ISBN 978-981-14-4787-7
Proceedings of 2020 the 10th International Workshop on Computer Science and Engineering (WCSE 2020)
Shanghai, 19-21 June, 2020, pp. 474-484
doi: 10.18178/wcse.2020.06.069

Program Understanding and Requirement Validation Based on Accurate Value Flow Graph

Li Wu [1], Zhou Min [1 +] and Huang Hao [2]

[1] School of Software, Tsinghua University, Beijing 10084, China
[2] Department of software, AECC Aero Engine Control System Institue, WuXi Jiangsu 214063, China

Abstract. In software development, testing, verification and maintenance, it is a significant challenge to guarantee the consistency between software system and requirement. Software logic is usually implemented in the form of source code while requirements are usually described in the form of (structured) text. There are significant differences in form and connoation between the two, which brings many difficulties to software development. To solve this problem, this paper proposes a program analysis and requirement extraction algorithm based on accurate value flow graph. The algorithm uses static analysis to analyze program semantics and construct accurate value flow graphs. On this basis, the semantic information is extracted and summary is generated, which can be further used for requirements confirmation and defect analysis. Experiments show that the algorithm can analyze the control and data dependence of software code, and can accurately generate variable association information under different paths. The function summary can help developers to confirm whether the program semantics are consistent with the software requirements, help to find subtle semantic differences (including differences in conditions, dependencies, etc.), and effectively improve the efficiency of program understanding (up to 60% of time can be saved). This tool can be further used in the fields of defect analysis, influence domain analysis, etc., and has good application value.

Keywords: software verification and validation (V&V), program comprehension, value flow graph (VFG), function summary, static analysis.

1. Introduction

In software development activities, software verification and validation[1][2](V&V) is a very important step. The purpose of verification is to evaluate whether the software is implemented as definition while validation is to evaluate whether the software meets the expected requirements. In practice, software validation usually faces great difficulty[3]: to determine whether the implementation matches the requirements. However, the description of the two is quite different, the former is code with complex logical structure, while the latter is abstract text description.

The common method for the problem is to test whether the system has fulfilled the requirements one by one according to the requirements document[4]. However, this method is incomplete: in the case of program with complicated branch conditions, people often cannot enumerate all possible input and execution paths of the program, resulting in incomplete software validation. Another common approach is to try to understand the code during validation and then evaluate whether the code meets the requirements. There are still two problems: (1) the precise understanding of the code semantics is laborious, and need more human intervention[5][6][7], which is difficult to automate; (2) the description of software requirements is usually not formalized or even in most cases not specific enough.

Facing the above problems, this paper proposes an algorithm for program understanding and function summary generation based on accurate value flow graph (VFG). This method is intended to solve the following problems: (1) automatically extract the system semantics and then generate summary for comparison with requirements; (2) further help users refine their requirements according to the summary.

[+] Corresponding author. Tel.: +8617610069208;
E-mail address: liwu.thu@gmail.com.

The core of this method is to extract program semantics from the code quickly and automatically, establish the relationship between program implementation and requirements, and help developers understand the program for requirements validation. Previous research work on program understanding[8][9] is mainly based on program slicing, program labeling and execution visualization technology.

Program slicing[10] can reduce the amount of code reading by eliminating code statements that are not related to specified variables while program labeling[11] can provide more auxiliary information to the source code. Users still need to understand the logic of the code through the two method, which cannot significantly improve the reading efficiency. The execution visualization tool uses dynamic analysis to obtain and visualize the execution information, but the visualization effect will be poor when the program logic is complicated, and it needs to configure the runtime environment , which is difficult to get started.

The paper [12][13] pointed out that about 25% of the code maintenance workflow is problem-finding, modification and revalidation. At the same time, many developers understand the software by assuming and verifying the program behavior[14]. Therefore, create an algorithm (tool) to help developers extract program semantics will greatly reduce the relevant working time.

In this paper, we propose an automatic program static analysis algorithm based on VFG. The algorithm obtains the memory model by using pointer analysis then construct the VFG with it, and further analyzes the program semantics on the VFG. The method is a pure static analysis method, which does not depend on the specific execution path of the program and has a good completeness. The analysis results can be used by developers for program understanding and requirement validation, and also for automated defect analysis.

The main contributions of this paper are as follows:

1) Designed and implemented an accurate value flow graph analysis and construction method based on memory model, which can deal with the semantic relations that traditional data flow analysis cannot deal with, such as dynamic memory space, pointer alias, etc., and can distinguish the dependency conditions such as data flow dependency and control flow dependency;

2) The semantics of the analysis program can be constructed based on VFG, and this process can be automated and further used for requirements validation and defect analysis to improve program understanding efficiency and code quality;

3) Implemented a set of tools and carried out related experiments. The experimental results show that the tool can automatically generate readable program summaries and save up to 60% of the requirement validation time and eliminate the hidden defects in the code.

