roles, however, provide *sound* information, which can be *relied upon* during verification, and thus these roles can be viewed as types. We will explore these two dimensions of variable roles in future work.

In this paper we define 14 variable roles with a standard dataflow analysis. Our definition serves at the same time as an algorithm to compute the roles. In order to choose the roles, we have manually investigated 5.2 KLOC of C code from the cBench benchmark [1]. We assigned roles to the variables of basic types such as int, float and char. When choosing the roles, we were inspired by typical programming patterns for variable use in real life programs. We have chosen the roles in such a way that a small number of roles is able to classify each occurring program variable in the programs we considered. We have implemented a prototype of a tool which maps basic-type variables in C programs to sets of roles.

As this short paper is reporting work in progress, we are currently exploring applications for variable roles. An important natural application is the use of variable roles to create abstractions in software verification or choose abstract domains through a better understanding of the program. For example, in C programs integer variables are used to store boolean flags, because there is no boolean type. When creating an abstraction for a C program, we know that the predicate `x==0` provides sufficient information about a boolean variable `x`. However, most state-of-the-art verification papers consider a program as a logical formula and either ignore such implicit information, or treat it as undocumented heuristics. For example, the ASTREE static analyser [2] relies heavily on human insight for selecting the right domain. Variable roles could be used for automating this process, e.g., to suggest the use of octagon or polyhedra domains for variables which occur only in linear operations, BDDs for boolean variables, etc. This will save a verification tool from enumerating all possible domains. After submission of this paper, we learnt about current work in this direction by the developers of the CPAchecker verification tool [3].

Another important application area of our method is to classify source files, for example, from benchmarks for different verification competitions, according to the relative number of occurrences of variable roles in them. To demonstrate

ISBN 978-0-9835678-3-7/13. Copyright owned jointly by the authors and FMCAD Inc.

```
int x, y, n =^1 0;
...
y =^2 x;
while (x)
{
    n++^3;
    x =^4 x &^5 (x-1);
}
```
a) bitvector, counter, iterator

```
int fd =^1 open(path, flags);
int c, val =^2 0;

while (read^3(fd, &c, 1) > 0
    && isdigit^4(c))
{
    val =^5 10*val + c-'0';
}
```
b) character, file descriptor, linear

Fig. 1: Different patterns of use of integer variables

this point, we performed the following experiment with the benchmarks from the software competition SVCOMP 2013 [4]. The competition distinguishes several categories of source files, such as device drivers, embedded systems, concurrent programs, and so on. The classification is done by human experts, who manually analysed and comprehended the source code. With our tool we computed the frequency of different roles in each category and used this data to train a machine learning tool to predict the competition categories for new files. In a number of experiments, we randomly selected a subset of the competition source files for training and used the remaining source files to check our prediction against the human classification. Importantly, we used a machine learning technique not to infer roles, but rather to validate their predictive power. The results of the experiments are encouraging - the prediction is successful in more than 80% of the cases. We highlight that our choice of the roles was based on examples from cBench rather than SVCOMP.

The results of our experiment suggest that variable roles can be used to interpret experimental results for current verification tools. The strengths and weaknesses of the tools can be identified by computing code metrics in terms of the relative frequencies of the roles. This idea can be further elaborated for building a portfolio-solver where we first analyse the variable roles of the program under scrutiny and then select the verification tool that is best suited.

Contributions:

- We identify 14 variable roles that commonly occur in practical programs.
- We implement a prototype of a tool which assigns a set of roles to every basic-type variable in a C program.
- Using our tool and a machine learning technique, we predict the membership of a C program to a category of the SW verification competition SVCOMP 2013. We get encouraging results in a number of experiments.

II. FORMALISATION OF VARIABLE ROLES

A. Examples

We will use the C programs of Figure 1 to informally introduce variable roles, whose formal definitions are given later in the section. In the programs we have assigned labels to the statements and expressions to which we refer from the text.

The program in Figure 1a calculates the number of non-zero bits of the variable `x`. In every loop iteration, a non-zero bit of `x` is set to zero and the counter `n` is incremented. The loop continues until all bits are set to zero. Although the variables `x` and `n` are declared of the same type `int`, they are used differently. For a human reading the program, the statements `n=0` and `n++` in the loop body signal that `n` is a counter. Indeed, `n` is used to count the number of loop iterations. On the other hand, the value of the variable `x` as an integer is not used in calculations, but rather individual bits in its binary representation matter.

We define the roles by restricting the operations in which a variable occurs. We require that a bitvector occurs in at least one bitwise operation (bitwise AND, OR or XOR), like the variable `x` in expression 5. We require that a counter variable only changes its value in an increment or decrement statement or gets assigned zero. The variable `n`, which is assigned in statements 1 and 3, satisfies these constrains.

