Software Reliability Estimation Based on Static Error Detection

Mikhail Glukhikh
St. Petersburg State Polytechnical University
St. Petersburg, Russia
glukhikh@kspt.ftk.spbstu.ru

Mikhail Moiseev
St. Petersburg State Polytechnical University
St. Petersburg, Russia
mikhail.moiseev@gmail.com

Anatoly Karpenko
St. Petersburg State Polytechnical University
St. Petersburg, Russia
karpenko@kspt.ftk.spbstu.ru

Harald Richter
Clausthal University of Technology
Clausthal, Germany
hri@tu-clausthal.de

Abstract in English — **The estimation of a programs' reliability is an essential part in the process of software development. Existing methods for the analysis of software reliability are based on run-time data, program metrics, and properties of development process or program architecture. The disadvantage of these methods is that they use indirect information about the errors, which are the main cause of program unreliability.**

In the paper we present a novel approach for software reliability estimation. This approach is based on error detection using static source code analysis. We extend static analysis with developed algorithms which calculate error probabilities and program reliability characteristics. The characteristics are the probability of successfull program termination, the probability of the program is operable after execution of n statements, and mean number of executed statements before failure. The suggested approach has been implemented in the AEGIS tool and tested in numerous real-world software projects.

Keywords-software reliability estimation; static program analysis; program error; automatic program error detection.

I. INTRODUCTION

In modern software, reliability is one of the most critical issues. A common definition of software reliability is its capability to execute all prescribed tasks correctly and in time. It is well-known that the main reason for insufficient software reliability is semantical errors a programmer makes.

When an error-prone source code is executed and a necessary and sufficient condition for the error is existent, then a fault arises. The fault may lead to a program failure. A program failure is a situation when the program does not terminate, terminates prematurely or gives wrong results.

In general, software errors can be divided into the two classes of functional and non-functional errors. Errors from the first class violate the functional specification of the software under test. The second class comprises programming language rules violations, wrong library usage, and others. The most common non-functional errors in C/C++ programs are incorrect pointer dereferences, memory and resource leaks, use of non-initialized objects, buffer overflow [1].

In recent years, static analysis methods become more and more popular for automatic error detection in software. Two basic notions exist for error detection methods which are called soundness and precision. *Soundness* is the part of true errors, which are detected by a method, among all program errors. *Precision* is the part of true errors, which are detected by a method, among all detected errors.

Static analysis methods are highly-automated, some of them also have high soundness, and low resource consumption. However, this comes along with the detection of false positive errors. Also, the detection of trivial positive errors is possible. Such errors occur either seldomly or have a neglectable negative influence on the reliability. False positives and trivial positives are usually the biggest fraction of all detected errors. This requires a manual checking of static analysis results which is a time-consuming process. Manual checking and correction of all detected errors is possible only for some critical parts of a program or for small programs.

For ensuring program reliability it is mandatory to correct all critical errors. Furthermore, it is wise not to waste too much time on false or trivial positive ones. We therefore introduce a novel method of ranking of automatically detected errors in source code analysis. Errors are ranked by their influence on the program's reliability. Errors with high influence should be manually checked and corrected first.

During a program modification and error correction, the reliability of the program can deteriorate because of new errors made by programmers. Thus program reliability should be monitored in all stages of development process. For this purpose, an estimation of the influence of residual errors on the reliability can be useful. We propose a novel method for estimation of certain program reliability characteristics which are: the probability $P(n)$ that a program is operable after execution of n statements, the mean number $\overline{n}$ of executed statements to failure, and the probability $P(\infty)$ of getting correct results after program termination. These characteristics show reliability of different program types.

The rest of the paper is organized as follows: in Section II related work is reviewed, Section III describes the main idea of our approach. In Section IV the developed analysis algorithms

978-1-4673-0844-1/11/$26.00 ©2011 IEEE

are given. In Section V some experimental results are discussed. The paper ends in Section VI with a conclusion and an outlook to future work.

II. Related Work

There are several approaches for software reliability estimation. The most common approaches use the following techniques:

1. Estimations based on information about program failures which is given by run-time analysis.
2. Reliability evaluations based on program complexity metrics.
3. Evaluation methods which are taking into account some properties of the software development process.
4. Architecture-based estimation approaches which are based on the program architecture and the reliability of program components.
5. Approaches which use information about statically detected errors in program source codes.

The first class of methods is based on information such as number of program failures per time unit or mean time between program failures which is obtained during on run-time. Widely used in this class are so-called Software Reliability Growth Models (SRGMs) [4]. These models are extrapolations which take into account software specialties. SRGMs implement some principles which correlate the number of residual errors with the probability of their occurrence while testing. The first SRGMs were proposed by Jelinsky and Moranda [5]. After that, several dozens of SGRMs were suggested such as the Nonhomogeneous Poisson Process model [6], the Musa's Basic Execution Time model [7], the Weibull model, the Littlewood-Verrall Reliability Growth model [8]. In order to improve the accuracy of prediction, the Nonhomogeneous Poisson Process model was even extended by Neural Networks in [9]. SGRMs have the same disadvantages which are typical for run-time methods, such as low program coverage, unsoundness of given estimations, and impossibility of proper automation.