The structure of the paper is as follows: firstly, the related work of the paper is introduced in Chapter 1, and then the method is introduced through a specific case in Chapter 2. Secondly, Chapter 3 specifically introduces the implementation of the algorithm in detail, and Chapter 4 demostrates the effectiveness and practicability of the algorithm by combining experiments from both internal and external. Finally, Chapter 5 analyzes the reliability and follow-up work of the algorithm.

2. Related Works

At present, there are many researches on program understanding. According to the technical categories, they can be divided into program slicing, program labeling and execution visualization technology.

Program slicing was first proposed by Weiser[15] in 1979. It is an executable backward static analysis, which means the result contains all the program statements relied on by slicing criteria and can still be compiled and executed. However the results may not be significantly reduced sometimes due to the criteria. Subsequently, KoreL and Laski[16] put forward the dynamic slicing method which can obtain the runtime information (especially pointers and arrays) of the program. Because the generation of dynamic slicing depends on the execution of the program, the slicing results can't contain the possible related program statements. The advantage is that the slicing results are more simple and accurate but the disadvantage is that the results are only for specific input and are not compatible. Later, Gupta[17] and others proposed the hybrid slicing method, which uses both dynamic analysis and static analysis. Although slicing can reduce the amount of code reading by developers, it cannot provide abstract information on the code for programmers. Therefore, users still need to read the code to obtain knowledge.

The purpose of program labeling is to provide more information for programmers to understand the code. At present, there are many related papers and tools: SE-Editor[18] is an Eclipse plug-in, which displays code related charts, videos, etc. in the code by using hyperlink annotation. Tools such as Stacksplor[19], RegViz[20] and DepDigger[21] provide visual enhancements for source code, such as providing graphical

call information for functions, grouping and labeling group numbers for regular expressions, and changing background color according to the size of variables. There are also some tools that provide runtime information, such as In-situ[22] creates annotations for functions that include runtime performance information. Similar to the work in this paper is [23], which can provide summary information for functions. However, this information is in the form of keywords. Users cannot directly understand the behavior of functions through reading the summary.

In essence, the above tools provide auxiliary information for programmers to help understand, but in practice, people still need to read the source code in the application process, and the understanding speed has not been improved qualitatively.

Execution visualization tools such as ExtraVis[24] and ExplorViz[25] provide another way to assist understanding. On the basis of dynamic analysis, such tools try to visualize the execution path of the program based on multiple executions. The problem is that their compatibility will inevitably weaken because their technical basis is dynamic analysis, and they cannot fully represent the execution path of the program under different input conditions. If more input samples are added, the view will become more complex and difficult to understand, which violates the original intention of simplifying understanding. In addition, it has a certain learning cost, which is still not suitable for rapid understanding of code.

3. Research Motivation

Program understanding is often the most time-consuming work in software development, software verification and maintenance. This paper selects some software requirements and code implementation of the control logic of a motor vehicle transmission as an analysis case to introduce the algorithm.

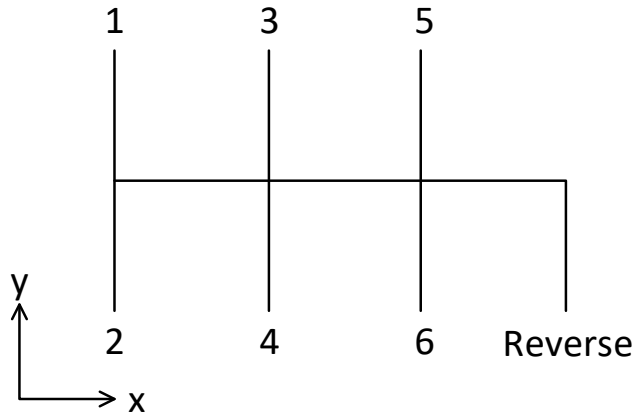


Fig. 1: Gear mapping of transmission gear lever

The control logic is applied to the six speed gearshift lever and support down-reversing, and the gear position map is shown in Figure 1. It is required to complete the control logic shown in Figure 1 by implementing the relevant functions, and the code shall comply with the following requirements:

- The function should take four parameters: x, y, diverse and oldGear;
- The set of returned gear values is {0, 1, 2, 3, 4, 5, 6, -1, -2}. 1-6 is forward and -2 is reverse gear;
- X is the travel of the transmission gearshift lever in the horizontal direction. When the value is in the range [0, 380), it means gear 1 or 2; when the value is in the range [380, 690], it means gear 3 or 4; when the value is greater than 690, it means gear 5, 6 or - 2 (reverse gear);
- Y is the travel of the transmission gearshift lever in the vertical direction. When the value is in the range [0, 160), the possible value of the gear is 2, 4, 6 and -2; when the value is in the range [160, 780], the possible value of the gear is 0 or -1 (it is required to press down the lever); when the value is greater than 780, the possible value of the gear is 1, 3 and 5;
- Diverse is the reverse signal. When pressing down the lever, the diverse value is 1, otherwise it is 0;
- The value of oldGear is the gear value before the transmission gear lever operation;

```
1.  int calculateGear(uint x, uint y, uint diverse, int oldGear)
2.  {
3.    int gear = 0;
4.    if (y < 160) {
5.      if (x < 380)
6.        gear = 2;
7.      else if (x > 690)
8.        gear = 6;
9.      else
10.       gear = 4;
11.   }
12.   else if (y > 780) {
13.     if (x < 380)
14.       gear = 1;
15.     else if (x > 690)
16.       gear = 5;
17.     else
18.       gear = 3;
19.   }
20.   if (diverse) {
21.     if ((gear == 6) && ((oldGear == -1 || oldGear == -2)))
22.       gear = -2;
23.     else if (gear == 0);
24.       gear = -1;
25.   }
26.   return gear;
27. }
```

Fig. 2: A function implementation of gear control

Figure 2 shows a function implementation corresponding to the above requirements. It is difficult for developers to obtain the relationship between input and output intuitively. For example, in the requirements document, the condition of gear value 6 is ($x > 690 \wedge y < 160 \wedge (diverse = 0 \vee (oldGear \neq -1 \wedge oldGear \neq -2)$). Only through expression analysis can they automatically learn from the code. Imagine that if such a behavior is completed manually, it will not only take time and effort, but also may result in condition omission and boundary condition error. More importantly, because of the instability of manual operations, it is difficult to grasp the small differences between requirements and code implementation. For example, there is a very serious semantic problem in the above code, which leads to the code execution is not consistent with the expectation when diverse $\neq 0$. Even with complete requirements documents and source code, it is very difficult to confirm the consistency without the support of tools.

The tool in this article can help solve the these issues. The expected return value and control logic are shown in columns 1 and 3 of Table 1, while the behavior of the actual code is shown in columns 1 and 2 of Table 1 (for convenience, the parameters diverse and oldGear are abbreviated to dv and og respectively). It is easy to find that the program cannot return gear = -2 and the gear is independent of the parameter oldGear, which is not as expected. Developers can guarantee the difference between the program behavior and requirements by comparing columns 2 and 3 to help find code problems. The root of this problem is that there is a semicolon at the end of the 23rd line of the program, which causes the control flow error.

Table. 1: Function summary generated from the code in Fig.2

Return Value	Actual Control Logic	Expected Control Logic
-2	-	$(x > 690) \wedge (y < 160) \wedge (og = -2 \vee og = -1) \wedge (dv \neq 0)$
-1	$dv \neq 0$	$(160 \leq y \leq 780) \wedge (dv \neq 0)$
0	$(160 \leq y \leq 780) \wedge (dv = 0)$	$(160 \leq y \leq 780) \wedge (dv = 0)$
1	$(x < 380) \wedge (y > 780) \wedge (dv = 0)$	$(x < 380) \wedge (y > 780)$
2	$(x < 380) \wedge (y < 160) \wedge (dv = 0)$	$(x < 380) \wedge (y < 160)$
3	$(380 \leq x \leq 690) \wedge (y > 780) \wedge (dv = 0)$	$(380 \leq x \leq 690) \wedge (y > 780)$
4	$(380 \leq x \leq 690) \wedge (y < 160) \wedge (dv = 0)$	$(380 \leq x \leq 690) \wedge (y < 160)$
5	$(x > 690) \wedge (y > 780) \wedge (dv = 0)$	$(x > 690) \wedge (y > 780)$
6	$(x > 690) \wedge (y < 160) \wedge (dv = 0)$	$(x > 690) \wedge (y < 160) \wedge (og \neq -2 \wedge og \neq -1)$

The next part of the paper mainly explains how to realize the above process automatically. It focuses on the key steps of value flow graph construction, semantic analysis, and expression analysis.

4. Program Understanding and Requirement Validation Based on VFG

The method is an automatic program analysis algorithm based on accurate value flow graph (VFG). Value flow graph is a kind of data flow representation of program, which can be obtained automatically by semantic analysis based on memory model. Based on the VFG, a variety of static analysis algorithms can be executed to get the program semantics of structural representation, and then confirm the requirements.

The workflow of the algorithm is shown in Figure 3. First, the source code is transformed into a semantically equivalent control flow automaton (CFA) with the help of the existing static analysis framework, and the accurate VFG is further generated according to the memory read-write information provided by CFA. After getting VFG, the expression analysis algorithm is executed to get the symbolic expression relationship between variables. Finally, a module summary is generated according to the expression information and saved in the source file.