The program in Figure 1b reads a decimal number from a text file and stores its numeric representation in the variable `val`. In contrast, the variables `fd` and `c` are used to store the output of the library functions `open()` and `read()` respectively. The difference between the two variables is that `c` is later used in calculations, while `fd` is only passed to the function `read()` as a black box because its value does not directly affect the result of the computations. One can conjecture that `c` is a character, because it is passed as an input to the function `isdigit()`, which checks whether its parameter is a decimal digit character. Even though `isdigit()` is declared to take a parameter of type `int`, the documentation states that the parameter is a character to be tested, cast to int [5].

We define character, file descriptor and linear roles as follows. We require that a character variable is assigned at least once a character literal (e.g., `c='a'`) or another character variable, or is used in a standard C function for manipulating characters (e.g., `c=getchar()` or `isdigit(c)`). A file descriptor is required to be used in a standard C function for manipulating files (e.g., `fd=open(path,flags)` or `read(fd,&c,1)`). A linear variable can be assigned only linear combinations of linear variables. The variables `c` and `val`, assigned in statements 3 and 2,5 respectively, satisfy the latter constraint.

B. Definition of the analysis

We define variable roles using classical intraprocedural dataflow analysis [6]. In this section we use the notation as follows. **Var** denotes the set of program variables, and **Num** denotes all scalar constant literals (e.g., 0, 0.5, 'a'). **S**, **E** and **B** denote the set of program statements, arithmetic and boolean expressions respectively. For the elements of these sets we use the same names in the lowercase version (e.g., var for a program variable).

For a program s∈ **S** the result of analysis R is computed using the function Res^R, which is defined as follows:

$$Res^R = Init^R \bigsqcup gen^R(s),$$

$$
\begin{array}{rcl}
\textbf{BITVECTOR} & & \text{Init} = \emptyset,\ \bigsqcup = \cup,\ \text{c} = \mathbf{o} \\
\text{gen(var := e)} & = & \begin{cases} \{\text{var}\} & \text{if e ::= } e_1 \text{ bitop } e_2 \\ \emptyset & \text{otherwise} \end{cases} \\
\text{gen(if b then } s_1 \text{ else } s_2) & = & \text{gen(b)} \cup \text{gen}(s_1) \cup \text{gen}(s_2) \\
\text{gen}(s_1;\ s_2) & = & \text{gen}(s_1) \cup \text{gen}(s_2) \\
\text{gen(skip)} & = & \emptyset \\
\text{gen(while b do s)} & = & \text{gen(b)} \cup \text{gen(s)} \\
\text{gen(var) = gen(num)} & = & \emptyset \\
\text{gen}(e_1 \text{ bitop } e_2) & = & \text{IsVar}(e_1) \cup \text{IsVar}(e_2) \\
 & & \cup\ \text{gen}(e_1) \cup \text{gen}(e_2) \\
\text{gen}(e_1 \text{ arithop } e_2) = & = & \text{gen}(e_1) \cup \text{gen}(e_2) \\
\text{gen(bitnot e)} & = & \text{IsVar(e)} \cup \text{gen(e)} \\
\text{IsVar(e)} & = & \begin{cases} \{\text{var}\} & \text{if e ::= var} \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{LINEAR} & & \text{Init} = \textbf{Var},\ \bigsqcup = \setminus,\ \text{c} = \mathbf{f} \\
\text{gen(var:=e)} & = & \begin{cases} \{\text{var}\} & \text{if lin(e)=false} \\ \emptyset & \text{otherwise} \end{cases} \\
\text{gen(if b then } s_1 \text{ else } s_2) & = & \text{gen}(s_1) \cup \text{gen}(s_2) \\
\text{gen}(s_1;\ s_2) & = & \text{gen}(s_1) \cup \text{gen}(s_2) \\
\text{gen(skip)} & = & \emptyset \\
\text{gen(while b do s)} & = & \text{gen(s)} \\
\text{gen(e)} & = & \emptyset \\
\text{lin(num)} & = & \text{true} \\
\text{lin(var)} & = & \begin{cases} \text{true} & \text{if var} \in \text{Res}^{LINEAR} \\ \text{false} & \text{otherwise} \end{cases} \\
\text{lin}(e_1 + e_2) & = & \text{lin}(e_1) \wedge \text{lin}(e_2) \\
\text{lin}(e_1 * e_2) & = & \begin{cases} \text{lin}(e_2) & \text{if } e_1 \in \textbf{Num} \\ \text{lin}(e_1) & \text{if } e_2 \in \textbf{Num} \\ \text{false} & \text{otherwise} \end{cases} \\
\text{lin}(e_1 \text{ bitop } e_2) & = & \text{lin(bitnot e)} = \text{lin}(e_1/e_2) = \text{false}
\end{array}
$$

Fig. 2: Formal definition of roles BITVECTOR and LINEAR

where $Init^R \in \mathcal{P}(\mathbf{Var})$ is the *initial* set of variables, the function $gen^R : \mathbf{S} \cup \mathbf{E} \cup \mathbf{B} \rightarrow \mathcal{P}(\mathbf{Var})$ maps every statement and expression to a set of *generated* variables, and the sign $\bigsqcup$ is used as a placeholder for a set operation and must be instantiated for each analysis.