The class of methods based on program complexity metrics uses the assumption that there is a correlation between software complexity and its reliability. This assumption has some reasons, because high program complexity is a cause of developers' mistakes. Experiments e.g. in [10][11] show that correlation. The simplest of these approaches use Lines of Source Codes or Halstead metric [12] for reliability estimation. The transformation from code complexity to reliability is made by empirical coefficients. Other approaches use information about program structure, including the number of loops, the number of program variables, functions and modules, interconnection between modules and so on [13][14]. So these approaches are based on general program properties but not specific errors. However, program errors are usually localized in few parts of the source codes. Such estimation methods are therefore not sensitive to specific errors. For example, if a program were treated such that all errors would be corrected then its complexity would not change much, but its reliability would increase significantly. These approaches can thus deliver only results with low precision.

The approaches based on information about development process usually take into account factors like the duration of development, the number of developers and their qualification, the process organization methodology and tools for automation of development. For reliability estimation, these factors are normalized, weighted with empirical coefficients and added together. The coefficients usually are obtained from previously completed projects. Some of these approaches are presented in [15]. The applicability of these methods is limited because they do not consider a program's specialties, nor changing conditions such as new tools and technologies used in development process and neither human factor as well.

Architecture-based methods are suggested for estimation of reliability of software with reusable components [16]. Such components must have been developed and tested in previous projects or taken from component libraries. The methods treat a program as a set of components with interconnections between them. If the reliability of every component is known then the overall reliability can be estimated. This is similar to approaches which are used for complex-structured hardware systems. There are several such methods which use Finite State Machines [17], Dynamic Bayesian Networks [18], and Petri nets [19].

The last class of approaches works with statically detected errors. In [20] program reliability is estimated by retrieving statically detected errors and information about the probabilities of program branches execution. Branch prediction information is obtained by program testing. A Bayesian Network is used for an overall assessment.

The approach presented in this paper is also based on statically detected errors. However, in contrast to [20] we use an analysis of source code for error probability estimation that leads to higher code coverage and lets to automate reliability estimation.

III. Description of the Approach

The suggested approach for software reliability estimation is based on information about errors in source codes, which are detected by static analysis. The approach delivers:

- A ranking of errors with respect to the probability that the error will lead to a failure.
- An estimation of some characteristics of the reliability.

The approach can be used to make program debugging more effective. Manual checking of errors is made easier because of the ranking of the errors.

In the paper we discuss single-threaded C/C++ programs. The following types of errors are considered:

- Uninitialized variable use,
- Incorrect pointer dereference,
- Pointer out of bounds.

These error types were chosen because they often lead to program failure and are wide-spread in real C/C++ programs [1, 2].

A. Features of the Approach

The features of our approach are caused by the following properties of static analysis:

- Analysis of an abstract program model without compilation and running.
- Analysis of all possible program execution paths and possible input parameters.

As a consequence, our approach does not consider quantitative time, because no information about hardware and execution time is used. We use instead time the number of executed statements. This weakness can be overcome by use of time weights for statement types. The other trait is that program analysis gives results which are true for the program itself. That is good, so these results are applicable for all conditions. From the other viewpoint it does not consider normal program exploitation. This problem can be solved with annotation of program exploitation environment, which will be used by static analysis algorithms.

B. Error Ranking

The error ranking is based on error occurrence probability. To calculate it, we estimate the probability that a statement will be executed which has a potential error. We furthermore estimate the probability that a specific program state which leads to an error is achieved before the statement. Program state determination and probability calculation are performed by extended analysis algorithms.

C. Overall Reliability Estimation

For overall reliability estimation it is necessary to differentiate between two types of programs:

- *Computational programs* which produce a useful result and terminate. The probability $P(\infty)$ is used as an indicator of successful program execution.
- *Server programs* which continuously produce useful results. The used characteristic is $P(n)$, which is the probability that n statements are successfully executed. Also, the mean number $\overline{n}$ of executed statements before failure is estimated.

The calculations of these characteristics are based on program states probabilities in different program statements.

IV. Analysis Algorithms

The estimation of program reliability is performed by the following steps:

- Program model building,
- Program state determination,
- Error detection,
- Error ranking,
- Overall reliability estimation.

A. Program Model Building

All static analysis methods use an abstract model to represent source codes. There are several models known in literature [21]. In this paper, we use Control Flow Graph which is extended by additional edges. Furthermore, all assignments are transformed to three-operand form for analysis simplification. Control flow statements are represented by an If-statement node, where the flow is divided into two paths, and a special node called ϕ-function, where two or more flows are joined together [22].

B. Program States Determination

For program error detection, static analysis algorithms should extract information about the program state Q in different program points. Usually program states are determined before every program statement. A program state consists of data objects and its values. Data objects are variables (including arrays and structures) and dynamic memory chunks. A program state can be represented as a set of tuples $Q = \{(o_j, v_k)\}$, where o_j is a data object, and v_k is one of its possible values. v_k can assume one of three types:

- A numeric value which is either an interval $v_k = (i_{\min}, i_{\max})$, or an exact value $v_k = i$.
- A pointer value is represented by the object and an offset inside of that object $v_k = (o_m, offset_m)$.
- A resource descriptor value which contains information about a resource object.