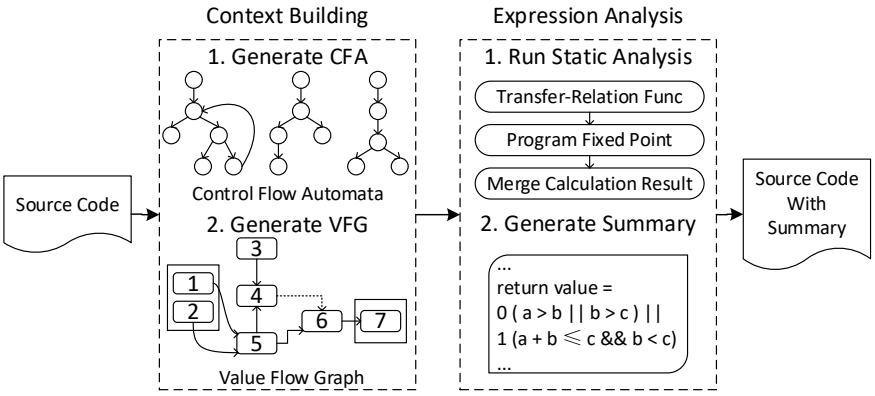


Fig. 3: Flow chart of the algorithm

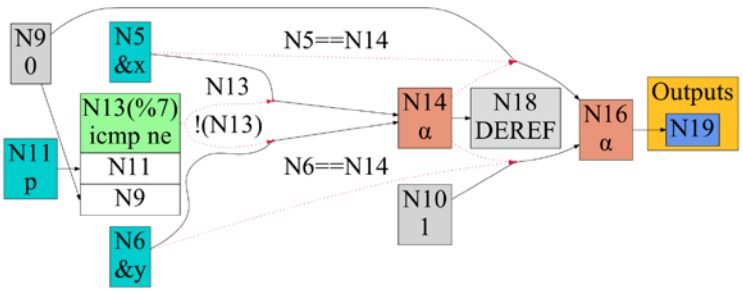
In this chapter, we will introduce the reason for using accurate value flow graph and its definition and construction method in Section 4.1. In Section 4.2, we will introduce the expression analysis algorithm based on VFG. Finally, we will analyze the specific application scenario of the summary in Section 4.3.

4.1. Definition and Construction of Accurate Value Flow Graph

In the traditional data flow analysis method based on control flow graph, the calculation results of the same variable under different paths are different. In order to ensure the correctness of the algorithm, this kind of algorithm usually defines the state merging function at the intersection node, which approximately combines the possible values of variables in specific use. But this will ignore the path information and lead to the lack of analysis accuracy. The accurate VFG combines variable value, data dependency and control dependency through memory behavior analysis and pointer alias operation analysis. It is path sensitive and can be used to solve the problem of path loss in the process of state calculation.

```
1.  int main() {
2.    int x = 0, y = 1;
3.    int *a;
4.    int p;
5.    if (p)
6.      a = &x;
7.    else
8.      a = &y;
9.    return *a;
10. }
```

(a)



(b)

Fig. 4: Code example and the VFG generated from it

Taking the code shown in Figure 4(a) as an example, if path sensitive data flow analysis is used, it can be expressed as: a ←{&x, p ≠ 0, (&y, p = 0)}. In this way, we can take the data flow • ← • as the edge, and the data carried on the data flow as the point of the VFG. The constructed VFG is shown in Figure 4(b).

The VFG is divided into different modules according to functions. Each *Module* = (*N*, *E*) is a digraph, where *N* is the node set, and each node represents a variable or constant in the program. Node $n \in N$ can also

have conditions, indicating control dependence. $E \subseteq N \times N$ is a set of directed edges, indicating the flow direction of values. Edge $e \in E$ can be attached with conditions that indicate data dependence.

In this paper, we divide VFG nodes into the following categories according to their functions:

ConstNode: represents a constant in the program, such as integer constant, string constant, function pointer constant, address constant of global variable, etc., which is represented by < *C* >, *C* is literal constant. As shown in Fig. 4(b), node N_9 can be represented as < 0 >.

StartNode: indicates the data initialization node in the value flow graph, such as the initial value of variables, parameters, and return value of function calls in the program. According to the definition, the StartNode has no predecessor node in the VFG, and the node is not a constant. The initial node is represented by < *ap* > where *ap* is the access path to the memory location of the corresponding data. As shown in Fig. 4, node N_{11} can be represented as < p >.

CopyNode: refers to the copy relationship of value. Its value is copied to the predecessor node, which is represented by < *n* >. Here *n* is the node in the VFG. As shown in Figure 4(b), node N_{19} can be represented as < N_{16} >.

OperatorNode: indicates the calculation result by the operator and its input. The operation node takes its predecessor as the input of the operation. OperationNode is represented by < op, op_1, ... , op_n >, where *op* is operator, op_1, ... , op_n is the operand. Here, the operand is the predecessor of the OperatorNode. In Figure 4(b), node N_{13} can be expressed as < ≠, N_{11}, N_9 >.