Analysis R is therefore defined by a tuple ($Init^R$, $\bigsqcup$, gen^R, c), where c$\in \{\mathbf{f}, \mathbf{o}\}$ indicates how to evaluate Res^R. When c is defined as **f**, a fixed point of Res^R is computed, i.e. Res^R is iteratively recalculated until it does not change. When c is defined as **o**, Res^R is calculated in one iteration.

C. Example of role definition

In Figure 2 we formally define the analysis for the roles BITVECTOR and LINEAR. Due to the lack of space we give only an informal definition of the remaining roles in Table I. We now show step by step the computation of the BITVECTOR and LINEAR roles for the example program in Figure 1a.

The analysis for the role BITVECTOR starts with the empty set ($Init = \emptyset$). The operation $\bigsqcup$ is defined to be set union, and the result set is calculated in one iteration (c = **o**). When statement 4 is processed, the variable `x` is added to the result set because in this statement `x` is assigned the result of a bitwise AND operation. At expression 5, the variable `x` is also added to the result set because `x` occurs in a bitwise operation. After that, the result set does not change anymore, and the analysis yields the result {`x`}, as shown in Figure 3a.

TABLE I: Informal definition of variable roles

Role name	**Informal definition**
SYNT_CONST	not assigned any value in the program
CONST_ASSIGN	assigned only numeric literals or CONST_ASSIGN variables
COUNTER	assigned only in increment and decrement statements, or assigned zero
LINEAR	assigned only linear combinations of LINEAR variables
BOOL	assigned only 0,1, the result of a boolean expression or BOOL variables
INPUT	modified by an external function
BRANCH_COND	occurs in the condition of an if statement
BITVECTOR	occurs in a bitwise operation or assigned the result of a bitwise operation
UNRES_ASSIGN	assigned a pointer dereference or modified by a function
CHAR	assigned only character literals, CHAR variables, or passed to a library function which manipulates characters
LOOP _ITERATOR	occurs in the condition of a loop and assigned in the loop body
FILE_DESCR	passed to a library function which manipulates files
ARRAY_INDEX	occurs in an array subscript expression
ARRAY_SIZE	passed to a memory allocating library function

BITVECTOR	
label	gen(s)
4,5	{x}
Init(vd)=∅, Res={x}	

a) bitvectors

LINEAR		
Iter.	label	gen(s)
1	4	{x}
2	2	{y}
Init(vd)={x,y,n}, Res={n}		

b) linear variables

Fig. 3: Step-by-step computation of roles

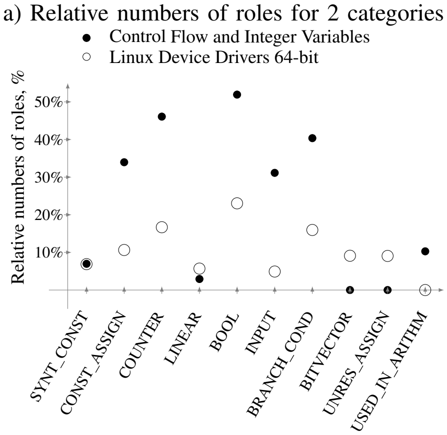
The analysis for the role LINEAR is computed as a fixed point of the function Res^R (c = **f**). It starts with the set **Var** of all program variables, which evaluates to {`x`,`y`,`n`}. The operation $\bigsqcup$ is defined to be set subtraction. In the first iteration, the variable `x` is excluded from the result set at statement 4 because it is assigned a non-linear expression. In the second iteration, the variable `y` is excluded from the result set at statement 2 because it is assigned `x`, which does not belong to the result set. In the third iteration, the result set does not change, and the analysis yields the result {`n`}, as shown in Figure 3b.

III. Implementation and experiments

We used the clang compiler [7] to implement a prototype of a tool, which assigns a subset of variable roles to every basic-type variable. The current implementation does intraprocedural analysis and does not include a pointer analysis. We replace all function calls (e.g., `c=getchar()`) and pointer dereferences (e.g., `n=*ptr`, `n=arr[i]`) with fresh variables. For example, the statement `c=getchar()` would be rewritten as `c=`t_1, and in the LINEAR analysis the variable t_1 would not be excluded from the result set, but rather assigned the role "unresolved assignment", which is a trade-off between soundness and precision.