During program execution, values of any data object are known exactly. We use analysis algorithms which join program states by means of ϕ-functions, in order to have multiple values for a data object as program state. In this case there are several tuples in the program state for such object.

State determination can be done by using different static analysis methods such as abstract interpretation, control flow analysis, type and effect systems [23]. For calculating probability of error occurrence and reliability estimation, we extend the tuple with a probability, forming the triple $Q = \{(o_j, v_k, p_{jk})\}$. The sum of all triples probabilities for one object must obey the following rule:

$$\forall o_j : \exists (o_j, v_k, p_{jk}) \in Q \Rightarrow \sum_{\forall (o_j, v_k, p_{jk}) \in Q} p_{jk} = P(Q), \quad (1)$$

where $P(Q)$ is the probability of program state Q. Value $P(Q)$ is the probability that Q is the state before s. This means that the statement s will be executed with state Q and probability $P(Q)$. We use context-sensitive and flow-sensitive algorithms, so s can be executed with different program states

in different function call stack and different set of loops' iterations.

Static analysis algorithms perform state transformations at every statement. Input state Q^{in} of statement s is transformed to output state Q^{out}. In these transformations data object values and triples' probabilities can be changed. Analysis algorithms must obey the following rule:

$$\sum_{\forall Q_j^{in} \in Input(s)} P\left(Q_j^{in}\right) = \sum_{Q_j^{out} \in Output(s)} P\left(Q_j^{out}\right), \tag{2}$$

where $Input(s)$ and $Output(s)$ are the sets of input and output states for s. The number of states in $Input(s)$ and $Output(s)$ corresponds to the number of inputs and outputs of s. If-statement has one input and two outputs; ϕ-function has two or more inputs and one output; all other statements are *sequential statements*, they have one input and one output.

In the following the developed analysis rules are discussed with an emphasis on the calculation of probabilities.

1) Sequential Statement Analysis

Thus a sequential statement has one input and one output. It holds according to (2): $P\left(Q^{in}\right) = P\left(Q^{out}\right)$. Statement analysis can remove existing triples from the program state, create new triples, and modify existing triples.

For example consider program state transformation for statement "`a = b + c`", where "`a`", "`b`" and "`c`" are numeric variables. For calculating possible values of "`a`", we must take into account all pairs of "`b`" and "`c`" values. So, state transformation rule has the following form:

$$\begin{aligned} Q^{out} = Q^{in} / \bigcup_{\forall (a, v_{k_1}, p_{k_1}) \in Q^{in}} \left(a, v_{k_1}, p_{k_1}\right) \cup \\ \bigcup_{\forall (b, v_{k_2}, p_{k_2}), (c, v_{k_3}, p_{k_3}) \in Q^{in}} \left(a, v_{k_2} + v_{k_3}, p_{k_2} \cdot p_{k_3}\right). \end{aligned} \tag{3}$$

Let $P\left(Q^{in}\right) = 1, b = (1...2), c = (3...6)$ with even probability distribution. An example of rule (3) working is shown in Fig. 1.

There are two possible variants for analysis of the example. The first one is considering variables' values as integer numbers. It is precise but memory-consuming method. The second variant is considering values as intervals, for example $(b, (1,2), 1.0)$, $(c, (3,6), 1.0)$, then result will be $(a, (4,8), 1.0)$. In this case we assume that probability for an interval is evenly distributed. Use of second variant leads to some precision loss, but gives us the opportunity to reduce resource consumption. In practice we use the second variant. Results of both variants are presented in Fig. 1.

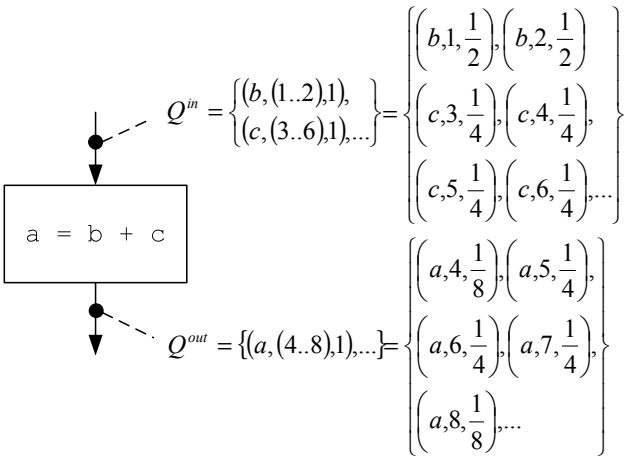

Figure 1. Applying of the rule for "a = b + c".

2) If-statement Analysis

Analysis of If-statement performs division of input program state Q^{in} into two subsets with possible crossover. First subset Q^{true} is program state in true branch of If-statement, second subset Q^{false} is program state in false branch. For If-statement we have that $Q^{in} = Q^{true} \bigcup Q^{false}$ and according to (2) $P\left(Q^{in}\right) = P\left(Q^{true}\right) + P\left(Q^{false}\right)$.