JoinNode: the JoinNode has multiple predecessor nodes. The value of this node is one of the predecessor nodes. JoinNode are represented by < (N_1, C_1) , ... , (N_i, C_i) >, where N_i is the predecessor node of JoinNode and C_i is the data dependency, indicating that when C_i is true, the value of JoinNode is N_i. In Figure 4(b), node N_{14} can be expressed as < (N_5, N_{13}), (N_6, ¬N_{13}) >.

For each module *M*, two special node sets In(*M*) and Out(*M*) are defined to represent the input node set and output node set of the module respectively. The input node includes function parameters and global variables, and its type is always StartNode. The output node includes the return value of the function and the global variables, and its type is always CopyNode.

To facilitate the subsequent description of the algorithm, we define the following functions: define the function *literalVal*::*ConstNode* → *Value* to take the literal value of the ConstNode; define the function *symbolic*::*StartNode* → *Value* to take the symbolic value of the StartNode; define the function *pcond*::*JoinNode* → *Condition* to take the data dependency of the JoinNode.

4.2. Expression Analysis Algorithm

Define *ValueCase* as a tuple (*Value*, *Condition*), where *Value* is the possible value of a node on the VFG. This value can be either a specific constant value, such as 123, "abc", or a symbolic expression, such as "a + 3"; *Condition* is a Boolean expression, which means that when the *Condition* of a node on the VFG is true, its value is *Value*. Define function *val*::*ValueCase* → *Value* to take the *Value* in *ValueCase*; function *cond*::*ValueCase* → *Condition* to take the value of *Condition* in *ValueCase*. Define abstract function *T*::*Node* → *ValueSet* as a mapping from the node of VFG to the abstract domain, where *ValueSet* is the set of *ValueCase*. The expression analysis algorithm based on the VFG can be expressed as shown in Figure 5.

Input: *nodes*: current value flow graph's node set

```
1: blockSet ← ∅
2: pending ← {n | n ∈ nodes, n is ConstNode or StartNode }
3: while pending ≠ ∅ do
4:     cur ← peek(pending)
5:     valueSet ← updateNodeValue(cur)
6:     if valueSet = ∅ then
7:         blockSet ← blockSet ∪ cur
8:     else if valueSet is an update for cur then
9:         pending ← pending ∪ successor(cur)
10:        move each node from blockSet to pending which is unblocked due to valueSet
11:    end if
12: end while
```

Fig. 5: Algorithm on Value Flow Graph

In the algorithm, the function *updateNodeValue*::*Node* → *ValueSet* represents the calculation value of each node in the iterative calculation process. For different types of nodes, we define different calculation methods, and the specific calculation methods are shown in Table 2.

Table. 2: Value updating algorithm for all kinds of nodes

NodeType	UpdatedValue
ConstNode	{ (*literalVal*(*cur*), true) }
StartNode	{ (*symbolicVal*(*cur*), true) }
CopyNode	*updateNodeValue*(*pred* (*cur*))
OperatorNode	{ (*operatorValue*(*opr*(*cur*), *val*(opd_1), ... , *val*(opd_n)), *cond*(opd_1) ∧ ... ∧ *cond*(opd_n)) | $opd_i \in T$ (pre_i), $pre_i \leftarrow pred_i(cur)$ }
JoinNode (Loop)	{ (μ, true) | $\mu \leftarrow \mu \cup val(pred(cur))$ }
JoinNode (Normal)	{ (*val*(pre_i), *cond*(pre_i) ∧ *pcond*(pre_i)) | $pre_i \leftarrow pred_i(cur)$ }

This algorithm is a kind of VFG analysis algorithm, it defines two sets of *pending* and *blockSet*, which are respectively used to store the nodes to be analyzed and the blocked nodes in the analysis algorithm.

When the algorithm is initialized, set the *blockset* as null, and put all StartNode and ConstNode into the *pending* set. During each iteration, a node is taken from the *pending* set. If its predecessor or data dependency is not calculated, the node is put into the *pending* set; otherwise, the node is calculated according to Tab. 2. If the node has never been calculated before, or the calculation result is different from the previous one, put all the subsequent nodes into the *pending* set, and check whether there are nodes in the *blockset* that depend on this calculation result can be calculated. If there are any, put the corresponding nodes into the *pending* set again. Keep iteration until all nodes are calculated and reach the fixed point, the iteration ends.

4.3. Program Understanding and Requirement Validation

Taking the transmission gear control code (Figure 2) described in Chapter 3 as an example, the corresponding value flow diagram is shown in Figure 6.