We ran two experiments. In the first one we computed the relative number of the occurrences of each role in every category. We calculated it by summing up the numbers in all files of a category and normalising them by the total number of variables in these files. The results for the categories "Control

a) Relative numbers of roles for 2 categories



b) Prediction error in different settings

Training set, % of all files	Error of choice 1	Error of choice 2
90%	15.94%	2.90%
80%	14.81%	5.93%
70%	16.20%	7.98%
60%	19.77%	7.98%
50%	18.60%	8.54%

Fig. 4: Comparison of categories and automatic classification of files

Flow and Integer Variables" and "Linux Device Drivers" are shown in Figure 4a. We observed higher frequencies of boolean flags and branching operations, counters, arithmetic operations and constant assignments in the first category, and high numbers (in comparison to other categories in SVCOMP) of bitvectors and pointers in the second one.

In our main experiment, whose summary was given in Section I, we used machine learning to create a classifier for source files into the categories of the competition SVCOMP as a function of the frequencies of variable roles in a file. Since a program would typically share similarities with several categories, we used a multiclass vector support machine [8] to predict the category of a source file with some probability. For example, we predict that a file is a driver with the probability 60%, a concurrent program with the probability 35%, and so on. We translated the relative numbers of roles into the input format of the machine learning tool Weka [9] as follows: each source file represented one training example with the category corresponding to the class, and relative numbers of roles representing the vector of float attributes. We ran the experiments for varying sizes of the training sets from 90% to 50% of all files and analysed the remaining files by our tool. Figure 4b shows the percentage of the files for which the most likely category (second column) or the two most likely categories (third column) do not include the actual file category.

IV. Related work

The term variable roles was inspired by the work in program comprehension [10] which informally defines roles as *patterns of how variables are initialised and updated*. The authors have defined nine roles, implemented a tool for assigning roles to variables using static analysis, and evaluated it on Pascal programs from textbooks. The work leaves open the question of formalising the notion of variable roles as well as of the possibility of applying the method to real-word programs.

The commercial bug finding tool Coverity [11] uses implicit knowledge in the form of programmer's *beliefs*, i.e. propositional statements about program variables and functions. The authors use static analysis to extract two types of statements – the sound statements which follow from the requirements of safety, non-redundancy and reachability of the code (e.g., "a pointer is not null") and hypotheses which follow from the statistics of observations (e.g., "calls to functions *f()* and *g()* should be paired").

Rondon et al. [12] use predicate abstraction over a fixed set of predicates to infer so called *liquid* types, i.e. types refined with a conjunction of propositional predicates (e.g., $x{>}0 \wedge x{<}5$). We consider this approach to be complementary to ours, because it does not use any information from the source code other than the transition relation, and concentrates on arithmetic properties of variables.

Variable names and comments as an additional source of knowledge about a program have been systematically studied in program comprehension. The *Latent Semantic Indexing* technique [13] allows to query the program source code using words in natural language, based on the number of occurrences of the words in variable names and comments. A study has been made of the naming rules for variables in real-word programs [14], and of expanding abbreviated identifiers to full words [15]. We plan to use these techniques in future work.

Acknowledgment

This work is supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grants PROSEED and ICT12-059.

References

[1] http://ctuning.org/wiki/index.php/CTools:CBench.

[2] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "The astrée analyzer," in *Programming Languages and Systems*. Springer, 2005, pp. 21–30.

[3] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. von Rhein, "Domain types: Selecting abstractions based on variable usage," CoRR, Tech. Rep. abs/1305.6640, 2013.

[4] http://sv-comp.sosy-lab.org/2013/benchmarks.php.

[5] http://www.gnu.org/software/libc/manual/pdf/libc.pdf.

[6] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer-Verlag New York Incorporated, 1999.

[7] http://clang.llvm.org.

[8] J. Weston and C. Watkins, "Multi-class support vector machines," Department of Computer Science, Royal Holloway, University of London, Tech. Rep. CSD-TR-98-04, 1998.

[9] http://www.cs.waikato.ac.nz/ml/weka.

[10] J. Sajaniemi, "An empirical analysis of roles of variables in novice-level procedural programs," in *Proceedings of the IEEE Symposia on Human Centric Computing Languages and Environments*, 2002, pp. 37–39.

[11] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," *SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 57–72, 2001.

[12] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2008, pp. 159–169.

[13] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[14] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006.

[15] D. Lawrie, H. Feild, and D. Binkley, "Extracting meaning from abbreviated identifiers," in *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2007, pp. 213–222.