For calculation of $P\left(Q^{true}\right)$ and $P\left(Q^{false}\right)$ the condition of If-statement should be interpreted. For this purpose it is necessary to determine data objects which are related to the condition (directly or indirectly). Let us designate the set of objects related with condition of the If-statement as O_{rel}.

In the simplest case when If-statement condition contains only one data object o_j any triple with o_j falls in one output state only, i.e. in Q^{true} or in Q^{false}. Probabilities of triples of o_j are not changed (they are the same as in Q^{in}). In this case:

$$\begin{aligned} P\left(Q^{true}\right) = \sum_{\forall (o_j, v_k, p_{jk}) \in Q^{true},\, o_j \in O_{rel}} p_{jk}, \\ P\left(Q^{false}\right) = \sum_{\forall (o_j, v_k, p_{jk}) \in Q^{false},\, o_j \in O_{rel}} p_{jk}. \end{aligned} \tag{4}$$

In general case, we should consider a set of triples' combinations. Each combination includes exactly one triple for each object from O_{rel}. For each combination it is possible to determine is the If-statement condition true or false. All combinations are divided into two disjoint subsets C^{true} and C^{false}. Probabilities of Q^{true} and Q^{false} are calculated as:

$$\begin{aligned} P\left(Q^{true}\right) = \sum_{c \in C^{true}} \prod_{(o_j, v_k, p_{jk}) \in c} p_{jk}, \\ P\left(Q^{false}\right) = \sum_{c \in C^{false}} \prod_{(o_j, v_k, p_{jk}) \in c} p_{jk}. \end{aligned} \tag{5}$$

Triples for objects from O_{rel} fall in output state (or states) using the following rule:

$$\begin{aligned}
&\forall o_j \in O_{rel} : (o_j, v_k, p_{jk}) \in Q^{in} \Rightarrow \\
&(o_j, v_k, P_{true}(o_j, v_k, p_{jk})) \in Q^{true}, \\
&(o_j, v_k, P_{false}(o_j, v_k, p_{jk})) \in Q^{false}, \\
&P_{true}(o_j, v_k, p_{jk}) = p_{jk} \cdot \sum_{\substack{c \in C^{true}, \\ (o_j, v_k, p_{jk}) \in c}} \prod_{(o_l, v_l, p_{il}) \in c} p_{il}, \\
&P_{false}(o_j, v_k, p_{jk}) = p_{jk} \cdot \sum_{\substack{c \in C^{false}, \\ (o_j, v_k, p_{jk}) \in c}} \prod_{(o_l, v_l, p_{il}) \in c} p_{il}.
\end{aligned} \tag{6}$$

If a triple (o_j, v_k, p_{jk}) probability $P_{true}(o_j, v_k, p_{jk})$ or $P_{false}(o_j, v_k, p_{jk})$ is zero, then this triple does not fall in true or false branch correspondently.

Triples for other objects (objects not from O_{rel}) fall in both output states with probability normalization:

$$\begin{aligned}
&\forall o_j \notin O_{rel} : (o_j, v_k, p_{jk}) \in Q^{in} \Rightarrow \\
&\left(o_j, v_k, \frac{p_{jk} \cdot P(Q^{true})}{P(Q^{true}) + P(Q^{false})}\right) \in Q^{true}, \\
&\left(o_j, v_k, \frac{p_{jk} \cdot P(Q^{false})}{P(Q^{true}) + P(Q^{false})}\right) \in Q^{false}.
\end{aligned} \tag{7}$$

Example of applying the rules is shown in Fig. 2. In this example we should consider all combinations of "a" and "b" values. There are 172 combinations where "a < b", and 28 combinations where "a ≥ b", so $P(Q^{true}) = 0.86$ and $P(Q^{false}) = 0.14$.

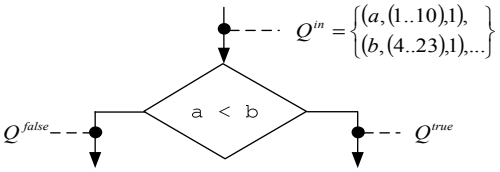

Figure 2. Applying of If-statement rules.

Applying the rule (6) we should calculate individual coefficient $P_{true}(o_j, v_k, p_{jk})$ for each triple with "a" or "b" according to number of true/false combination with it. In the example, Q^{true} contains for variable "a" the following triples: $(a,1,0.1)$, $(a,2,0.1)$, $(a,3,0.1)$, $(a,4,0.095)$, $(a,5,0.09)$, …, $(a,10,0.065)$; Q^{false} – $(a,4,0.05)$, $(a,5,0.1)$, …, $(a,10,0.35)$.

At last, applying the rule (7) we should use coefficients $P(Q^{true})$ and $P(Q^{false})$ for every other variable in state (not "a" and not "b").