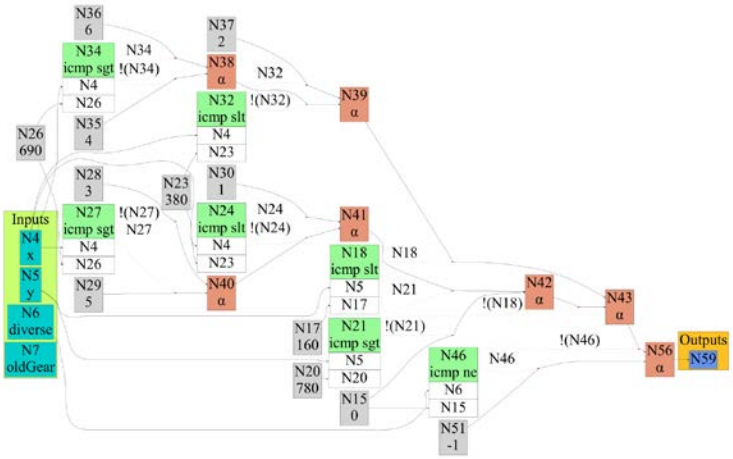


Fig. 6: The VFG generated according to the code in Fig. 2

The above algorithm is applied to calculate all nodes, and the results of some nodes are shown in Table 3.

Table. 3: Calculation results of some nodes on VFG

Node	T(Node)
N_4	{ (x, true) }
N_{26}	{ (690, true) }
N_{34}	{ (x > 690, true) }
N_{38}	{ (4, x ≤ 690), (6, x > 690) }
N_{41}	{ (1, x < 380), (3, 380 ≤ x ≤ 690), (5, x > 690) }
N_{42}	{ (0, y ≤ 780), (1, x < 380 ∧ y > 780), (3, 380 ≤ x ≤ 690 ∧ y > 780), (5, x > 690 ∧ y > 780) }
N_{56}	{ (-1, dv ≠ 0), (0, 160 ≤ y ≤780 ∧ dv = 0), (1, x < 380 ∧ y > 780 ∧ dv = 0), (2, x < 380 ∧ y < 160 ∧ dv = 0), (3, 380 ≤ x ≤ 690 ∧ y > 780 ∧ dv = 0), (4, 380 ≤ x ≤ 690 ∧ y < 160 ∧ dv = 0), (5, x > 690 ∧ y > 780 ∧ dv = 0), (6, x > 690 ∧ y < 160 ∧ dv = 0) }

Since In(*M*) describes all possible read-in of module *M* and Out(*M*) describes all possible outputs, the summary can be constructed according to the above. As shown in Figure 6, node N_{59} in Out(*M*) is a CopyNode, its result is same as N56, so the final module summary is shown in columns 1 and 2 of Table 1.

The summary clearly describes the corresponding relationship between the input and output of modules under different branch conditions, so as to improve the efficiency of program understanding and requirements validation. On the other hand, while running static analysis, we can also apply the checking rules on it to implement the code inspection. A typical application is to check the value (expression) of each node under different program paths, such as zero division exception, null pointer dereference, multiple memory release, etc., so as to improve the code quality.

5. Evaluation and Results

5.1. Experimental Design

Considering that the process of generating accurate VFG is very complicated, the implementation of this algorithm is based on the Tsmart framework. It uses static analysis to generate control flow automata and accurate VFG. At the same time, it is configurable and can easily obtain the program context information. The framework diagram of the tool is shown in Figure 7:

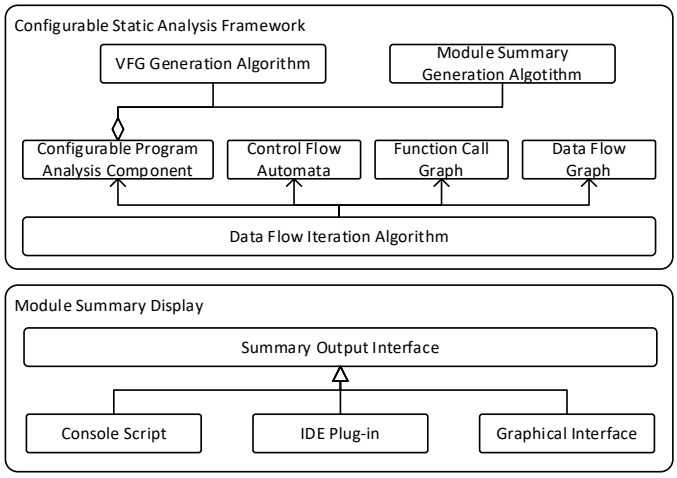


Fig. 7: Frame diagram of the tool

This paper evaluates the utility of the algorithm (tool) from two aspects: objective and subjective. Objective evaluation focuses on the completeness and accuracy of the algorithm. Specifically, we examine the quality of summary generated by the algorithm, while subjective evaluation focuses on the usability and practicability of tool in a specific production environment.