3) Φ-function Analysis

In ϕ-functions, we join program states from all inputs. If there are identical triples (triples with the same object and value) in two or more input states, their probabilities are added together. Rule for two input states is the following:

$$\begin{aligned}
&\forall o_j, v_k : (o_j, v_k, p_{jk}) \in Q_1^{in}, (o_j, v_k, r_{jk}) \in Q_2^{in} \Rightarrow \\
&(o_j, v_k, p_{jk} + r_{jk}) \in Q^{out}
\end{aligned} \tag{8}$$

Also, according to (2), $P(Q^{out}) = P(Q_1^{in}) + P(Q_2^{in})$.

4) Loop Analysis

Our methods perform statement analysis according to statement sequence in program model. A loop is represented in program model as input ϕ-function and output If-statement (for multi-input loop several ϕ-functions are used, for multi-output loop several If-statements are used). Input ϕ-function has an input from outside of the loop and an input from loop body. Output If-statement has two successor nodes: one is in the loop body and other is outside of the loop. Analysis of loop ϕ-function and If-statements is performed in the usual way.

5) Interprocedural Analysis

An important issue in static analysis methods is procedure (or function) call analysis. There are two basic approaches:

- The context-sensitive approach [24] repeats analysis of procedure source code each time when this procedure is called, with substitution of formal parameters by the concrete values.
- The context-insensitive approach [25] performs only one analysis of procedure source code, with undefined values of formal parameters.

The context-sensitive approach is resource-consuming but has better soundness and precision. It is used in our approach and includes the following steps:

- Evaluation of formal parameters,
- Analysis of the procedure body,
- Evaluation of the procedure result.

A special case that has to be considered separately is a function call via a pointer. In this case, our analysis algorithm determines and evaluates all possible functions which are pointed in the call statement, and analyze all of these functions calls. Then, the program states before returning results are calculated and joined by means of a ϕ-function.

C. Error Detection

Error detection is performed for each statement. It uses program state in the input of the statement – Q^{in}. Let us discuss about error detection rules for chosen error types.

1) Uninitialized Variable Use

If a statement reads an object value, it should be checked whether the object is initialized or not. For initialization check,

special value $v_{noninit}$ is used. So a triple with uninitialized object value looks like $(o_j, v_{noninit}, p_k)$, where p_k is the probability of uninitialized condition. If object o_i is read and a triple $(o_j, v_{noninit}, p_k)$ exists in Q^{in}, then "uninitialized variable use" error occurs with probability p_k.

2) Incorrect Pointer Dereference

If a statement is processed that dereferences some pointer value, it is checked whether this pointer value is correct or not. Pointers can have at least three types of incorrect values:

- Null value,
- Non-initialized value,
- Corrupted value (points to object that was already released, was cast from integer value, etc.).

For incorrect pointer values special values v_{null}, $v_{noninit}$ and $v_{invalid}$ are used. If a pointer was dereferenced and triples with its incorrect value from Q^{in} have overall probability p_k, then "incorrect pointer dereference" error occurs with probability p_k.

3) Pointer Out of Bounds

Before a pointer is dereferenced or an array element is accessed it is checked whether the pointer or element is in the object bounds (assuming that pointer value is correct in terms of previous paragraph). Let Q^{in} contains triple $(o_i, (o_j, offset_j), p_k)$, where o_i is the pointer, o_j an object with size $sizeof(o_j)$ and p_k is the probability of this triple. The pointer is inside object bounds if $0 \le offset_j < sizeof(o_j)$. In the opposite case, "pointer out of bounds" error occurs with probability p_k.

4) Error Inhibition

In the beginning of a program start, there is one state with probability 1 because the first statement is always executed. During program analysis, probabilities of program states in program execution paths are decreased after detection of errors. If an error is detected in a statement with probability p_k, then a fault and after that a program failure can happen with probability p_k. For this reason, we remove triples which cause the fault. It leads to decreasing program state probability: $P(Q^{out}) = P(Q^{in}) - p_k$. Also, we should normalize triple probabilities for other data objects to satisfy (1).

D. Error Ranking

Error ranking is based on the probability of errors occurrence. After probability derivation, all errors are sorted according to it. This eases program development process because the error with the highest probability is the most dangerous one. It should be checked and corrected first of all, in the assumption that associated fault types will always lead to program failure. In practice, this is precisely true only for incorrect pointer dereferences. Uninitialized variables and pointers out of bounds may lead later to a program failure with high probability, but successful program execution is also possible.

Finally, it can happen that several errors are detected in one statement. These errors can be one error which is detected in different loop iterations or in different calls of the same function. Such errors have the same type and the same involved objects. These errors are joined and their probabilities are added together.

E. Overall Reliability Estimation

Probability of the program is operable after execution of n statements ($P(n)$) and the probability of getting correct results ($P(\infty)$) are calculated with use of program states' probabilities. For $P(n)$ calculation we summarize probabilities of program states after execution of n statements. For this purpose we use statement counter in a program state. Statement counter is incremented after analysis of each statement.

Parameter $\overline{n}$ is obtained using the following formula:

$$\overline{n} = \sum_{n=0}^{n_{\max}} (P(n) - P(n+1)) \cdot n, \qquad (9)$$

where $n_{\max}$ is maximal number of statements. Value $n_{\max}$ is determined as maximal n for which $P(n) > 0$. For some programs $P(n) > 0$ for any n or $n_{\max}$ can be too big, then it is limited on some appropriate value. In the last case $\overline{n}$ indicate mean number of statements which are executed till $n_{\max}$-th statement.

Calculation of $P(\infty)$ is performed by summarize probabilities of program states in the program termination statements. We suppose that program execution till termination point without faults gets correct results. Determining value $P(\infty)$ makes sense for computational programs only.

V. Experimental Results

The developed approach is implemented in AEGIS static analyzer [3]. AEGIS performs automatic analysis of C/C++ program source code and detects the most spread types of program errors. AEGIS uses sound and high-precision algorithms of points-to, interval and resource analysis, that allows getting good results.

Program reliability characteristics are estimated during static analysis algorithms execution. Triple probabilities and program state probabilities are calculated by extended rules which are used for analysis of program statements. When detection rule finds a new error, its probability is calculated. At the same time the state probability is normalized. For $P(\infty)$ calculation the probabilities of program state are stored as soon as reaching the program exit statements. For $P(n)$ calculation the probabilities of program state are stored after n_i analyzed

statements (for instance n_i can be 0, 1000, 2000, …). Value of $\bar{n}$ is calculated in the end of analysis, it is based on $P(n)$ results.

For the evaluation of the described approach, two sets of test programs were analysed by AEGIS:

- 14 students' projects with code lengths of 0.5-5KLOC. These projects contained errors, only some of them were known before analysis.
- 12 real-world projects for existing embedded system with code lengths of 30-50KLOC. These projects contained errors which were detected by AEGIS.

A. Experimental Results for Students' Projects

The purpose for analyzing students' projects was to test our approach. We calculated therefore manually the probabilities of detected errors and compared them to results obtained automatically by AEGIS. In the following example, a C program is given that checks Cyrillic words rewritable into latin characters. Parts of the program including errors are presented below (key lines are bold).

```
void prov(int *t, int i, char st [25])
{
  char lat[11] = {'E','T','O','P','A','H','K','X',…};
  int j,k,l; // should be k = 0
  for (j = *t; j < i; j++) {
    k = 0;
    for (l = 0; l < 11; l++) {
       if (st[j] == lat[l]) k = 1;
       if (k == 0) break;
    }
    if (k == 1) { … } // k may be uninitialized
  }
}
int main (void)
{
  char st[25] = "";
  int i,t;
  FILE* f = fopen("in.txt","r");
  while (!(feof(f))) {
    fscanf(f,"%s",st); // possible overflow
    i = t = 0;
    prov(&t, strlen(st), st);
  }
}
```

Errors are the possible overflow in "`fscanf`" call and the possible uninitialized variable "k" in If-condition. AEGIS has detected both errors but only the second error is of one of the considered error types. AEGIS results are presented below:

```
   Uninitialized variable error in main.c line 75
with probability 0.250
   n      P(n)
   -----------
   0       1.0
   1000    0.833
   2000    0.777
   3000    0.759
   4000    0.753
   ...
   P in program finish points is 0.750
```

The finish point in the AEGIS output means the last executed statement. Furthermore, the error probability can be estimated manually in the following way. The loop condition in "`while(!(feof(f)))`" cannot be interpreted, so the probability of its truth is 0.5. The variable "`k`" can be 0 or 1 after evaluation, and it can be uninitialized if the length of string "`st`" is zero. So a fault occurs in one of three cases (k = 0, 1, noninit). Finally, the following three possibilities exist on each loop iteration:

- If the while condition is false with 0.5 probability then the loop terminates.
- If the while condition is true with 0.5 probability, then a fault occurs in one of three cases with 0.5 * 0.333 = 0.167 probability.
- Otherwise, one loop iteration terminates successfully with 1.0 – 0.5 – 0.167 = 0.333 probability and then next iteration starts.

So $P(\infty) = \frac{1}{2}\left(1 + \frac{1}{3} + \frac{1}{9} + ...\right) = \frac{1}{2} \cdot \frac{3}{2} = 0.75$ is the same that the AEGIS results. After the error was fixed (by changing "`int k`" to "`int k = 0`"), AEGIS results were changed to $P(\infty) = 1.0$ with no detected errors.

B. Experimental Results for Embedded Software Projects

AEGIS was tested on 14 real-world projects where it found more than 500 errors, about 2/3 of them are of considered error types. The amount of errors in each project is shown in Fig. 3. The average error density is 0.8 errors/1KLOC which correlates with our earlier experiments conducted in [26] and other results [1].