5.2. Objective Evaluation

The objective evaluation method mainly includes the following steps: (1) select 8 representative code segments in mathematical calculation, computer application, industrial field and embedded field as experimental objects (see Table 4 for details); (2) generate corresponding function summaries with the tools; (3) determine the gap between the summaries generated by the tools and the actual requirements.

Table. 4: Various types of codes we selected

Code Type	Selected Code	File Name	Lines of Code
Mathematical calculation	Triangle Determination	triangle.c	24
	Absolute Value Calculation	abs.c	11
Computer application	Weighted Filtering Algorithm	filter.c	19
	Checksum Algorithm	checksum.c	22
Industrial Field	Vehicle Control System Code	CDL_ARC429.c	124
	Aero-Engine Control System Code	ISP_TOOL.c	178
Embedded Field	Code with Goto Statement	msp430-decode.c	57
	Code with Function Pointer	xen-ops.c	36

Because the summary result is related to the conditional branches of the program, this paper compares the generated results under each branch. At the same time, record the time and space cost of the tool, including the summary generation time, the memory usage and the generated VFG size, as shown in Table 5.

Table. 5: Comparision between automatic summary generation and manual summary generation

Code	Requirement	Generated Summary				Accuracy
	Branch Num	Branch Num	Time/s	RAM/MB	VFG Size/KB	
Triangle Determination	4	4	3	134	9	1.0
Absolute Value Calculation	2	2	2	122	2	1.0
Filtering Algorithm	1	1	2	128	4	1.0
Checksum Algorithm	1	1	2	145	12	1.0
Vehicle Control System	12	12	5	167	53	1.0
Aero-Engine Control System	15	15	6	171	61	1.0
Code with Goto Statement	8	8	4	137	34	1.0
Code with Function Pointer	3	3	3	132	6	1.0

Experiments show that the tool can completely generate summaries of different types of codes, and the number of conditional branches in summary is consistent with the requirements. The resources usage are relatively small, the memory consumption is less than 200MB, and the generated VFG is less than 1MB. The tool can generate function summary quickly, and takes less than 1 minute for code with hundred lines.

5.3. Subjective Evaluation

To ensure the fairness and accuracy of the subjective evaluation, a total of 30 students with equivalent conditions are selected from the school to participate in the experiment. They all meet the following conditions: (1) they all have more than 3 years of programming experience in C language and passed the school's programming level test; (2) they have a similar understanding of the background of the given code.

The experimental method consists of the following steps: (1) divide 30 students into two groups called source code group and summary group for a control experiment, and provide them with 8 source code (or source code with summary) as shown in Table 4 respectively; (2) ask each student to read the code, then question them the code function, and record the complete time T1; (3) ask each student to analyze the output of the function under different input conditions, and record Time T2; (4) count and compare the time usage T1 and T2 by the two groups on answering questions; (5) after the source group students finished questions, provide them a summary, then ask two groups of students about their opinion on the summary.

As shown in Tab. 6, the average time taken by two groups is T1 and T2. By comparison, when code is short (within 10 lines) and the logic is not complex, using summary is not significantly helpful. However, when faced with dozens or even hundreds of lines of code, using summaries can significantly reduce the time. In the experimental sample with goto statement, 60.5% of the program understanding time can be saved.

It is known through the experiment that this tool can significantly improve the efficiency of the user's understanding of the code, and can help the user quickly extract the program semantics and validate the requirements. At the same time, with the increasing complexity of code branches, the advantage of using tools for program understanding and requirement validation will become more greater.

Table. 6: Time for each group to understand and answer each type of code (second)

Code	Souce Code Group		Summary Group	
	T1/s	T2/s	T1/s	T2/s
Triangle Determination	45	32	37	14
Absolute Value Calculation	5	13	11	14
Filtering Algorithm	61	15	27	15
Checksum Algorithm	97	43	54	38
Vehicle Control System	294	86	103	74
Aero-Engine Control System	318	92	114	91
Code with Goto Statement	195	48	64	31
Code with Function Pointer	82	33	40	29

In step 5, we designed several questions, and the results are shown in Tab. 7. The participants were positive about the automatic generation of summary tool, and thought that the summary was helpful for understanding the code. Regarding whether the summary can help analyze the impact scope of changes and software maintenance, some students hold a wait-and-see attitude. They believe that the generated summary

is a function level summary, while the change impact scope analysis and software maintenance may involve global code, which needs to be further analyzed in combination with function call and other information.

Table. 7: Questionnaire design and response

Question Setting	Agreed Number	Negative Number	N/A
Is summary helpful for your program understanding?	28	2	0
Is summary helpful for change impact scope analysis?	25	4	1
Is summary helpful for software maintenace?	22	7	1

6. Conclusion and Future Works

This method relies on the accurate generation of accurate value flow graph. Currently, there are relatively few tools available, and it consumes much time and space in the face of large programs. This will have an impact on the results of this tool. One possible solution is to optimize the original algorithm of VFG, and optimize the results for loops and arrays.