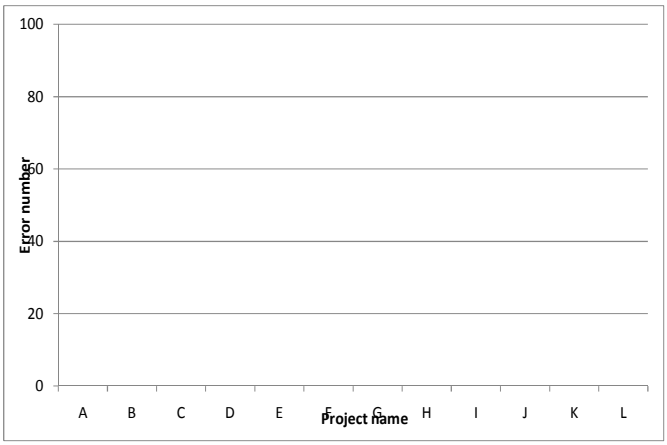

Figure 3. Amount of errors in real-world projects A-L.

The results of error ranking are presented in Fig. 4. A high amount of errors may happen only with low probability. About 10% of the errors have a probability higher than 0.01 and should be scrutinized first by the programmer. This shows that error ranking is really useful to reduce the effort for manual checking and correction.

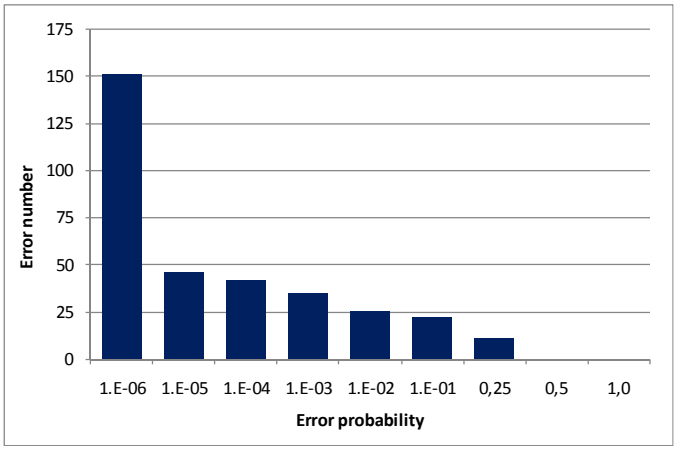

Figure 4. Distribution of error number of reliability

The reliability estimation was tested with project D. For this project we have got $P(n)$ function of several n. After that we have corrected *two most probable errors* (of 90 errors in project D) and have repeated the analysis. Curves of $P(n)$ for two versions of project D are shown in Fig. 5.

Probability of corrected version of D is increased for five times in average starting from $n = 35$Mstatements. The value of $\overline{n}$ for original version is 36.5Mstatements, for corrected version is 43.9Mstatements. The following correction of a few numbers of next errors in the ranked list allows achieving appropriate reliability of the program. In case when user accidentally chooses errors for correction, achieving the same reliability demands much more time.

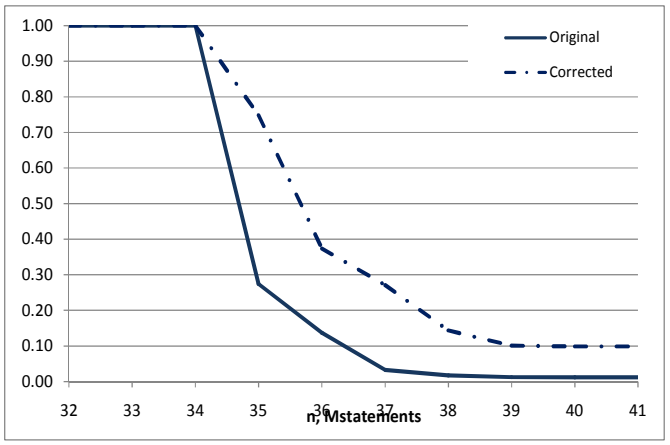

Figure 5. Comparison of two versions of project D

Finally, we have tested the overall performance and computational scalability of AEGIS by measuring the analysis time for all mentioned projects. The analysis time is in the interval of 3 to 30 minutes for these projects (projects sizes 30KLOC – 50 KLOC). The largest project ever analyzed by AEGIS has 470KLOC and need less than 10 hours to analyze. All time measures have been made on a workstation with one single-core processor.

VI. CONCLUSION

In this paper, a novel approach for software reliability estimation is presented. The approach is based on error detection in program source codes using static analysis methods.

Detected errors are ranked by the probability of occurrence which allows to check and correct errors with high probability first. For observe and control the program reliability growth during error correction our approach estimates some overall program reliability characteristics. Reliability is expressed by the probability that the program is operable after execution of n statements or by the probability of successful program termination. Also the mean number of executed statements before failure is evaluated. These characteristics directly exhibit program reliability and can be useful for program developers. The described approach was implemented in AEGIS tool.

In this paper we do not discuss false positives problem. False positives in static analysis results may make inaccurate the given reliability estimations. For cope with this problem we should estimate probability of error truth. The other important thing is extending our approach to parallel programs. There are static analysis algorithms which allow to detect errors in parallel programs [27]. Using of these analysis algorithms gives us the necessary information for estimation of parallel program reliability. We are planning to implement these tasks in the future developments.

REFERENCES

[1] Open Source Reports – 2008 and 2009 – http://www.scan.coverity.com/report/.

[2] CERT C Secure Coding Standard – https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard

[3] AEGIS Error Detection Tool – http://www.digiteklabs.ru/aegis/

[4] Lyu M.R, "Software Reliability Engineering: A Roadmap", Proceedings of Future of Software Engineering, IEEE Computer Society Press, 2007, pp. 153-170.

[5] Z. Jelinski and P.B. Moranda, "Software Reliability Research", in Proceedings of the Statistical Methods for the Evaluation of Computer System Performance, Academic Press, 1972, pp. 465-484.

[6] A.L. Goel and K. Okumoto, "Time-Dependent Error-Detection Rate Model for Software and Other Performance Measures", IEEE Transactions on Reliability, vol.R-28, no.S.August 1979, pp. 206-211.

[7] J.D. Musa, A. Lannino, and K. Okumoto, "Software Reliability, Measurement, Prediction, Application", McGraw-Hill, NewYork, 1987.

[8] B. Littlewood, "The Littlewood-Verrall Model for Software Reliability Compared with Some Rivals", Journal of Systems and Software, vol. 1, no. 3, 1980, pp.251-258.

[9] J. Zheng, "Predicting Software Reliability with Neural Network Ensembles", Expert Systems with Applications, Vol. 36, Pergamon Press, 2009, pp. 2126-2122.

[10] H. Zhang, X. Zhang, M. Gu, "Prediction defective software components from source code complexity measures", Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing, IEEE Computer Society, 2007, pp. 93-96.

[11] J.E. Simensen, C.Gerst, B.A. Gran, J. Martz, and H. Miedl, "Establishing the Correlation between Complexity and a Reliability Metric for Software Digital I&C-Systems", Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security, Springer-Verilag, 2009, pp. 55-66.

[12] S.H. Kan, "Metrics and Models in Software Quality Engineering", Addison-Wesley, 2002, 512 p., ISBN 0-201-72915-6.

[13] D.S. Kushwaha and A.K. Misra, "Cognitive Complexity Metrics and its Impact on Software Reliability Based on Cognitive Software Development Model", ACM SIGSOFT Software Engineering Notes, ACM NewYork, 2006, pp.1-6.

[14] S. Misra and I.Akman, "A Unique Complexity Metric", Proceedings of the international conference on Computational Science and Its Applications, Springer-Verilag, 2008, pp. 641-651.

[15] G.N. Tcherkesov, "Hardware-Software Systems Reliability", St. Petersburg, Piter, 2005, 480 p., ISBN: 5-469-00102-4.

[16] I. Krka, L. Cheung, G. Edwards, L. Golubchik, and N. Medvedovic, "Architecture-Based Software Reliability Estimation: Problem Space, Challenges, and Strategies", Proceedings of the DSN Workshop on Architecting Dependable Systems, 2008.

[17] R.H. Reussner, H.W. Schmidt and I.H. Poernomo, "Reliability prediction for component-based software architectures", Journal of Systems and Software, Elsevier Science Inc., 2003, pp.241-252.

[18] R. Roshandel, N. Medvidovic, and L. Golubchik, "A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level", Proceedings of the Quality of software architectures 3rd international conference on Software architectures, components, and applications, Springer-Verlag Berlin, 2007, pp.108-126.

[19] A. Dimov and S. Punnekkat, "Fuzzy Reliability Model for Component-Based Software Systems", Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE Computer Society, 2010, pp. 39-46.

[20] W. Schilling and M. Alam, "Modeling the Reliability of Existing Software using Static Analysis", Proceedings of lectro/information Technology IEEE International Conference, IEEE Computer Society, 2006, pp.366-371.

[21] V. Itsykson, M. Glukhikh, A. Zozulya, and A. Vlasovskikh, "Research of C/C++ Source Code Model Extraction Tools", Scientific Journal of Saint Petersburg State Polytechnical University, Computer science section, vol. 1 (72), 2009, pp. 122-130.

[22] R. Cytron, J. Ferrante, B. Rosen, and M. Wegman, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", ACM Transactions on Programming Languages and Systems, ACM, New York, 1991, pp. 451-490.

[23] F. Nielson, N. Nielson, and C. Hankin, "Principles of Program Analysis", Corr. 2nd printing, Springer-Verlag Berlin, 2005, XXI, 452 p., ISBN 978-3-540-65410-0

[24] O. Lhotak and L. Hendren, "Context-sensitive Points-to Analysis: Is it worth it", Proceedings of International Conference on Compiler Construction, Springer-Verlag Berlin, 2006, pp. 47-64.

[25] J. Midtgaard, "Control-Flow Analysis of Functional Programs", BRICS Report Series, RS-07-18, December 2007, 44 p.

[26] M. Glukhikh, V. Itsykson, and V. Tsesko, "Using Dependencies to Improve Precision of Program Static Analysis", 6th International Computer Science Symposium in Russia, The Second Workshop on Program Semantics, Specification and Verification: Theory and Applications, Yaroslavl, 2011, pp. 51-58.

[27] M. Moiseev, "Defect detection for multithreaded programs with semaphore-based synchronization", Proceedings of the 6th CEE Software Engineering Conference in Russia, IEEE Computer Society, 2010, pp. 64-70.