On the other hand, in order to ensure the convergence and time cost of the algorithm, symbol value μ is introduced into the algorithm when processing the loop, and the result of the algorithm is a symbol expression with μ, which will lead to a certain loss of precision. In the future, loop invariants can be used to further improve the analysis accuracy.

In this paper, we present a novel approach to program understanding algorithm and requirement validation. The essense of our work is value flow graph, which is a representation of program semantics. Although the method proposed in this paper has certain limitations, the results of this study is quite promising and provides useful ideas and guidance for the design and implementation of program understanding, variable expression analysis and program change impact analysis tools.

7. References

[1] Wagner S, Abdulkhaleq A, Bogicevic I, et al. How are functionally similar code clones syntactically different? An empirical study and a benchmark[J]. PeerJ Computer Science, 2016, 2: e49.

[2] Wallace D R, Fujii R U. Software verification and validation: an overview[J]. Ieee Software, 1989, 6(3): 10-17.

[3] Wagner S. Software product quality control[M]. Heidelberg: Springer, 2013.

[4] Ramler R, Biffl S, Grünbacher P. Value-based management of software testing[M]//Value-based software engineering. Springer, Berlin, Heidelberg, 2006: 225-244.

[5] Ko A J, DeLine R, Venolia G. Information needs in collocated software development teams[C]//Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007: 344-353.

[6] Murphy G C, Kersten M, Findlater L. How are Java software developers using the Elipse IDE?[J]. IEEE software, 2006, 23(4): 76-83.

[7] Corbi T A. Program understanding: Challenge for the 1990s[J]. IBM Systems Journal, 1989, 28(2): 294-306.

[8] Boysen J P. Factors affecting computer program comprehension[J]. 1979.

[9] Sackman H, Erikson W J, Grant E E. Exploratory experimental studies comparing online and offline programing performance[R]. SYSTEM DEVELOPMENT CORP SANTA MONICA CA, 1966.

[10] Binkley D W, Gallagher K B. Program slicing[M]//Advances in Computers. Elsevier, 1996, 43: 1-50.

[11] Sulír M, Porubän J. Labeling source code with metadata: A survey and taxonomy[C]//2017 Federated Conference on Computer Science and Information Systems (FedCSIS). IEEE, 2017: 721-729.

[12] B. W. Boehm. Software Engineering[J]. IEEE Trans. Comput. 1976, 25(12): 1226-1241.

[13] Maalej W, Happel H J. Can development work describe itself?[C]//2010 7th IEEE working conference on mining software repositories (MSR 2010). IEEE, 2010: 191-200.

[14] Maalej W, Tiarks R, Roehm T, et al. On the comprehension of program comprehension[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2014, 23(4): 31.

[15] Weiser M. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method[J]. PhD thesis, University of Michigan, 1979.

[16] Korel B, Laski J. Dynamic program slicing[J]. Information processing letters, 1988, 29(3): 155-163.

[17] Gupta R, Soffa M L, Howard J. Hybrid slicing: integrating dynamic information with static analysis[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 1997, 6(4): 370-397.

[18] Schugerl P, Rilling J, Charland P. Beyond generated software documentation—A web 2.0 perspective[C]//2009 IEEE International Conference on Software Maintenance. IEEE, 2009: 547-550.

[19] Karrer T, Krämer J P, Diehl J, et al. Stacksplorer: call graph navigation helps increasing code maintenance efficiency[C]//Proceedings of the 24th annual ACM symposium on User interface software and technology. ACM, 2011: 217-224.

[20] Beck F, Gulan S, Biegel B, et al. Regviz: Visual debugging of regular expressions[C]//Companion Proceedings of the 36th International Conference on Software Engineering. ACM, 2014: 504-507.

[21] Beyer D, Fararooy A. DepDigger: A tool for detecting complex low-level dependencies[C]//2010 IEEE 18th International Conference on Program Comprehension. IEEE, 2010: 40-41.

[22] Beck F, Moseler O, Diehl S, et al. In situ understanding of performance bottlenecks through visually augmented code[C]//2013 21st International Conference on Program Comprehension (ICPC). IEEE, 2013: 63-72.

[23] Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization[C]//Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2. ACM, 2010: 223-226.

[24] Cornelissen B, Holten D, Zaidman A, et al. Understanding execution traces using massive sequence and circular bundle views[C]//15th IEEE International Conference on Program Comprehension (ICPC'07). IEEE, 2007: 49-58.

[25] Fittkau F, Waller J, Wulf C, et al. Live trace visualization for comprehending large software landscapes: The ExplorViz approach[C]//2013 First IEEE Working Conference on Software Visualization (VISSOFT). IEEE, 2013: 1-4.